KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

FPGA Design for Large Flow Detection in High-Speed Networks

Arish Sateesan

Supervisors:
Prof. dr. ir. Nele Mentens
Dr. ing. Jo Vliegen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

January 2024

FPGA Design for Large Flow Detection in High-Speed Networks

Arish SATEESAN

Examination committee:
Prof. dr. ir. Albert Van Bael, chair
Prof. dr. ir. Nele Mentens, supervisor
Dr. ing. Jo Vliegen, supervisor
Prof. dr. ir. Toon Goedemé
Prof. dr. Josep Balasch
Ir. Marc Moeskops
(Belden, Belgium)
Prof. dr. Dirk Koch
(Heidelberg University, Germany; University of Manchester, UK)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD)

January 2024

© 2024 KU Leuven – Faculty of Engineering Technology
Uitgegeven in eigen beheer, Arish Sateesan, Wetenschapspark 27 bus 15152, 3590 Diepenbeek (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

Preface

In the pursuit of knowledge and the relentless passion for discovery, I am humbled and privileged to present this doctoral thesis. This work represents the culmination of four years of rigorous research and exploration to advance the frontiers of FPGA Design for Network Security. The journey through this academic pursuit has been an exciting and transformative experience, pushing the boundaries of my understanding and reshaping my perception of the world.

Throughout this endeavor, I have been fortunate to receive guidance, support, and encouragement from numerous individuals who have played pivotal roles in shaping my intellectual growth. First and foremost, I would like to express my sincere gratitude to my esteemed supervisor, Prof. Nele Mentens, for granting me the invaluable opportunity to pursue my doctoral studies under her guidance at the prestigious KU Leuven. Her relentless support, profound expertise, and insightful feedback have been instrumental in shaping the trajectory of this work. I am sincerely thankful for her mentorship and for providing me with a multitude of opportunities, including interactions with accomplished researchers, access to state-of-the-art technology, and participation in research events and workshops. I believe a tranquil mind is a researcher's greatest asset, and I cannot express enough gratitude to her for keeping me in my comfort zone.

I extend my heartfelt thankfulness to my co-supervisor, Dr.ir. Jo Vliegen, for his invaluable guidance and insights throughout my doctoral studies. His profound expertise and encouraging demeanor have played a vital role in propelling my research to new heights. I would also like to thank Prof. Joan Daemen (Radboud University, The Netherlands) for his invaluable assistance in the initial stage of my Ph.D. in the design of a non-cryptographic hash function, which has been instrumental throughout my research. I express my sincere gratitude to Dr. Smitha KG (Nanyang Technological University, Singapore) for her guidance and mentorship during the early phase of my research career.

I wish to express my appreciation to my colleagues at ES&S for fostering a collaborative and enriching environment. Their discussions and debates have

expanded my horizons and pushed me to refine my ideas further.

I am thankful to all the members of my committee, Prof. Toon Goedemé, Mr. Marc Moeskops, Prof. Dirk Koch, and Prof. Josep Balasch, for their helpful feedback and suggestions. I am grateful for their time and expertise, which have helped me to improve my thesis.

I am indebted to KU Leuven, FWO, SNSF, and CIF for supporting my research financially. Their investment in academic research creates an environment that nurtures discovery and innovation, empowering future generations of scholars to explore unexplored territories. I wish to express my gratitude to the researchers from the Network Security Group at ETH Zurich for their collaboration on the research project ESCALATE (funded by FWO and SNSF), of which my thesis was a part.

Last but not least, I deeply express my gratitude to my friends and family for their love and support throughout my Ph.D. journey. They have been a source of strength and encouragement, and I am grateful for their willingness to listen to me talk about my research for hours on end. A special thanks go to my mother, Usha, and my wife, Akshara, for their boundless love, support, and sacrifices during various stages of my life. I would not be where I am today without their love, support, and sacrifices.

As I submit this thesis to the academic community, I am acutely aware of its limitations and recognize that knowledge is ever-evolving, with infinite possibilities for future exploration. I hope that this work will inspire further research and spark curiosity among fellow scholars, igniting new avenues of exploration.

Arish Sateesan

Abstract

We inhabit a world that is heavily immersed in digital connectivity, relying extensively on smart devices like smartphones, Internet of Things (IoT) devices, and personal computers. The surge in data rates and the growing number of devices connected to the Internet have led to a significant rise in security threats. Network security has become an increasingly challenging research area, particularly in the era of big data, where the frequency of new network threats, such as DDoS attacks, is on the rise. Detecting malicious network flows, particularly large flows or heavy-hitters, has gained significant importance in safeguarding network infrastructure. However, the high data rates pose a bottleneck for real-time processing of incoming data, necessitating the use of hardware acceleration techniques, such as FPGA-based acceleration in data centers.

This thesis aims to design and implement hardware-efficient algorithms and architectures for network security applications, with a specific focus on deploying large flow detection algorithms on FPGAs. The main challenges addressed include reducing memory usage, enhancing accuracy, improving throughput, and accommodating heterogeneity in terms of attack vectors. Memory reduction is vital in high data rate environments with limited on-chip memory. Moreover, maintaining high accuracy while limiting memory usage is crucial to prevent the misidentification of legitimate and malicious flows, which could have catastrophic consequences in network security. The proposed hardware architectures aim to achieve high operating frequency and throughput, aligning with the requirements of high-speed Ethernet. The architectures are designed to be heterogeneous, making them applicable not only to large-flow detection algorithms but also to other network security applications.

The contributions of this thesis lie in the design and implementation of hardware-efficient hash functions, lookup, and network flow measurement architectures, which serve as the fundamental building blocks of network security applications. Probabilistic lookup and measurement architectures

are particularly emphasized for their speed and memory efficiency compared to exact lookup and measurement architectures. To this end, we propose efficient probabilistic lookup and measurement architectures that strike a balance between memory usage, accuracy, and throughput. Hash functions are essential components in probabilistic architectures, and this thesis presents a set of hardware-efficient non-cryptographic hash functions tailored specifically for high-speed network security applications. Finally, this thesis proposes a detection framework capable of integrating multiple detection algorithms to handle a wide range of network attacks. The detection framework helps to efficiently deploy multiple detection algorithms as a single stack in commercial hardware. By addressing the challenges of memory reduction, accuracy, throughput, and heterogeneity, this work contributes to enhancing network security in the face of ever-evolving threats and high data rates.

Beknopte samenvatting

We leven in een wereld die sterk ondergedompeld is in digitale connectiviteit en grotendeels afhankelijk is van slimme apparaten zoals smartphones, Internet of Things (IoT)-apparaten en personal computers. De stijging van de datasnelheden en het groeiende aantal apparaten dat met internet is verbonden, hebben geleid tot een aanzienlijke toename van cyberaanvallen. Netwerkbeveiliging is een steeds uitdagender onderzoeksgebied geworden, zeker in tijden van big data, waar de frequentie van nieuwe netwerkaanvallen, zoals DDoS-aanvallen, toeneemt. Het detecteren van kwaadaardige netwerkstromen, met name grote stromen of heavy-hitters, is aanzienlijk belangrijker geworden bij het beveiligen van de netwerkinfrastructuur. De hoge datasnelheden vormen echter een uitdaging voor de realtime verwerking van binnenkomende gegevens, waardoor het gebruik van hardware-gebaseerde versnellingstechnieken noodzakelijk is. Hiervoor zijn op FPGA-gebaseerde oplossing voor handen in datacenters.

Dit proefschrift heeft tot doel hardware-efficiënte algoritmen en architecturen voor netwerkbeveiligingstoepassingen te ontwerpen en implementeren, met een specifieke focus op algoritmen voor detectie van grote stromen op FPGA's. De belangrijkste uitdagingen die worden aangepakt zijn onder meer het verminderen van het geheugengebruik, het verbeteren van de nauwkeurigheid, het verhogen van de doorvoer en het bewerktstelligen van heterogeniteit in termen van aanvalspatronen. Geheugenreductie is van cruciaal belang in omgevingen met een hoge datasnelheid en met slechts een beperkt geheugen op de chip. Bovendien is het handhaven van een hoge nauwkeurigheid en tegelijkertijd het beperken van het geheugengebruik cruciaal om de verkeerde identificatie van legitieme en kwaadaardige stromen te voorkomen. Dit verkeerde identificatie zou catastrofale gevolgen kunnen hebben voor de netwerkbeveiliging. De voorgestelde hardware-architecturen zijn gericht op het bereiken van een hoge verwerkingssnelheid en doorvoer, wat in lijn ligt met de eisen van high-speed Ethernet. De architecturen zijn ontworpen om heterogeen te zijn, waardoor ze niet alleen toepasbaar zijn op detectie-algoritmen voor grote stromen, maar ook op andere toepassingen in netwerkbeveiliging.

De bijdragen van dit proefschrift liggen in het ontwerp en de implementatie van hardware-efficiënte hash-functies, opzoek- en netwerkstroommetingsarchitecturen, die dienen als de fundamentele bouwstenen van netwerkbeveiligingstoepassingen. Er wordt vooral aandacht besteed aan probabilistische opzoek- en meetarchitecturen vanwege hun snelheid en geheugenefficiëntie in vergelijking met exacte opzoek- en meetarchitecturen. Daartoe stellen we efficiënte probabilistische opzoek- en meetarchitecturen voor die een evenwicht vinden tussen geheugengebruik, nauwkeurigheid en doorvoer. Hash-functies zijn essentiële componenten in probabilistische architecturen, en dit proefschrift presenteert een reeks hardware-efficiënte, niet-cryptografische hashfuncties die specifiek zijn ontworpen voor netwerkbeveiligingstoepassingen. Tenslotte wrodt in dit proefschrift een detectieframework voorgesteld dat meerdere detectie-algoritmen kan integreren om een breed scala aan netwerkaanvallen te kunnen afwenden. Dit detectieframework helpt bij het efficiënt inzetten van meerdere detectie-algoritmen als ware het één enkele stack in commerciële hardware. Door de uitdagingen van geheugenreductie, nauwkeurigheid, doorvoer en heterogeniteit aan te pakken, draagt dit werk bij aan het verbeteren van de netwerkbeveiliging in het licht van de steeds evoluerende bedreigingen en hoge datasnelheden.

List of Abbreviations

A-CM Approximate-Count Minimum.

AAE Average Absolute Error.

AFL Active Flow List.

AI Artificial Intelligence.

AMQ Approximate Membership Query.

APT Advanced Persistent Threats.

ARE Average Relative Error.

ARM Advanced RISC Machine.

ARPA Advanced Research Projects Agency.

ARPANET Advanced Research Projects Agency Network.

ASIC Application-Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

AXIS Advanced eXtensible Interface Stream.

bps bits per second.

BRAM Block RAM.

CAGR Compound Annual Growth Rate.

CAM Content Addressable Memory.

CDC Clock Domain Crossing.

CF Cuckoo Filter.

CHT Cuckoo Hash Table.

CIA Confidentiality, Integrity, and Availability.

CLB Configurable Logic Block.

CMI Control and Management Interface.

COVID-19 Coronavirus disease of 2019.

CPU Central Processing Unit.

CSMA/CD Carrier Sense Multiple Access/Collision Detection.

CUDA Compute Unified Device Architecture.

DDoS Distributed Denial of Service.

DDR Double Data Rate.

DHCP Dynamic Host Configuration Protocol.

DNS Domain Name System.

DoS Denial of Service.

DRAM Dynamic Random Access Memory.

DSP Digital Signal Processor.

EMQ Exact Membership Query.

FAC Fingerprint-Address Cell.

FCS Frame Check Sequence.

FF Flip Flop.

FIFO First In First Out.

flow ID flow Identifier.

FN False Negative.

FP False Positive.

FPGA Field Programmable Gate Array.

FPR False Positive Rate.

FSM Finite State Machine.

FT Flow Table.

FTP File Transfer Protocol.

GB Gigabyte.

GB/s Gigabyte per second.

GbE Gigabit Ethernet.

Gbps Giga bits per second.

GPIO General Purpose Input/Output.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

HBM High-Bandwidth Memory.

HDL Hardware Description Language.

HLS High Level Synthesis.

HNDL Harvest Now, Decrypt Later.

HSAC Hardware-oriented Simple Active Counter.

HT Hash Table.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

I2C Inter-Integrated Circuit.

ICMP Internet Control Message Protocol.

IEEE Institute of Electrical and Electronics Engineers.

IOB Input/Output Block.

IoT Internet of Things.

IP Intellectual Property.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

ISP Internet Service Provider.

KB Kilobyte.

Kb Kilobit.

Kbps Kilo bits per second.

LAN Local Area Network.

LFSR Linear Feedback Shift Register.

LLC Logical Link Control.

LOFT Low-Rate Overuse Flow Tracer.

LSB Least Significant Bit.

LUT Look-UP Table.

LVCMOS Low Voltage Complementary Metal Oxide Semiconductor.

LVDS Low-Voltage Differential Signaling.

MAC Media Access Control.

MB Megabyte.

Mb Megabit.

Mbps Mega bits per second.

MitM Man-in-the-Middle.

Mpps Million packets per second.

MSB Most Significant Bit.

MSS Maximum Segment Size.

MTU Maximum Transmitted Unit.

NC Non-Cryptographic.

NFA Non-deterministic Finite Automata.

NIC Network Interface Card.

NIST National Institute of Standards and Technology.

NRE Non-recurring Expense.

NTP Network Time Protocol.

OFD Overuse Flow Detector.

OSI Open Systems Interconnection.

P4 Programming Protocol-independent Packet Processors.

PATA Parallel Advanced Technology Attachment.

PBF Parallel Bloom Filter.

PCIe Peripheral Component Interconnect express.

PDU Protocol Data Unit.

PE Processing Element.

PHT Peacock Hash Table.

PISA Protocol-independent switch architecture.

PowerPC Performance Optimization With Enhanced RISC–Performance Computing.

pps packets per second.

PSL Probe Sequence Length.

QSFP Quad Small Form factor Pluggable.

RAM Random Access Memory.

RMT Reconfigurable Match action Table.

rps requests per second.

RTL Register Transfer Level.

SBF Standard Bloom Filter.

SHA Secure Hash Algorithm.

SLR Super Logic Region.

SME Small and Medium-sized enterprises.

SMTP Simple Mail Transfer Protocol.

SNMP Simple Network Management Protocol.

SoC System-on-Chip.

SPArch Sketch-based Pseudo-associative-array Architecture.

SPI Serial Peripheral Interface.

SRAM Static Random Access Memory.

SSTL Stub Series Terminated Logic.

SYN Synchronize.

TB Terabyte.

TB/s Terabyte per second.

TbE Terabit Ethernet.

Tbps Tera bits per second.

TCAM Ternary Content Addressable Memory.

TCP/IP Transmission Control Protocol/Internet Protocol.

TN True Negative.

TP True Positive.

TTL Time To Live.

TTLT Time-To-Live Timestamp.

UART Universal Asynchronous Receiver-Transmitter.

UDP User Datagram Protocol.

VNx Vitis Network example.

VXLAN Virtual Extensible Local Area Network.

XUP Xilinx University Program.

Contents

Abstract iii

Beknopte samenvatting v

List of Abbreviations xii

Contents xiii

List of Figures xvii

List of Tables xxiii

1 Introduction 1
- 1.1 The path to a connected world . . . 1
- 1.2 The threats in the connected world and the importance of security 3
- 1.3 The need for hardware acceleration and hardware-based security in networking . . . 7
- 1.4 Thesis objectives and outline . . . 9

2 Background of network security, large flow detection, and FPGA 13
- 2.1 Introduction to FPGAs . . . 13
  - 2.1.1 FPGA architecture . . . 14
  - 2.1.2 FPGA connectivity . . . 18
  - 2.1.3 FPGA design flow . . . 19
  - 2.1.4 FPGA versus other hardware platforms . . . 20
  - 2.1.5 Target FPGA platform in this thesis: Alveo data center accelerator card . . . 23
- 2.2 Basics of Computer networks . . . 25
  - 2.2.1 Network Protocols . . . 25
  - 2.2.2 OSI and TCP/IP reference models . . . 26
  - 2.2.3 Ethernet . . . 29

2.3 Network security . . . 33
2.3.1 Information security, cyber security, and network security 33
2.3.2 Types of network security . . . 35
2.3.3 Threats in network security . . . 36
2.4 Distributed denial of service (DDoS) attacks . . . 38
2.4.1 Classes of DDoS attacks . . . 40
2.4.2 Types of DDoS attacks . . . 41
2.4.3 Defending DDoS attacks: Large flow detection . . . 42
2.5 Large flow detection . . . 43
2.5.1 Existing research: algorithmic approaches in large flow detection . . . 44
2.5.2 Research challenges in large flow detection . . . 46
2.6 Existing research and challenges in implementing large flow detection algorithms in FPGAs . . . 48

3 Hardware-efficient non-cryptographic hashing for networking 55
3.1 Introduction to hash functions . . . 56
3.1.1 Properties of hash functions . . . 56
3.1.2 Cryptographic and non-cryptographic hash functions . . . 57
3.1.3 Applications of hash functions . . . 58
3.1.4 Hash functions for network security applications . . . 59
3.2 Related work on non-cryptographic hash functions . . . 60
3.3 Design and analysis of Xoodoo-NC . . . 63
3.3.1 Avalanche behavior of Xoodoo-NC . . . 64
3.3.2 Hardware evaluation of Xoodoo-NC . . . 66
3.4 Implementation and analysis of cryptographic cipher-based NC hash functions . . . 68
3.4.1 Algorithmic details of the lightweight cryptographic ciphers 69
3.4.2 Analysis of avalanche properties . . . 75
3.4.3 Hardware Evaluation . . . 78
3.5 Conclusion . . . 85

4 Fast lookup architectures on FPGA and the analysis of their hardware-compatibility 87
4.1 Introduction . . . 88
4.2 Optimized Bloom filter architectures for lookups . . . 91
4.2.1 Bloom filter . . . 91
4.2.2 Related work . . . 94
4.2.3 Bloom filter algorithms for analysis . . . 96
4.2.4 Experimental setting and basis of comparison . . . 99
4.2.5 Hardware architectures . . . 101
4.2.6 Hardware evaluation . . . 104

4.3 A thorough analysis of the hardware friendliness of AMQ architectures . . . 110
4.3.1 An insight into various AMQ data structures . . . 110
4.3.2 Hardware architectures . . . 122
4.3.3 Hardware evaluation . . . 124
4.4 Conclusion . . . 132

5 Designing network flow measurement architectures for FPGAs 135
5.1 Network flow measurement and its challenges . . . 136
5.2 Count-Min Sketch . . . 142
5.3 Design and analysis of A-CM Skech . . . 144
5.3.1 Background of approximate counters . . . 145
5.3.2 Algorithms . . . 146
5.3.3 Hardware architecture . . . 148
5.3.4 Evaluation and Results . . . 151
5.4 Design and analysis of SPArch . . . 157
5.4.1 Preliminaries . . . 158
5.4.2 Architecture of SPArch . . . 161
5.4.3 Analysis of SPArch . . . 168
5.4.4 Organizing counters in multiple buckets . . . 177
5.4.5 Hardware architecture of SPArch . . . 183
5.4.6 Hardware evaluation . . . 184
5.5 Conclusion . . . 187

6 Towards a universal large flow detection framework on FPGA for Terabit Ethernet 191
6.1 Introduction . . . 192
6.2 Challenges in the design of a comprehensive detection system . 193
6.2.1 The need for dedicated algorithms for comprehensive flow monitoring . . . 193
6.2.2 Challenges in deploying individual detection algorithms on hardware . . . 195
6.3 Background on universal flow monitoring . . . 195
6.4 Proposed detection framework - FLARE . . . 197
6.4.1 Parser and Network filter . . . 198
6.4.2 Detection framework . . . 199
6.4.3 Blacklist module . . . 201
6.4.4 Skid buffer . . . 203
6.5 Proof of concept . . . 204
6.5.1 Experimental setup . . . 204
6.5.2 Implementation details . . . 205
6.5.3 Hardware implementation of LOFT . . . 206
6.5.4 Implementation results . . . 213

6.6 Conclusion 214

7 Conclusion **215**
7.1 Summary of contributions 215
7.2 Future work 217

8 Valorization **221**
8.1 Introduction 221
8.2 Market Dynamics 225
8.3 Valorization potential and commercialization of the research . . 225
8.3.1 Summary of research results 225
8.3.2 What does this research offer to the market? 226
8.3.3 Target audience 229
8.3.4 Exploitation strategies 229
8.4 Economic and social value 233

Bibliography **235**

Curriculum Vitae **259**

List of publications **261**

List of Figures

1.1 The first bipolar junction transistor; a small step for technology, a quantum leap for human innovation. (Source: [28]) 2
1.2 Examples of impactful standards, services and applications in the history of the Internet 3
1.3 Internet speeds and speed standards over the years 4
1.4 Impact of COVID-19 on cybercrime (Source: [14]) 5
1.5 Ten biggest data breaches in history (Source: [19]) 6
1.6 Median cost of all cyber attacks in Belgium in the recent years (Data: [27]) 7
1.7 Overview of chapters 10

2.1 FPGA architecture (Source: [125]) 15
2.2 A CLB block of Ultrascale FPGA 16
2.3 DSP48E2 slice (Source: [248]) 16
2.4 FPGA design flow 19
2.5 Performance comparison of CPU, GPU, ASIC, and FPGA for data acceleration (Source: [22]) 22
2.6 Fixed function ASIC switch versus PISA switch (Source: [31]) . 24
2.7 Xilinx Alveo U250 floorplan 24
2.8 OSI reference model versus TCP/IP model 27
2.9 Structure of an Ethernet packet 30
2.10 Data flow through the layers 31
2.11 5-tuple of an IPv4 packet 32
2.12 Information security, cyber security, and network security . . . 34
2.13 Layers of network security 35
2.14 DDoS attack 39
2.15 Representation of a typical Large flow detection system 44
2.16 Algorthmic approaches in large flow detection 45

3.1 Change in output with respect to the change in input of a hash function . . . 56
3.2 Splitting a single hash function to generate hash values . . . 62
3.3 Graphical representation of the Xoodoo state . . . 64
3.4 Xoodoo-NC block diagram . . . 67
3.5 Xoodoo-NC Round function . . . 68
3.6 Functional representation of a single round of the eight chosen ciphers: SPECK, Pyjamask, GIFT, AES, SKINNY, PHOTON, Sparkle, and TinyJAMBU. . . . 70
3.7 The avalanche metrics for SPECK-NC, Pyjamask-NC, GIFT-NC, AES-NC, SKINNY-NC, PHOTON-NC, Sparkle-NC, and TinyJAMBU-NC respectively. The areas show the obtained metric while the horizontal lines indicate the target. The legend indicates which colors are the Avalanche Dependence (D_{av}), the Avalanche Weight (w_{av}), and the Avalanche Entropy (H_{av}). . . . 76
3.8 General hardware setup of hash implementation . . . 79
3.9 Timing analysis of SPECK-NC . . . 80
3.10 Timing analysis of Pyjamask-NC . . . 80
3.11 Timing analysis of GIFT-NC . . . 80
3.12 Timing analysis of AES-NC . . . 80
3.13 Timing analysis of SKINNY-NC . . . 81
3.14 Timing analysis of PHOTON-NC . . . 81
3.15 Timing analysis of Sparkle-NC . . . 81
3.16 Timing analysis of TinyJAMBU-NC . . . 81
3.17 Throughput and latency with respect to resource utilization . . . 83
3.18 Maximum frequency of the ASIC implementations . . . 85
3.19 ASIC area at maximum frequency . . . 85
3.20 Area of the ASIC implementations at 100 MHz . . . 86

4.1 Role of an AMQ data structure for lookup applications . . . 89
4.2 Role of AMQ data structures for lookups in network attack detection systems . . . 89
4.3 Standard Bloom filter (SBF) representation . . . 92
4.4 The false positive rate of Bloom filter against varying number of hash functions . . . 93
4.5 Bloom-1 representation . . . 96
4.6 Parallel Bloom filter logic (a) using Bi-SBF and (b) using Uni-SBF 98
4.7 Maximum operating frequency and resource utilization of the CAM directly implemented with the Xilinx tools (referred to as Xilinx CAM IP) and the speed-optimized CAM architecture (Custom Modified CAM IP) . . . 100
4.8 FNV-1a - serialized hardware architecture block diagram . . . 101
4.9 FNV-1a - Pipelined hardware architecture block diagram . . . 102

4.10 Bloom-1 hardware architecture 104
4.11 Parallel Bloom Filter hardware architecture 104
4.12 Resource Utilization of Bloom-1: Logic resources and DSP units 105
4.13 Maximum Operating Frequency and Delay of Bloom-1 106
4.14 Dynamic Power Consumption of Bloom-1 (Using FNV & Xoodoo-NC), and CAM 108
4.15 Functionalities of lookup and key-value store data structures . . 111
4.16 Representation of hash table. (a) A hash table with no collision during an addition, (b) Chaining in hash table during a collision. 112
4.17 Illustration of Robinhood hashing during a collision. (a) an incoming key c is mapped to the same slot where key a is stored, indicating a collision. As both c and a has equal PSL of 0, c probes the next slot. PSL of c is increased to 1., (b) PSL of c > PSL of b, and swapped. b probes the next slot, and PSL of b becomes 1., (c) b finds an empty slot and is added to that slot. 113
4.18 Illustration of 2-choice and 2-left hashing during a collision. (a) 2-choice hashing: the incoming key f is inserted into the bucket having the lowest load, (b) 2-left hashing: the incoming key f is inserted into the left table as there is equal load in both buckets. 114
4.19 Illustration of Cuckoo hashing during a collision. (a) Incoming key f is hashed to two buckets, and none are empty., (b) f is swapped with the key c from a randomly chosen cell. c rehashed to find the indexed locations of c., (c) A cell in the hashed bucket is empty and c is placed in that cell. 115
4.20 Illustration of Peacock hashing during a collision. (a) An incoming key c is hashed to the largest table T_1, and there is a collision., (b) c is hashed to the subsequent table T_2, and the mapped cell is empty. c is added to the empty cell. 116
4.21 Bloom filters 117
4.22 A representation of Cuckoo filter 119
4.23 Memory access cycles for hash tables 129
4.24 Memory access cycles of Cuckoo and Bloom filters 129
4.25 Insertion throughput 130
4.26 Insertion failures in hash tables 131

5.1 CAM-based counter array 138
5.2 Count-Min Sketch 143
5.3 Data structure of the Approximate-CM Sketch (A-CM Sketch) 149
5.4 Hardware architecture of the approximate-CM Sketch 151
5.5 (a) Maximum possible estimation error for HSAC vs. SAC; (b) Mean estimation error for HSAC vs. SAC; (c) Maximum and mean estimation error for HSAC with a varying number of *count* bits (k) 152

5.6 System on chip for functional test . . . 153
5.7 Accuracy variation of A-CM Sketch compared to CM Sketch . . 154
5.8 BRAM utilization and frequency as a function of the sketch width for different implementations . . . 155
5.9 Floorplan of BRAMs (in red) in the naive (top) and optimized (bottom) A-CM Sketch . . . 156
5.10 Data structure of HeavyKeeper . . . 159
5.11 Data structure of Elastic Sketch . . . 160
5.12 Data structure of SPArch . . . 161
5.13 Update scenarios if the flow is not present . . . 164
5.14 Update scenarios if the flow is present . . . 165
5.15 Query scenarios . . . 167
5.16 Accuracy vs number of flows . . . 170
5.17 Accuracy vs memory size . . . 170
5.18 AAE vs number of flows . . . 171
5.19 AAE vs memory size . . . 171
5.20 ARE vs number of flows . . . 172
5.21 ARE vs memory size . . . 172
5.22 Flow detection: precision vs number of flows . . . 174
5.23 Flow detection: precision vs memory size . . . 174
5.24 Flow detection: recall vs number of flows . . . 175
5.25 Flow detection: recall vs memory size . . . 175
5.26 Accuracy vs number of flows (SPArch) . . . 176
5.27 Accuracy vs Fingerprint size (SPArch) . . . 176
5.28 ARE vs Fingerprint size (SPArch) . . . 177
5.29 Flow detection: precision vs fingerprint size (SPArch) . . . 177
5.30 Flow detection: recall vs fingerprint size (SPArch) . . . 177
5.31 Organization of buckets: separate memory blocks . . . 178
5.32 Organization of buckets: single memory block . . . 180
5.33 Hardware architecture of SPArch . . . 183
5.34 Throughput versus memory and area . . . 187

6.1 System architecture of FLARE . . . 198
6.2 Aggregator . . . 200
6.3 TTL logic . . . 201
6.4 Architecture of blacklist . . . 203
6.5 Evaluation testbed . . . 205
6.6 Architecture of the system model of LOFT. Source: [217] . . . 207
6.7 LOFT architecture with a detailed look into Active Flow List . 209
6.8 Update counter array and estimate logic of LOFT . . . 210
6.9 Switching and fixed counter array logic . . . 211
6.10 A timeline of LOFT . . . 212

8.1 Growth in Carpet Bomb attack (Data: [32]) 222
8.2 Growth in Mirai botnet attack (Source: [32]) 222
8.3 DDoS defense solutions 223
8.4 Timeline to market 233

List of Tables

3.1 Notational conventions of Xoodoo-NC 65
3.2 Avalanche Scores for Xoodoo-NC 66
3.3 Comparison of hash implementations on a Xilinx Virtex Ultrascale+ 67
3.4 The number of rounds needed for meeting the avalanche criteria 78
3.5 Maximum operating frequency of hash implementations 81
3.6 Resource utilization 82
3.7 Comparison of maximum frequencies, throughput (Tp), throughput per LUT (Tp/LUT), and delay with related work 82

4.1 Comparison of Bloom-1, PBF, and CAM implementations on a Xilinx Virtex Ultrascale+ 106
4.2 Comparison of the properties of the Bloom filter architectures considered in this study 107
4.3 Comparison with related work for $n = 1024$ 109
4.4 Features of basic AMQ data structures 111
4.5 FPR and time complexities of data structures 121
4.6 FPR and memory access delay vs various load factors for lookup data structures 125
4.7 Performance of Lookup architectures on FPGA 126
4.8 Accuracy and memory access delay vs various load factors for key-value stores 127
4.9 Performance of hash table architectures on FPGA 128
4.10 Results of partial key Cuckoo hashing as partial key-value store 128

5.1 Comparison of counting architectures 141
5.2 Implementation results of the SOC on the Virtex-7 FPGA . . . 153
5.3 Comparison of approximate-CM Sketch with exact (full-precision) counter based CM Sketch 154
5.4 Optimized versus Naive A-CM Sketch implementation for smaller parameter sizes 155

5.5 Throughput computation of the optimized A-CM Sketch 157
5.6 Throughput comparison of FPGA-based sketch implementations 157
5.7 Notational conventions of SPArch 163
5.8 Comparison of hardware resources and latency of SPArch against CM Sketch and Heavykeeper 186

6.1 Resource utilization 214

Chapter 1

Introduction

1.1 The path to a connected world

The growth of technology has been rapid and still continuing at a remarkable pace. The advancement of technology, especially electronics, has been far-reaching and touches almost all aspects of human life, including healthcare, education, communication, entertainment, and many more in one way or another. The invention of bipolar junction transistors in 1947 blazed a trail to the development of modern electronic devices that are smaller, more portable, and more energy-efficient than anything that had come before. The first-ever bipolar junction transistor assembled is displayed in Figure 1.1. Three-quarters of a century later, the world has become truly limitless with the help of devices that can perform very complex tasks effortlessly in no time. These devices range from powerful supercomputers to tiny nanorobots and from sturdy space electronics to flexible chips, offering unprecedented possibilities.

All these powerful devices and technology would have been meaningless if there were no connectivity. Humans have a natural inclination to connect with one another. It is a fundamental aspect of our social essence as a species. Through social connections, we can share ideas, emotions, experiences, and knowledge. This innate drive was the motivation that eventually led to the invention of the Internet. The Internet can be seen as a network of networks, enabling devices located anywhere in the world to communicate and exchange information with each other.

In the late 1960s, the United States Department of Defense's Advanced Research Projects Agency (ARPA) created ARPANET (Advanced Research



Figure 1.1: The first bipolar junction transistor; a small step for technology, a quantum leap for human innovation. (Source: [28])

Projects Agency Network) [205]. The purpose of ARPANET was to establish a reliable communication system that could link together the country's research institutions and government agencies. The development of ARPANET was a significant milestone in the evolution of the Internet. It was the first network to use packet-switching technology, which enabled the transmission of data in small packets over a network of computers.

ARPANET was launched in 1969 with four connected nodes situated at universities in California and Utah. Over time, the network expanded, and by the mid-1970s, hundreds of computers were connected to ARPANET. The protocols developed for ARPANET, such as Transmission Control Protocol/Internet Protocol (TCP/IP), became the foundation of the Internet, which has since grown into a global network connecting billions of devices worldwide. The never-ending quest for a connected world made it possible that every device can be connected, and everyone can be connected to each other through these devices. The world is now connected like never before. A brief timeline of the history of the Internet is shown in Figure 1.2. By 1985, Ethernet and Ethernet standards were introduced. Dial-up Internet came into existence in 1989, and the Internet went commercial in 1990. Since 1990, the boom of the Internet was so rapid. From ARPANET connecting 4 US universities in 1969 to more than 14 billion devices connected to the Internet in 2023, the Internet has become the backbone of modern society, powering communication, commerce, and entertainment. But with this increased connectivity comes an equally increased threat: the danger of malicious actors using the network to gain unauthorized access, steal sensitive information, or wreak havoc on critical systems.

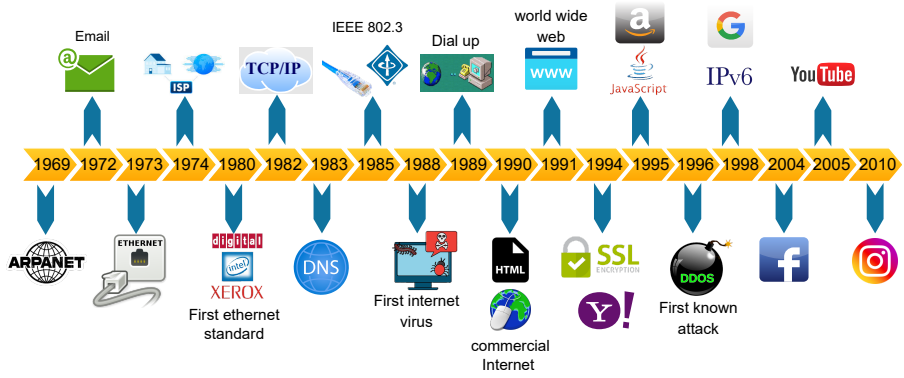


Figure 1.2: Examples of impactful standards, services and applications in the history of the Internet

1.2 The threats in the connected world and the importance of security

Increasing connectivity and its consequences: Since ARPANET, connectivity has grown at a fairly rapid pace. Ethernet came into existence in 1973, and the first Ethernet standard was proposed in 1980. The term 'Internet' came into existence in 1974. Internet speeds and bandwidths started increasing. Since 2010, there has been a rapid advancement in Internet speeds and Ethernet speed standards, as shown in Figure 1.3. The global average Internet speed, which was 14.4 Kbps in 1991, reached approximately ≈100 Mbps in 2022 [10]. The 800Gb and 1.6Tb Ethernet are expected to be introduced by 2025 [7]. The number of Internet users around the world has gone from 2.6 million in 1990 to 5.1 Billion in 2022 [6]. As of 2022, every day, approximately 5.6 billion Google searches are made, 90 million photos are shared on Instagram, 720,000 hours of videos are uploaded to YouTube, 500 million tweets are posted, and a total of 2.5 Quintilian bytes of data are generated [12].

With the increasing data rates and the increasing number of devices connected to the Internet, the number of cyber crimes also increased drastically. In recent years, the world has seen a significant increase in cyber attacks, with individuals, businesses, and governments falling prey to sophisticated cybercriminals. These attacks can range from simple phishing emails to perilous intrusion in nuclear facilities (Stuxnet [155]). With advancing technology, cyberattacks have evolved into increasingly complex and sophisticated forms, posing a significant threat to individuals and organizations worldwide. The encrypted data that is currently

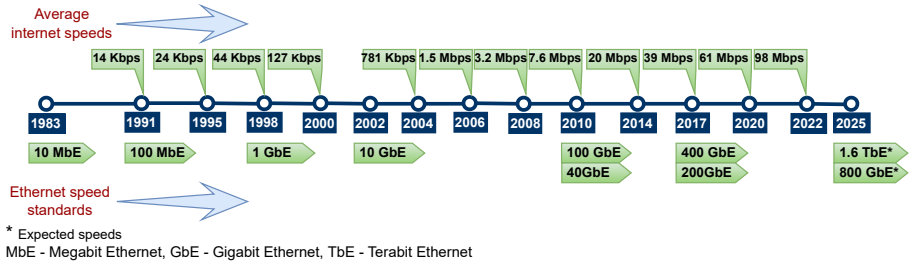


Figure 1.3: Internet speeds and speed standards over the years

impossible for attackers to decrypt can still be valuable to them in the future. Harvest Now, Decrypt Later (HNDL) [11] theory states that encrypted data can be collected and stored now and decrypted at a later time using a quantum computer.

The rise in cyber attacks can be attributed to several factors, including the increasing reliance on technology and the Internet, the ease of access to hacking tools, and the increasing value of digital assets. The COVID-19 pandemic has brought about a significant shift towards remote and online working, as many businesses and organizations had to adapt to lockdowns and social distancing measures. Remote working has become a necessity for many businesses, as it allows employees to work from home or other remote locations, reducing the risk of COVID-19 transmission in the workplace. This shift towards remote working has further accelerated the rise in cybercrime as cybercriminals exploit the vulnerabilities and security gaps in remote work environments to gain access to sensitive information, steal data, or disrupt operations. The data shows that there is a sudden spike in cyber crimes in 2020 shortly after the beginning of the pandemic [179, 14]. Although the pandemic has now subsided, remote work arrangements are still in effect. Figure 1.4 shows the impact of COVID-19 pandemic on cybercrimes. The COVID-19 pandemic serves as a warning that similar unforeseen circumstances could lead to a recurrence of the cybercrime incidents that occurred during the pandemic.

The cost of cyber attacks: Cyber attacks can have a significant impact on the world in various ways. They can disrupt critical infrastructure and services such as power grids, healthcare systems, and financial networks, causing widespread chaos and loss of life. Additionally, cyber attacks can compromise sensitive data, leading to identity theft, financial fraud, and espionage. They can also damage the reputation and credibility of businesses and governments, causing a loss of trust from the public and investors. Moreover, cyber-attacks can be used as a tool for political or ideological agendas, causing tensions and conflicts

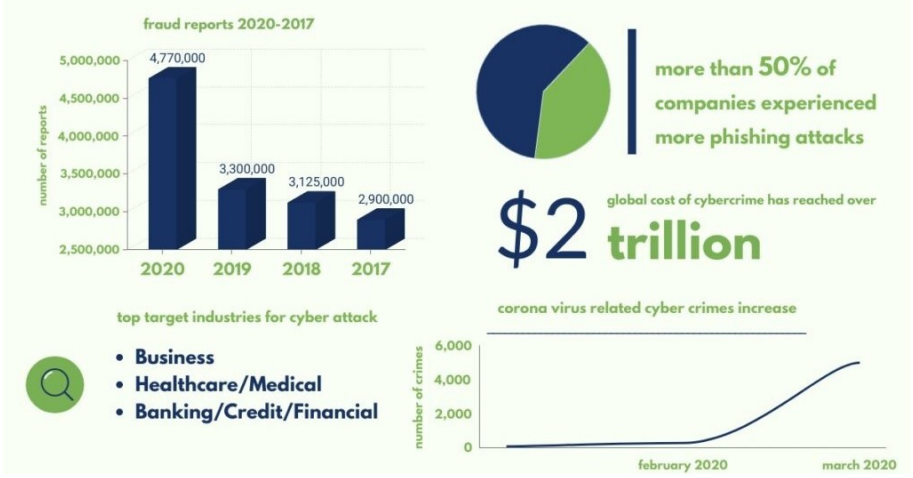


Figure 1.4: Impact of COVID-19 on cybercrime (Source: [14])

between nations. Overall, cyber attacks have the potential to cause significant harm to individuals, organizations, and entire nations, making cybersecurity a critical concern in today's interconnected world.

Most organizations, especially small to medium businesses, fail to create a safe remote-working environment despite technological advancements. Only 14% of small and medium businesses have a cyber security defense mechanism in place, which makes them the favorite targets of scammers compared to well-established organizations with high-end security systems. Data shows that 43% of all attacks were targeted at small and medium businesses [176]. These attacks directly or indirectly affect the customers as well. IBM's cost of data breach report 2022 [3] states that 83% of the organizations met with more than one data breach, and 60% of the organizations pass the money loss to customers by increasing the prices.

Examining the most significant data breaches recorded throughout history, as shown in Figure 1.5, unveils a startling revelation: these breaches have occurred within the largest global repositories of personal information, including renowned entities like Yahoo, Aadhar, Facebook, and Microsoft. Notably, Aadhar retains even the biometric data of all Indian citizens. The consequences of these breaches extend beyond mere operational disturbance and financial ramifications for the affected companies, comprising identity theft and intellectual property losses.

The global annual cost of cyber attacks is going to reach $8 trillion as of 2023, which is equal to $257,000 per second. The average cost of a single data breach

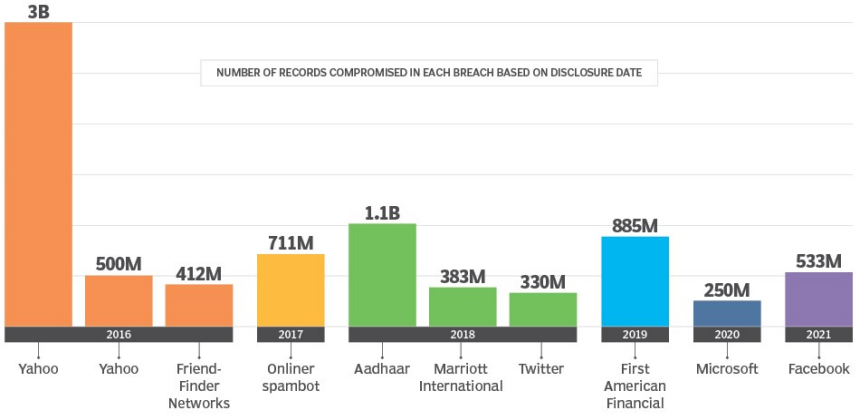


Figure 1.5: Ten biggest data breaches in history (Source: [19])

was $4.35 million in 2022. The healthcare sector suffers the most, as the average cost of a data breach is 10.1 million in 2022. It requires an average of 277 days to identify and contain a data breach [3]. The average cost of a breach becomes $1.07 million higher when the breach happens due to remote work. The transition to remote work has resulted in an increase in the frequency of phishing attacks, and it is worth mentioning that phishing is the primary cause of data breaches. The breaches that start with phishing using malicious URLs were ≈7 million in 2019 and have increased to ≈30 million in 2022 [16]. The cost of data breaches in recent years is also impacted by COVID-19, underscoring the importance of implementing strong security measures to address unprecedented circumstances. The annual cost of data breaches in Belgium in the last four years is depicted in Figure 1.6. The data reveals that the cost of cyber attacks skyrocketed in 2020, largely attributed to the sudden onset of COVID-19. However, organizations with advanced security measures have encountered a mere 2% of the total attacks that happened in Belgium [27], highlighting the critical significance of employing robust security measures.

The importance of network security: The increasing threats in cyberspace mark the importance of network security. Network security refers to the measures taken to protect computer networks from unauthorized access, misuse, modification, or destruction. In today's hyper-connected world, network security is more important than ever, as the value of the information that travels across networks increases with each passing day. From government agencies to large corporations to individual users, network security plays a vital role in protecting against cyber threats and ensuring the confidentiality, integrity, and availability of information. In the world of network security, the stakes are high, and the

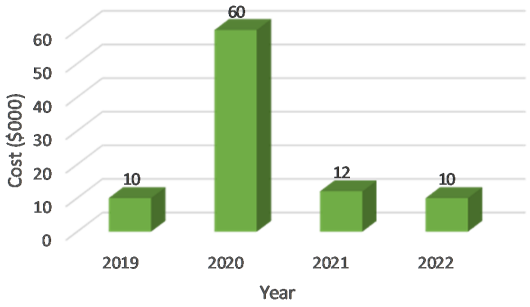


Figure 1.6: Median cost of all cyber attacks in Belgium in the recent years (Data: [27])

consequences of failure can be catastrophic. From the constant barrage of phishing emails and malware attacks to the ever-present danger of data breaches and cyber-espionage, the need for robust and effective network security measures has never been greater. Network security experts face a daunting challenge, one that requires vigilance, expertise, and an unrelenting commitment to keeping the network safe from those who would do it harm.

1.3 The need for hardware acceleration and hardware-based security in networking

The performance of network security algorithms was near-optimal when the line rate was low, and the number of networked devices was limited. The big data era and the proliferation of connected devices give rise to more sophisticated network attacks. The existing algorithms are ineffective in protecting the system against emerging threats and attacks. Moreover, increasing line rates made real-time monitoring using sequential execution model algorithms increasingly difficult. These scenarios give rise to more effective but complex algorithms with better detection capabilities. The rise of artificial intelligence (AI) and machine learning-based detection systems pushed the computational complexities of the algorithms to the next level.

These sophisticated algorithms drastically increased the detection overhead necessitating enormous amounts of computational power to process large amounts of data and perform complex calculations. Traditional processors are not well-suited for these tasks, as they are optimized for sequential processing and lack the parallel processing capabilities needed for high-speed data processing

and AI workloads. Achieving real-time execution with these workloads is unattainable. Nevertheless, monitoring and analyzing network traffic to detect and respond to security breaches and threats in real-time is important to prevent or minimize damage. Considering the enormous amounts of data generated at present times, the task of monitoring and analyzing network traffic poses a significant challenge due to the sheer magnitude of the data involved. In present times, the collective capacity of large-scale Internet routers is in Terabits per second (Tbps), making detection particularly challenging when the line rates are high. This prompts us to set denial of service (DoS) attack detection on high-speed hardware as our primary goal. A DoS attack targets disabling a system by sending an overwhelming volume of malicious network traffic. DoS attacks can cause significant damage compared to other attacks, and real-time detection of these attacks is difficult because of the high data rate. Large flow detection is one approach to detecting these attacks, and our focus lies in efficiently implementing large flow detection algorithms on high-speed hardware to detect such attacks in real-time. In addition to the performance challenges, the existing algorithms are primarily designed for sequential execution on software platforms. Moreover, software implementations are more vulnerable to attacks as they can be easily modified.

These existing challenges prompt the need for hardware acceleration. Hardware acceleration involves employing dedicated hardware components to enhance the performance of specific computational tasks. In today's data centers, the significance of hardware acceleration is growing due to the increasing volume and complexity of real-time data processing requirements. Hardware acceleration platforms such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) can significantly improve the performance of computationally intensive applications because of their extensive parallel processing capabilities. FPGAs can be programmed to perform specific tasks, making them highly flexible and efficient for specific workloads. In recent times, FPGAs have gained widespread adoption in networking and security applications due to their versatility and ability to process data at high speeds. The integration of hardware acceleration into the network security infrastructure became a necessary aspect that could not be avoided. Overall, the need for hardware acceleration in networking applications is driven by faster and more efficient processing of complex workloads. It benefits not only artificial intelligence (AI) and machine learning workloads but also other data center workloads, including database processing, encryption, and video transcoding. Offloading these tasks to specialized hardware components helps to improve performance, reduce power consumption, and lower costs.

Furthermore, FPGA/hardware-based security is less vulnerable to attacks compared to software implementations such as software-based firewalls [239].

Implementing the network security algorithms on FPGAs provides improved security compared to the software-based implementations in general-purpose processors/platforms [139]. An instruction set-based general-purpose processor can be programmed to perform any given task, which leads to an unlimited number of possible states. Software implementation using such a processor increases the likelihood of potential vulnerabilities and flaws. On the other hand, FPGAs can be customized to perform a single specific task, which minimizes the number of states. Also, FPGAs are programmed by changing the hardware circuitry, which makes the physical access of the FPGA necessary for potential attacks. Attackers have access to many open-source hacking tools for targeting software implementations on general-purpose processors, whereas the customized circuitries in FPGAs make it difficult to find suitable open-source hacking tools. Such design features of FPGAs minimize the vulnerabilities and design flaws compared to software implementations [15].

In this work, our goal is to integrate reconfigurable hardware (FPGA) for in-network large flow detection in real-time. When it comes to data processing, FPGAs provide several benefits over general-purpose microprocessors thanks to their ability to perform parallel computing and their possession of a large number of input/output pins. These features make FPGAs well-suited for implementing large flow detection algorithms, which inherently require high-bandwidth data connections and are capable of being parallelized.

1.4 Thesis objectives and outline

The aim of this thesis is to alleviate the challenges in hardware implementation/acceleration of network security algorithms, especially large flow detection, and implement these algorithms efficiently on FPGA. Network security algorithms combine multiple building blocks, such as hashing, memory lookup, and flow measurements, to perform a specific task. Hence, it is necessary to ensure the hardware-efficiency of such components to improve the performance on hardware. Moreover, the limitations of the existing detection algorithms to detect a wide range of attacks calls the need for a universal large flow detection framework. Design and implementation of such a framework on hardware must consider critical aspects such as speed and resource-efficiency.

The primary objectives of this work to achieve the aforementioned goals can be listed as:
1. Design hardware-efficient components and algorithms for large flow detection.
2. Implement the large flow detection algorithms using the designed components.

3. Design and implement a detection framework to integrate distinct detection algorithms.

For the first objective, we pinpoint the bottlenecks in the implementation of large flow detection algorithms on FPGAs. Hash functions, look-up modules, and flow measurement architectures are identified as the components that cause bottlenecks in the FPGA implementation. We have developed efficient designs for hash functions, look-up modules and flow measurement architectures that can significantly boost the performance of detection algorithms on FPGAs. As for the second objective, large flow detection algorithms are implemented on a Xilinx Alveo data center accelerator card, which is equipped to handle data rates of up to 200 Gbps. The implementation makes use of the hardware-efficient components that are designed in the previous objective. To fulfill the third objective, a universal detection framework is also designed and implemented on Alveo data center accelerator card. The individual detection algorithms are integrated using the detection framework on FPGA to detect a diverse range of attack patterns.

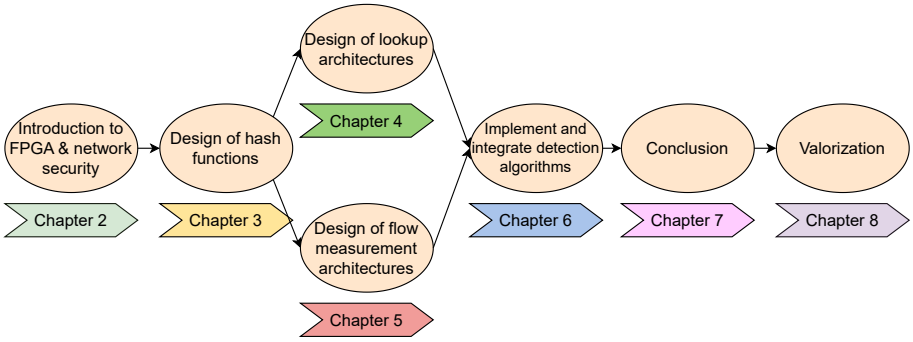


Figure 1.7: Overview of chapters

Overview of chapters

The above-mentioned objectives are addressed in the following chapters of this thesis. A graphical overview of the chapters in this thesis is presented in Figure 1.7. **Chapter 2** introduces FPGAs and provides background information regarding large flow detection and the use of FPGAs in large flow detection. Given that hash functions are the major components in the design of lookup and measurement modules, the design of novel non-cryptographic hash functions using lightweight block ciphers for network security applications is discussed in **Chapter 3**. **Chapter 4** focuses on the design of optimized lookup architectures

for FPGAs, while **Chapter 5** explores the design of various hardware-efficient flow measurement architectures for FPGAs. Based on the components designed in chapters 3, 4, and 5, the thesis progresses to the implementation of a comprehensive large flow detection framework on a high-speed FPGA platform, which is detailed in **Chapter 6**. **Chapter 7** summarizes the contributions made in this thesis and outlines the considerations for future work. Finally, **Chapter 8** explores the possibilities for valorizing the research results.

Chapter 2

Background of network security, large flow detection, and FPGA

This chapter provides a general overview of FPGAs, network security, and large flow detection. For the ease of understanding, this chapter is divided into 3 segments. The first segment (Section 2.1) provides a detailed introduction to FPGAs. In the second segment (Sections 2.2, 2.3, 2.4), an introduction to network security followed by a detailed description of DDoS attacks are provided. Finally, the third segment (Sections 2.5, 2.6) discusses large flow detection and the role of FPGAs in network security. This segment also covers the existing related work in large flow detection and the existing work and research challenges in the implementation of large flow detection algorithms on FPGAs.

2.1 Introduction to FPGAs

Field Programmable Gate Arrays (FPGAs) are versatile and flexible devices that have become increasingly popular in recent years. They are integrated circuits that allow for the creation of custom digital circuits and can be reprogrammed at any time. FPGAs are different from Application-Specific Integrated Circuits (ASICs) in that they are not fixed in their function and can be programmed to perform a wide range of tasks. This makes them ideal for prototyping, testing,

and implementation of digital circuits for various applications. FPGAs have numerous advantages over traditional computing platforms. They offer high flexibility and re-programmability, allowing for quick and easy design iterations. They are also highly configurable, which means that they can be optimized for specific applications. FPGAs are also highly parallel, which allows for the implementation of high-speed and high-throughput applications such as signal processing, data center applications, and cryptography. In addition, FPGAs can be used to implement multiple functions on a single device, which can reduce the overall system cost. GPUs excel at concurrently handling multiple data elements, making them beneficial for tasks such as machine learning, video processing, and graphics processing. Unlike GPUs, which are optimized for performing specific tasks, FPGAs can be programmed to perform any algorithm or function, making them highly versatile. FPGAs are ideal for applications that require custom logic circuits or real-time processing, such as digital signal processing, cryptography, and network processing. Moreover, the diverse range of FPGA platforms ranging from low-end cost-effective chips to powerful hardware accelerator cards, makes them suitable for any application.

2.1.1 FPGA architecture

In this section, we use AMD Xilinx [34] terminology to introduce the typical architecture of an FPGA, as we use AMD Xilinx FPGAs in this research. All the descriptions and architectures are based on AMD Xilinx FPGAs, but these architectural details are also applicable to FPGAs offered by other manufacturers. FPGAs consist of a matrix of configurable logic blocks (CLBs), input/output blocks (IOBs), and interconnects, which allow for the implementation of complex digital circuits. In addition to CLBs, IOBs, and interconnects, FPGAs contain on-chip memory called block RAM (BRAM) and dedicated digital signal processor (DSP) slices. High-end FPGAs such as data center accelerator cards also contain an off-chip high-bandwidth memory (HBM) for high-speed processing. CLBs can be configured to perform various functions, such as logic gates, arithmetic functions, and memory elements. The interconnects are used to connect the CLBs and enable the flow of data between them. Together, these elements make up the basic building blocks of an FPGA. The architecture of an FPGA is shown in Figure 2.1. An FPGA block is divided into multiple clock regions. In each clock region, the configuration layout of an FPGA follows a columnar structure, where CLBs, BRAMs, and DSPs are arranged in dedicated columns. A more detailed explanation of these various blocks is given below.

Configurable Logic Blocks (CLBs): CLBs are the building blocks of an FPGA. Until Virtex 7 series FPGAs of AMD Xilinx, each CLB consisted of two slices and each slice contained a collection of Look-Up Tables (LUTs),

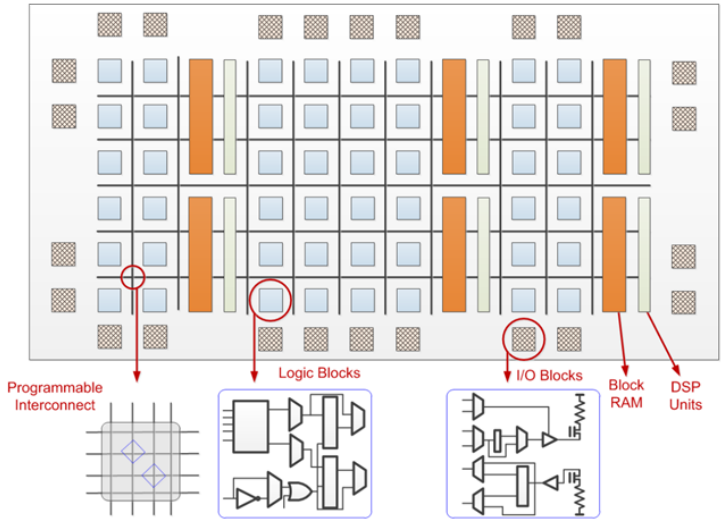


Figure 2.1: FPGA architecture (Source: [125])

flip-flops (FF), high-speed carry logic, and multiplexers. In the most recent Ultrascale series FPGAs, each CLB is enhanced for performance and consists of only a single slice, but the number of logic elements in the CLB is the same as in Virtex 7 series CLBs [17]. Each slice of Virtex 7 series FPGA contains four LUTs and eight FFs. A CLB, consisting of 2 slices, hence contains a total of eight LUTs and sixteen FFs. The Ultrascale FPGA also contains eight LUTs and sixteen FFs in a CLB but as a single slice. A simple representation of the only slice in a CLB of Ultrascale FPGA is shown in Figure 2.2. The output of each LUT can be connected to the slice output or be stored in the flip-flop. The flip-flops can also be driven by direct input to the slice (X or I), or by the carry chain or multiplexer outputs. LUTs are the primary computational elements of an FPGA, and they can implement any arbitrary combinational logic function. Ultrascale FPGAs contain 6-input LUTs, where each LUT can be configured to function either as a 6-input LUT with one output or as a 5-input LUT with two outputs. Flip-flops are used to store data and are the smallest memory elements in the FPGA. Carry logic helps to improve the performance of arithmetic functions by propagating carry through the LUTs. Virtex 7 series CLBs contain two separate carry chains, whereas the Ultrascale CLBs consist of a single carry chain. The carry logic is cascaded to other CLBs. Multiplexers help to connect multiple LUTs together within a single slice to implement larger functions.

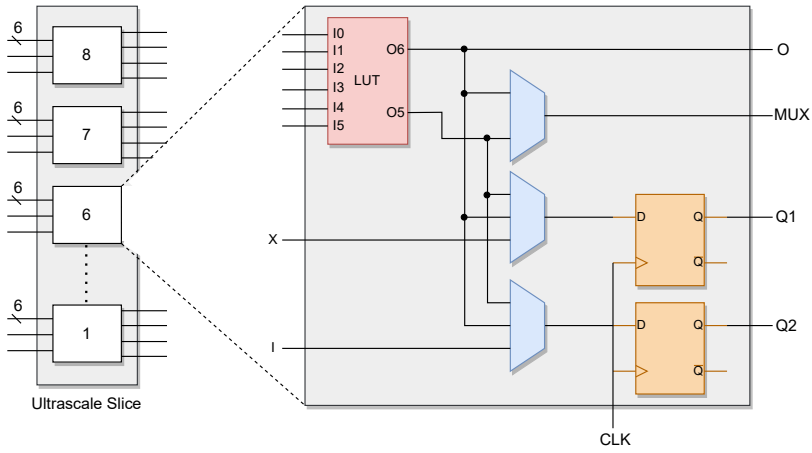


Figure 2.2: A CLB block of Ultrascale FPGA

DSP slice: FPGAs have dedicated DSP slices to enhance arithmetic and signal processing operations. The internal diagram of DSP48E2 present in the Ultrascale FPGA is shown in Figure 2.3. This DSP slice consists of a 27-bit pre-adder, 27x18 multiplier, and a 48-bit accumulator/logic unit. The accumulator/logic unit acts as a post-adder/subtractor, logic unit, and an accumulator.

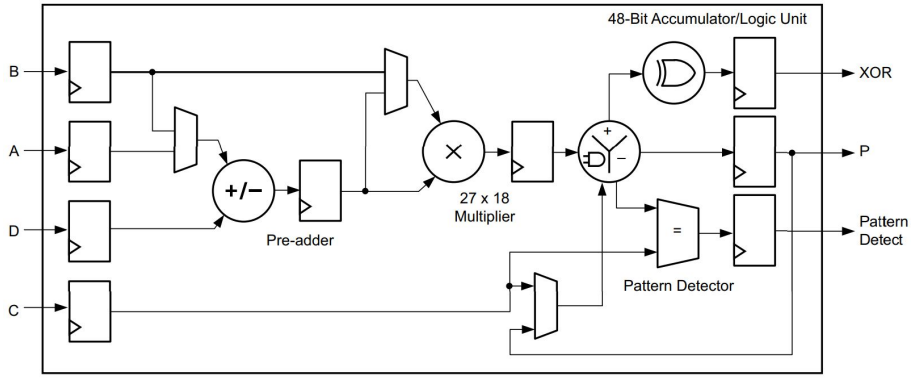


Figure 2.3: DSP48E2 slice (Source: [248])

Block RAM and UltraRAM Block RAM (BRAM) is the limited fast on-chip dual port memory available in the FPGA. In the most recent AMD Xilinx FPGAs, each BRAM is of 36Kb and can be configured either as a single 36Kb

RAM or two independent 18Kb RAMs. In addition to BRAMs, Ultrascale+ FPGAs contain UltraRAMs, which is a high-density dual port synchronous memory block. Each UltraRAM has a size of 4Kx72 bits. The total size (in Mb) of the UltraRAM available in an FPGA is significantly larger than the size of the available Block RAMs.

Intellectual property cores: Intellectual property (IP) cores are standalone and reusable logic units that are intended to perform specific computational/processing tasks. IP cores include arithmetic units, memory interface controllers, DSP units, protocol interfaces, etc. IP cores are optimized for performance and functionality and can be categorized into hard IP cores, firm IP cores, and soft IP cores.

Hard IP cores are a part of the FPGA device and are already hardwired to the device. DSP blocks, BRAM, PCIe, transceiver modules, and Central Processing Units (CPU) in AMD Xilinx FPGAs such as ARM and PowerPC are examples of hard IPs. These IPs result in no additional cost as they are already integrated into the FPGA device. However, hard IPs are not portable as these are FPGA and vendor-specific. Firm IP cores or semi-hard IP cores provide a gate-level netlist that can be used in the FPGA. IP cores like FIFO, adders, memory interfaces, etc., are examples of firm IP cores. By instantiating the netlist (.ngc/.xci files), the core can be included in the project. The resource utilization includes these IP blocks and these IPs can be modified to some level. These cores also have limited portability and can be FPGA or vendor-specific. Soft IP cores are very flexible and are independent of FPGA types or vendors, which makes them portable across different FPGA platforms. Softcore processors in AMD Xilinx FPGAs, such as Microblaze and Picoblaze, and bus interfaces are examples of soft IPs. Soft IPs are implemented using a Hardware Description Language (HDL) and can be modified.

Input/Output Blocks (IOBs): IOBs are used to interface the FPGA with the outside world. They provide buffers and drivers for input and output signals, and they also support different types of signal standards, such as Low-Voltage Differential Signaling (LVDS), Low Voltage Complementary Metal Oxide Semiconductor (LVCMOS), and Stub Series Terminated Logic (SSTL).

Interconnects: Interconnects are the wiring channels that connect the CLBs and IOBs. They are typically arranged in a two-dimensional grid, and they can be programmed to connect different CLBs and/or IOBs together. The interconnects can be configured to support different types of routing, such as direct routing, multiplexing, and global routing.

2.1.2 FPGA connectivity

To enable communication between an FPGA and other devices, there are different types of connectivity interfaces available, such as:

High-Speed Serial Interfaces: These interfaces provide high-speed communication between FPGAs and other devices over a long distance. In a serial interface, individual bits of a data byte are sent sequentially, over a single communication channel or wire. Examples include Serial RapidIO, Ethernet, PCI Express, and USB.

Parallel Interfaces: These interfaces allow FPGAs to communicate with other devices in parallel, meaning that multiple bits of data can be transmitted simultaneously over multiple wires. Examples include Double Data Rate (DDR) memory, LVDS, and Parallel Advanced Technology Attachment (PATA). While Ethernet is a serial interface, modern Ethernet can transmit multiple bits of data simultaneously as the communication channel can have more than one twisted pair of cables or wires, making it a parallel interface.

Low-Speed Interfaces: These interfaces are used for low-speed communication between an FPGA and other devices. Examples include General Purpose Input/Output (GPIO), Serial Peripheral Interface (SPI), Universal Asynchronous Receiver-Transmitter (UART), and Inter-Integrated Circuit (I2C).

AXI interface: In an FPGA design, the Advanced eXtensible Interface (AXI) interface is often used as a standard interface to connect different IP blocks as well as other peripherals within the FPGA fabric. In ultrascale+ devices, AXI interfaces are also used to communicate with high bandwidth memory (HBM). AXI interface is a specification of the AMBA (Advanced Microcontroller Bus Architecture) bus protocol, developed by ARM Holdings for on-chip communication and interfacing in System-on-Chip (SoC).

The communication interfaces, in general, can be categorized into on-chip and off-chip interfaces. On-chip interfaces facilitate communication within a single semiconductor chip, offering low latency and high bandwidth. Off-chip interfaces support communication between separate chips or devices and may have higher latency and varying bandwidth depending on the specific implementation. Except for the AXI interface, all the other interfaces mentioned above are off-chip interfaces.

The choice of interface depends on the specific application requirements, such as the speed of communication, the distance between devices, and the type of data being transmitted. FPGA designers must carefully select and implement the appropriate interface correctly to ensure reliable and efficient communication between the FPGA and other devices.

2.1.3 FPGA design flow

The FPGA design flow is the process of creating a digital circuit that can be implemented on an FPGA device. FPGA design flow is a complex process that requires specialized tools and expertise. The design flow can be automated to some extent using tools like AMD Xilinx Vivado [25], but it still requires significant manual effort and design expertise. The design flow typically involves several stages as shown in Figure 2.4.

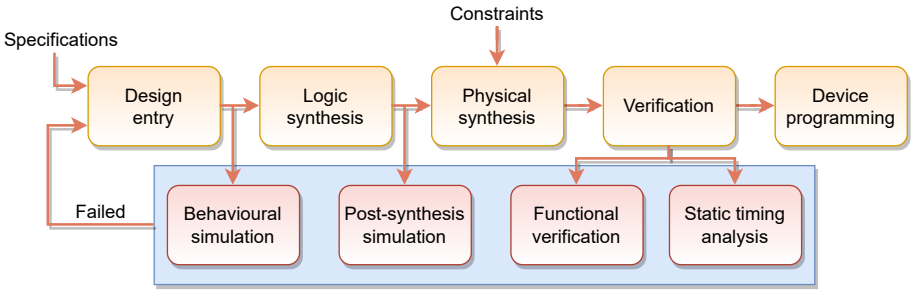


Figure 2.4: FPGA design flow

Design Entry: In this stage, the designer creates a high-level description of the circuit using a hardware description language (HDL) such as Verilog or VHDL. The advent of high level synthesis (HLS) tools have enabled the user to create a design in high-level languages like C++ or Python as well. The design can also be entered using a graphical user interface (GUI) using a tool like Vivado. The design can be simulated to ensure that it behaves as expected. The simulation can be performed using any simulator like the inbuilt simulator in Vivado. If the simulation fails, the design must be revisited for corrections.

Logic synthesis: In logic synthesis, the HDL code is synthesized into a gate-level netlist, which describes the circuit in terms of logic gates and flip-flops. The synthesis tool generates a report that provides information about the design, such as area, timing, and power. Logic synthesis can reveal design issues or errors that were undetected in the behavioural simulation. A post-synthesis simulation can be performed to evaluate the functionality.

Physical synthesis: In the physical synthesis stage, otherwise called the implementation stage, the design is mapped onto the target FPGA device. The implementation is a 3-step process that includes translate, map, and place-and-route steps, which determine the physical location of each gate in the FPGA. The translate step combines the netlist with the help of a constraint file. The constraint file includes specific requirements such as port assignments and

timing requirements. The map process divides the netlist into sub-blocks and maps these sub-blocks into FPGA logic blocks. The place-and-route process places the mapped sub-blocks into the assigned FPGA blocks and wires the blocks. The timing constraints must be taken into consideration during the placing process.

Verification: After the design is implemented on the FPGA, it is necessary to verify that it behaves correctly. This can be done using a variety of techniques, such as functional verification, static timing analysis, and hardware emulation. Static timing analysis ensures that the circuit meets the timing requirements and provides a comprehensive timing summary of the design. Static timing analysis checks for signal propagation delays and computes the critical path that has the longest possible delay. Timing analysis also provides information regarding the negative slack, set-up, and hold time.

Device programming: After the physical synthesis, the design can be converted into an FPGA-understandable file called bitstream using the bitstream generation tool. The bitstream is a binary file that contains the configuration information for the FPGA. The FPGA can be programmed by uploading the bitstream to the FPGA.

2.1.4 FPGA versus other hardware platforms

The discussion about the usefulness of various hardware platforms such as CPU, Application Specific Integrated Circuit (ASIC), FPGA, and GPU is always a tricky topic, especially when it comes to networking applications. The emergence of powerful hardware platforms such as FPGA and GPU has revolutionized the way electronic systems are designed, implemented, and deployed. Even though CPUs do not have the high-performance computing capabilities of FPGAs and GPUs, each technology has its unique features, advantages, and limitations that make it suitable for specific applications. The following set of parameters can be used to compare these platforms. The primary emphasis of the comparison lies in networking and data center applications.

Flexibility: FPGAs offer a high degree of flexibility thanks to their reconfigurable nature, allowing them to be programmed for tailored applications. On a single FPGA, it is feasible to implement entirely different circuits or even incorporate multiple CPUs, allowing them to operate simultaneously [138]. FPGAs can function independently as a computing unit without an associated CPU. GPUs are specifically designed for high-performance computing, focusing on optimized performance. Unlike FPGAs, GPUs are not intended for general-purpose computing tasks and cannot function as standalone units, resulting in a lower level of flexibility than FPGAs. Recent studies have demonstrated

that FPGAs outperform GPUs in accelerating unparallelizable workloads [124]. GPUs also have limited memory bandwidth, which can limit their performance in certain applications. ASICs, on the other hand, are designed only for a specific application and are not so flexible [191, 5]. CPUs offer high flexibility as they are designed for performing general-purpose computing applications.

Performance: The performance can be assessed in terms of throughput, latency, and power consumption. FPGA and GPU are highly parallel and can deliver higher throughput. CPU has a low level of parallelism and moderate power consumption. GPU and CPU consume considerably more significant amounts of power than FPGAs as it is required to access the cache and invoke the instruction set [5]. ASIC can deliver the optimal throughput with the lowest power consumption, but their efficiency is limited to the specific task they are designed for [191, 137]. FPGA and ASIC also have low latencies as they do not have to call an instruction set and do not require a shared memory system, while CPU and GPU have an increased latency due to the instruction set calls and cache access [5]. As the speed of CPUs is considerably lower, FPGA and GPU are the preferred platforms for time-critical processing.

Performance on networking applications: FPGAs and GPUs stand out as the prominent contenders in competition for networking and data center applications. In data centers, FPGAs have the edge over GPUs regarding flexibility, performance, power efficiency, and Non-recurring Expense (NRE) [22, 9]. High-performance accelerator cards from AMD Xilinx, such as Alveo [24], are explicitly designed for data center applications. Compared to GPU-based counterparts, Alveo offers 4× higher throughput and 3× lower latency [24] for compute-intensive applications such as Machine learning acceleration. The performance of FPGAs compared to other hardware platforms in data center accelerations is shown in Figure 2.5, and it is noticeable that FPGAs get the better of GPUs in data acceleration. Even though ASICs are better than other platforms in performance and power efficiency, their performance is the lowest when it comes to workload flexibility [191]. This advantage of workload flexibility over ASIC makes FPGAs far superior to ASICs in data centers where the workloads to be handled are versatile.

Non-reccuring engineering (NRE) costs: The NRE expenses can include the cost of design tools, software licenses, and the cost of developing and testing the design. The cost of FPGAs can vary depending on the applications, potentially resulting in higher unit costs. Nevertheless, the re-programmability and flexibility of FPGAs contribute to mitigating the overall non-recurring engineering expenses. GPUs have lower costs compared to FPGAs, and their re-programmability further aids in reducing NRE costs. However, when it comes to data center applications, FPGAs demonstrate greater efficiency as their long-term maintenance costs are lower compared to GPUs [9]. ASICs incur

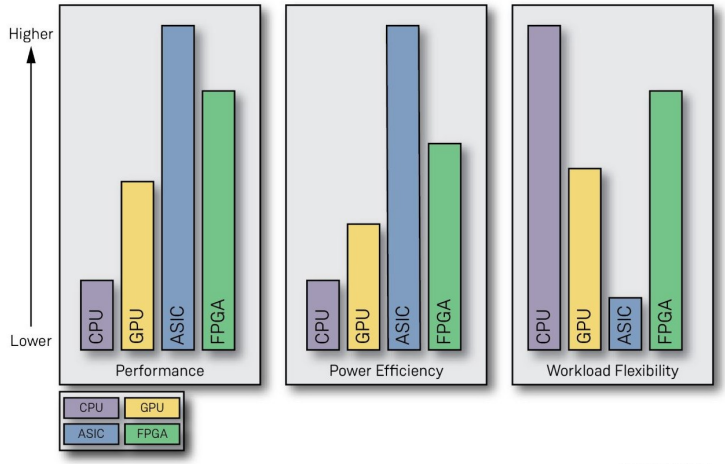


Figure 2.5: Performance comparison of CPU, GPU, ASIC, and FPGA for data acceleration (Source: [22])

substantial NRE expenses due to their reliance on specialized manufacturing processes and the potential need for costly mask sets, often amounting to hundreds of thousands or even millions of dollars, making them less suitable for low-volume production. CPUs' cost and overall NRE expenses are the lowest compared to other platforms.

Development time: FPGAs have a faster development time thanks to a simple design flow, re-programmability, and the availability of application-specific IP cores. The only limiting factor that can slow the development time of custom applications on FPGA is the unavailability of a high-level programming model/framework and libraries [211]. GPUs and CPUs have relatively shorter development times than FPGAs because of the availability of libraries and high-level programming models. ASIC has a complex design flow and manufacturing process that requires significant time and resources, making it less suitable for prototyping and design verification.

Portability: The code written in HDL/high-level languages can be effortlessly transferred to any FPGA, regardless of the manufacturer, given that no manufacturer-specific intellectual property cores are used. Similarly, programs written in high-level languages for CPUs are easily portable to other CPU platforms. However, when it comes to GPUs, there are certain limitations. Different manufacturers have distinct programming platforms, such as the exclusive support of NVidia GPUs by the Compute Unified Device Architecture (CUDA) computing platform [188].

Vulnerability to attacks: As instruction-based general-purpose architectures, CPUs, and GPUs are programmed via software and are very vulnerable to attacks using software-based methods. In contrast, FPGAs are programmed by altering the hardware circuitry, which makes them less vulnerable to software-based attacks. With physical access to the FPGA, side channel attacks are possible, but such attacks are equally likely to happen with CPU/GPU systems, and general-purpose CPU/GPU systems are much easier to attack. Moreover, it is not easy to find open-source hacking tools that target FPGAs, whereas the availability of open-source hacking tools is abundant for CPUs and GPUs. Another possible attack that can happen with FPGAs is reverse engineering of the bitstream. A bitstream is a stream of bits configuring the FPGA logic cells according to the routed netlist of the design. However, the attack would require expert-level knowledge to understand the relationship between the bits in the bitstream to reverse the bitstream to Register Transfer Level (RTL) description [195]. Moreover, the relationship between bits is manufacturer-specific and varies with different FPGAs. Defense mechanisms such as bitstream encryption and other bitstream obfuscation methods [130] can secure bitstreams from attacks, but as demonstrated in previously published research, bitstream encryption can be susceptible to attacks. [102]

Alternative platforms for networking applications - programmable switches: A programmable network switch allows for customized control and management of data traffic in a computer network. The introduction of PISA (Protocol-independent switch architecture) and P4 (Programming protocol-independent Packet Processors) [73] programming language helped to improve the performance of network switches multifold compared to fixed function ASIC switches. PISA is a version of reconfigurable match action table (RMT) [73] that support P4 as the domain-specific language. As shown in Figure 2.6, the fixed function ASIC supports only the already implemented types of packet processing, such as Ethernet logic and VXLAN logic, whereas the PISA switch can be programmed to perform any custom type of packet processing. Intel's Tofino [30] is one such programmable switch, and the latest version, Tofino-3, can support throughput up to 25.6Tbps. Despite the advantage of PISA switches, the current switch architecture offers reduced flexibility. Moreover, hardware constraints such as embedded memory limit the implementation of complex applications [170].

2.1.5 Target FPGA platform in this thesis: Alveo data center accelerator card

In this thesis, we employ Alveo U250 data center accelerator card [246] as the implementation platform. Alveo U250 is a robust FPGA-based solution

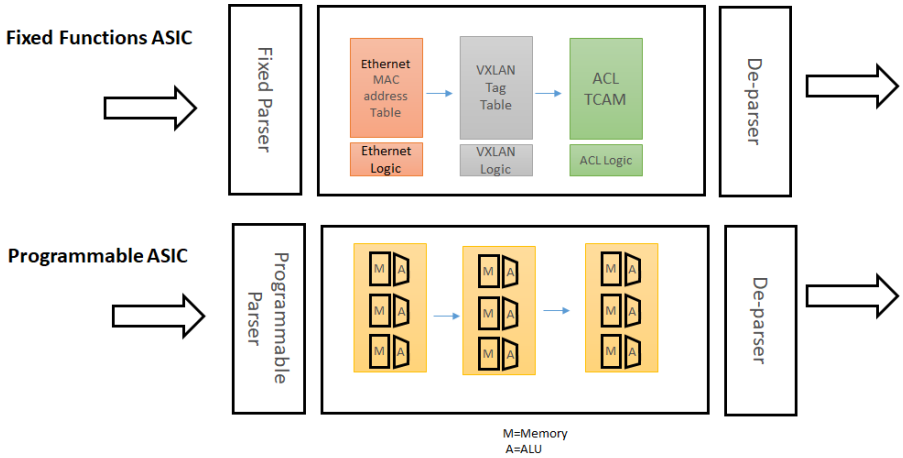


Figure 2.6: Fixed function ASIC switch versus PISA switch (Source: [31])

to accelerate data center workloads, supporting data rates up to 200Gbps. Alveo employs two QSFP28 ports as the network interface. The network interface can transmit/receive 512 bits per clock cycle. It has adequate hardware resources (1,341K LUTs, 2,749K flip flops, and 11508 DSP slices) to support computationally demanding applications. Alveo U250 has 54MB of fast on-chip memory with a total bandwidth of 38TB/s, along with 64GB of off-chip (DDR) memory with a bandwidth of 77GB/s. The on-chip memory includes 2000 Block RAMs of 36 Kb each and 1280 UltraRAMs of 288 Kb each.

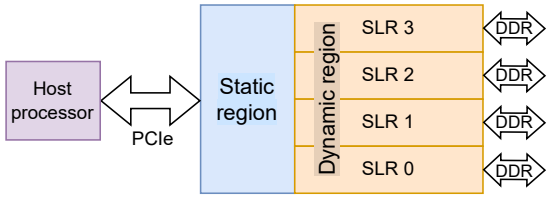


Figure 2.7: Xilinx Alveo U250 floorplan

Figure 2.7 shows the floorplan of the Alveo platform. The platform is divided into two physical regions, namely, the static region and the dynamic region. The static region establishes the fundamental framework for the platform, including essential elements such as PCIe connectivity, board management, sensors, clocking, and reset. Only the dynamic region is accessible to the users, and the custom RTL logic can be programmed into the dynamic region. In Alveo

U250, the dynamic region is subdivided into four super logic regions (SLR) to assist the high-performance designs. For a more comprehensive understanding of the Alveo platform, we refer to the Alveo user guide [247].

2.2 Basics of Computer networks

This section introduces the fundamentals of networking and the networking terminologies used throughout this thesis. A computer network is a collection of interconnected computing devices that communicate with one another to share resources and information. These interconnected devices include computers, servers, routers, switches, and other networking hardware. Computer networks, like Ethernet, permit communication between devices using a variety of protocols and technologies.

2.2.1 Network Protocols

A network protocol comprises a collection of rules and standards that govern the manner in which data is transmitted and received over a network. Some common network protocols are listed briefly here.

Transmission Control Protocol (TCP): TCP establishes a dependable, connection-based communication link between two computers, commonly employed for applications that demand data accuracy and assured delivery, such as web browsing and email. It ensures reliable and orderly data transmission between devices over a network.

Internet Protocol (IP): Internet protocol consists of a set of rules for routing and addressing data packets across a network without establishing continuous connections. It is responsible for addressing and delivering packets between devices on different networks. Internet protocol has two versions, namely IPv4 and IPv6.

Hypertext Transfer Protocol (HTTP): HTTP is used for transmitting and receiving information and other content over the Internet. It is the foundation of the World Wide Web and is used by client-side web browsers to communicate with web servers.

File Transfer Protocol (FTP): FTP is a protocol used to transfer files between computers over a network. It provides a standard way for users to access and transfer files over the Internet.

Simple Mail Transfer Protocol (SMTP): SMTP protocol is used for sending and receiving email messages over a network. It is responsible for routing email messages between mail servers and delivering them to the appropriate mailbox.

Domain Name System (DNS): DNS protocol helps in translating human-readable domain names into IP addresses, which are numerical values that computers can use to identify devices on a network. It is responsible for mapping domain names to IP addresses and providing a distributed directory service for the Internet.

Dynamic Host Configuration Protocol (DHCP): DHCP protocol automatically assigns IP addresses and other network configuration parameters to computers or devices connected to a network. It is commonly used in local area networks (LANs) to simplify network management and configuration.

Simple Network Management Protocol (SNMP): SNMP protocol is used for monitoring and managing network devices connected to a network. It helps network administrators to collect information, monitor network performance, and manage network devices.

2.2.2 OSI and TCP/IP reference models

It is necessary to have a set of protocols to establish the communications between devices in a computer network. The OSI (Open Systems Interconnection) model and the TCP/IP (Transmission Control Protocol/Internet Protocol) model are two conceptual models that define the protocols and communication systems used in computer networks. The major difference between these two models is that the OSI model guides how to set up communication over the Internet, whereas the TCP/IP protocol establishes the standards on which the Internet was built. While the OSI model is a theoretical model useful for understanding the general concepts behind networking, the TCP/IP model is a practical model used to describe the protocols used on the Internet and is widely used in the real world.

The OSI reference model consists of seven layers, whereas the standard TCP/IP model consists of 4 layers. Even though the standardized TCP/IP model has only four layers, the most commonly used model in practice is a 5-layer TCP/IP model. The network access layer in the original TCP/IP model is divided into physical and data link layers in the 5-layer TCP/IP model, the same as in the OSI model. The remaining layers of the original TCP/IP model also remain the same in the 5-layer model. Although the application layer is layer-5 in the 5-layer TCP/IP model, it is often called layer-7, as in the OSI model. Figure 2.8 shows a comparative diagram of the two models.

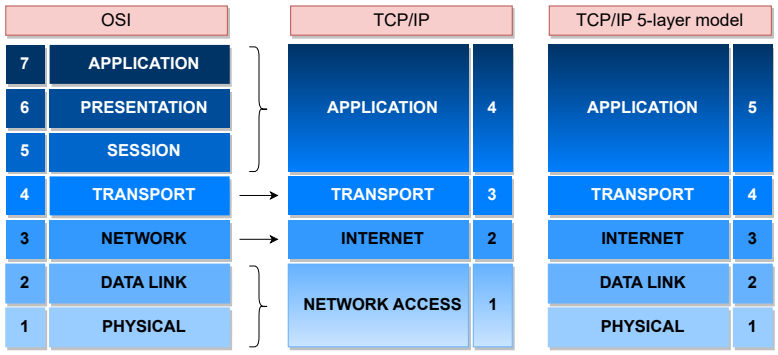


Figure 2.8: OSI reference model versus TCP/IP model

OSI reference model

The OSI reference model is a conceptual framework for understanding how different network protocols and technologies work together to facilitate communication between devices on a network. It consists of seven layers, each representing a different aspect of network communication:

Physical layer: This layer is responsible for the physical transmission of data between devices on a network. It deals with the physical characteristics of the network, such as the electrical and mechanical specifications, and the methods of transmitting data over various types of media (e.g., copper wires, fiber optic cables, radio waves).

Data link layer: The primary responsibility of the data link layer is to provide the functional and procedural way to transfer data frames between two nodes on the same network. It provides a reliable link between two devices by handling issues such as data framing, error detection, and flow control. This layer is divided into two sub-layers, namely the Logical Link Control (LLC) and the Media Access Control (MAC) sub-layers. Ethernet and Wi-Fi are the two most commonly employed protocols in the data link layer.

Network layer: The Network layer translates logical addresses to physical addresses and helps in forwarding and routing packets of data between different networks. It determines the best path for data transmission based on the network topology and routing protocols. The most well-known protocol in this layer is the Internet Protocol (IP), which provides logical addressing and routing services for data packets.

Transport layer: The transport layer provides reliable data transfer services between two end systems. It handles issues such as data segmentation and

reassembly, flow control, and error recovery. The most common protocols in this layer are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP is responsible for reliable and connection-oriented data transfer, whereas UDP provides connectionless data transfer that is unreliable.

Session layer: The session layer is responsible for establishing and terminating communication sessions between applications. This layer allows two applications on different devices to establish a connection, exchange data, and terminate the connection when the communication is complete.

Presentation layer: The presentation layer is responsible for data representation, encryption, and compression. It ensures that data is presented in a format that is meaningful to the application layer. It can also handle encryption and compression of data.

Application layer: The application layer provides network services for end-user applications and enables communication between software applications. It includes protocols such as HTTP (HyperText Transfer Protocol), FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol), and many others to support applications such as web browsing, file transfer, and email.

The OSI reference model is not a specific protocol or technology but a conceptual model that helps standardize network communication across different devices and networks.

TCP/IP model

The TCP/IP model is a communication protocol suite that describes the communication process between two or more computers over a network. The standard TCP/IP model consists of four layers, namely the Network access layer, the Internet layer, the Transport layer, and the Application layer.

Network access layer: This layer deals with the physical transmission of data over the network, similar to the OSI model's physical and data link layers. It includes protocols such as Ethernet and Wi-Fi.

Internet layer: This layer is responsible for the logical addressing and routing of data between different networks, similar to the network layer in the OSI model. It includes protocols such as the Internet Protocol (IP) and Internet Control Message Protocol (ICMP).

Transport layer: The TCP/IP model transport layer is similar to the OSI model transport layer, providing reliable data transfer services between two end systems. It includes protocols such as the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Application layer: The application layer in the TCP/IP model provides application services for end-user applications, similar to the application layer in the OSI model. It includes protocols such as HTTP, FTP, SMTP, and many others.

2.2.3 Ethernet

Ethernet is a network of interconnected devices that is widely used in local area networks (LAN), metropolitan area networks (MAN), and wide area networks (WAN). Xerox developed the Ethernet in 1973. The first commercial Internet standard was introduced in 1980, which was later standardized as IEEE 802.3 [13] in 1983. Over the years, the Ethernet has undergone significant enhancements. As of today, the Ethernet is capable of connecting billions of devices across the network and can support data rates in the order of Terabits, while still maintaining a significant degree of backward compatibility. You can refer to the IEEE 802.3-2022 Standard for Ethernet [29] for more detailed information about the Ethernet standard.

Gigabit and Terabit Ethernet

Gigabit Ethernet (GbE) refers to the Ethernet standard that supports data rates up to one gigabit per second. The Gigabit Ethernet standard came into existence in 1999, and since then, it has been instrumental in powering the modern Internet and has become the prevailing standard for small-scale home networks. Subsequently, faster Gigabit standards such as 10GbE and 40GbE were introduced and have become widely adopted in data centers.

The concept of Terabit Ethernet (TbE) implies the ability to support a data transfer rate of one Terabit per second or 1000 gigabits per second. However, achieving data rates of one Tbps is not yet possible. Currently, Terabit Ethernet is used to describe Ethernet speeds surpassing 100 Gbps until further advancements are made. 100GbE came into existence in 2010, followed by 200GbE and 400GbE in 2012. The introduction of Terabit Ethernet serves as a crucial step towards achieving genuine speeds of 1000 Gigabit/s Ethernet, thereby establishing a solid groundwork for its realization. It is anticipated that 800GbE and 1.6TbE standards will be established by 2025.

Ethernet frame

In Ethernet communication, the transmitting/receiving data between the connected nodes are broken down into strips of data called frames. The relevant information required for routing the frame through the network, such as source and destination addresses, are included in the frame. Carrier Sense Multiple Access/Collision Detection (CSMA/CD) algorithm is used to avoid data collision between the nodes. An Ethernet frame is a data unit that comes under the datalink layer protocol, and the IEEE 802.3 standard specifies the structure of the frame. The Ethernet frame is transported to the network through physical interfaces, and these physical transport mechanisms come under the physical layer. A data packet on the physical layer is called the Ethernet packet, which encapsulates the Ethernet frame. A complete Ethernet packet starts with a packet preamble and a start frame delimiter in addition to the Ethernet frame as payload. Figure 2.9 depicts the structure of the Ethernet packet along with the Ethernet frame.

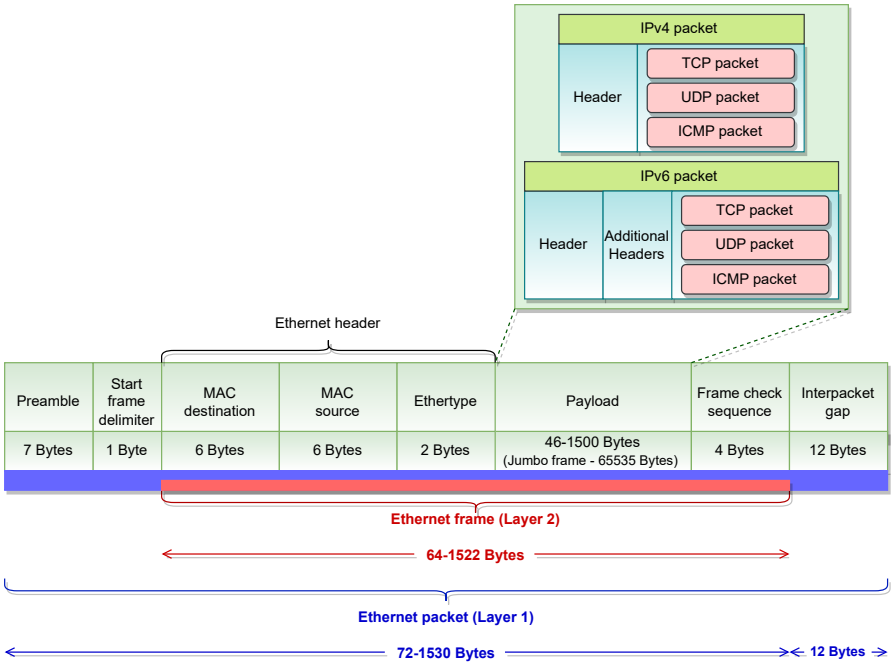


Figure 2.9: Structure of an Ethernet packet

An Ethernet frame consists of an Ethernet header, payload, and the frame check sequence (FCS). The Ethernet header contains the destination and source MAC addresses (6 bytes each) and the EtherType (2 Bytes). EtherType indicates the protocol of the encapsulated payload, such as IPv4 and IPv6. The encapsulated data packet in the payload can be of various types depending on the EtherType [135], including IP packets, and these packets come under layer-3 or the Internet layer. The IP packets consist of a header and a payload. The payload can be of various types depending on the 1-byte IP protocol, including a TCP, UDP, or ICMP. In this thesis, the focus is primarily on the IPv4 protocol.

Flow of data through the layers

The application layer forwards the generated data to the underlying layers. This data is broken down into small units, and these units of data in different layers are generally termed protocol data units (PDU). In the flow of data through the layers, the PDU in each layer is shown in Figure 2.10.

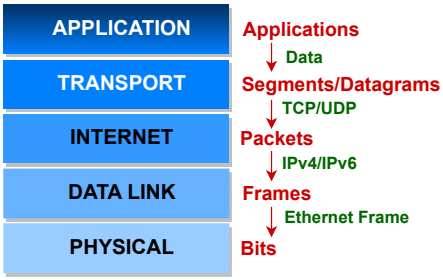


Figure 2.10: Data flow through the layers

The transport layer or layer 4 divides the generated data in the application layer into parts called segments based on the defined maximum segment size (MSS). The MSS value also considers the maximum transmitted unit (MTU), which defines the maximum packet or frame size that can be sent through the network interface. If the packet size exceeds the MTU, layer 3 performs the fragmentation process. A segment is then created and accompanied by a header that includes the source and destination ports, flag bits, sequence number of the segments, checksum, and the options field. The TCP data units are referred to as *segments*, while UDP data units are called *datagrams*. A UDP header comprises source and destination ports, length, and checksum.

In the Internet/network layer (layer 3), the term 'packet' represents the protocol data unit. The transport layer segments are received and combined with a header to create the packet. The packet header includes the source and destination IP

addresses, total length, identification, flags, fragment offset, TTL (time to live), protocol (TCP/UDP/ICMP), header checksum, and options field.

The data unit in the data link layer (layer 2) is called a frame. An Ethernet header is added to the packets received from the Internet layer in the data link layer to construct a frame. The Ethernet header contains the destination and source MAC addresses, EtherType, and a frame check sequence. Finally, the preamble and start frame delimiter are added to the Ethernet frame to create the Ethernet packet in the physical layer (layer 1), which is transmitted through the physical link.

Networking terminologies used in this work

Network flow: A network flow is the term used to represent all network packets streaming through the network that have the same flow identifier (flow ID). The flow ID is extracted from the IP packet header and is often characterized by the 5-tuple (source IP address, source port, destination IP address, destination port, and transport protocol) [59]. A 5-tuple uniquely identifies a TCP/UDP session. For an IPv4 packet, the 5-tuple consists of 104 bits, as shown in Figure 2.11. Flow identifiers can also be defined based on different levels of granularity, such as the source IP address, a combination of source IP address and source port, or any other combination of the 5-tuple. However, the finer the granularity, the more bits are required to represent the flow ID. This increases the storage overhead.

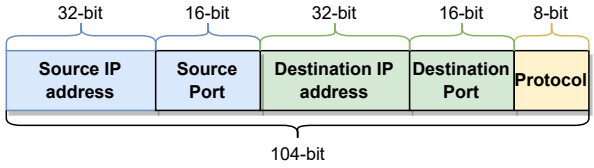


Figure 2.11: 5-tuple of an IPv4 packet

Flow measurement, flow size, and flow byte: Flow measurement is counting the total size of the packets during a measurement period. The total size can be measured by the number of packets in a flow or the byte volume. In this work, we define 'flow size' as the number of packets and 'flow bytes' as the byte volume. Per-flow measurement is one of the cardinal issues in networking in which each active flow is measured [103, 169, 81]. When assessing the performance of per-flow measurement, several metrics should be taken into account. Among those metrics, memory requirement, processing speed,

and measurement accuracy are the most significant [82]. The limited fast on-chip memory in hardware platforms makes the memory requirement a crucial metric for flow measurement. A higher memory requirement would lead to the dependency on off-chip memories, resulting in increased processing time and reduced operating frequency [215]. The processing speed is of great importance when it is required to achieve processing at line rate in high-speed networks. In almost all measurement architectures, the processing speed is dependent on the speed of memory accesses and hash computations [169, 161]. Measurement accuracy plays a crucial role in classifying large and small flows in applications such as large flow detection. An error in measurement can result in classifying a benign flow as malign and vice versa, which could result in blocking a legitimate user and providing access to a malicious user.

2.3 Network security

Network security involves the practices, technologies, and policies that protect computer networks from unauthorized access, misuse, modification, or denial of service. Network security measures include key components such as firewalls, intrusion detection systems, encryption, access control, authentication, and regular security updates to ensure the confidentiality, integrity, and availability of data transmitted over a network. The following sections provide a more detailed insight into network security, the threats, and the defensive mechanisms in network security.

2.3.1 Information security, cyber security, and network security

Security is a broad term with respect to information, the Internet, and networking. In today's data-driven world, approximately 328.77 million terabytes of data are generated every day as of 2023 [97]. Along with such a large data generation rate, cyber-attacks and data breaches are picking up the pace more than ever before. Whether you are an individual or an organization, safeguarding your data is paramount in this digital age. However, the term "security" is a broad and generic one. Information security, cyber security, and network security are the three most frequently used terms to define security when it comes to the Internet. These terms often need clarification as they share significant overlap. Figure 2.12 depicts the information security, cybersecurity, and network security and their overlap.

Information security is a broad field that covers the protection of all types of digital and non-digital information, including data stored on computers,

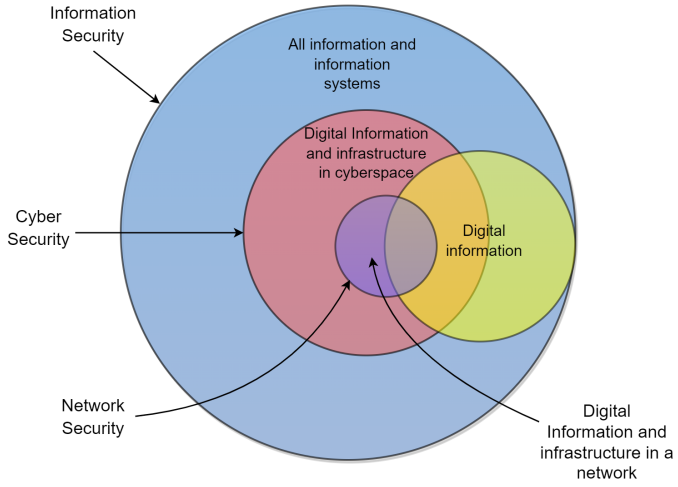


Figure 2.12: Information security, cyber security, and network security

documents, and other forms of media. According to NIST [190], information security (infosec) means "protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide integrity, confidentiality, and availability." Information security ensures the prevention of unauthorized access to all kinds of physical and digital information, not just the data available in networked devices. Information security has to deal with all kinds of attacks that are threats to data in any form. Cybersecurity and network security encounters only digital form of information.

Cyber security is a subset of information security. According to NIST [189], cyber security is "the ability to protect or defend the use of cyberspace from cyber attacks." *Cyberspace* defines the environment where communication happens between the networked devices. Cyber security ensures protection from unauthorized accesses/attacks to all kinds of information as well as the organization's network and systems. Cyber security covers all digital attacks targeting cyberspace, including the cloud, network, and infrastructure. Cyberspace only deals with digital forms of data. The most common cyber attacks include social engineering attacks, ransomware, spyware, password attacks, and denial of service attacks.

Network security is a subset of cyber security. Network security protects the network and infrastructure by preventing unauthorized access, interception, modification, misuse, and breakdown. It protects the usability and integrity of

the network and data. Although both information and cyber security deal with the data available in the network and the device, network security is concerned with the data transmitted in the network alone. Network security deals with all online attacks targeting the network infrastructure. The most common network security risks include denial of Service attacks, zero-day attacks, and trojans.

Developing a security system, whether information security, cyber security, or network security, is based on three components, namely Confidentiality, Integrity, and Availability. These three elements are commonly known as the CIA triad. Confidentiality ensures that only authorized personnel can access or modify the information. Integrity ensures that the information is accurate and unauthorized people cannot modify the information. Availability ensures that authorized people can access data whenever they need it.

2.3.2 Types of network security

There are several types of network security measures that organizations and individuals can use to protect their networks and devices from cyber threats. One way to picture network security is to imagine a fortress with multiple layers of protection. Each layer represents a different aspect of network security that helps safeguard the network from various cyber threats. Figure 2.13 depicts the types of network security in a layered manner.

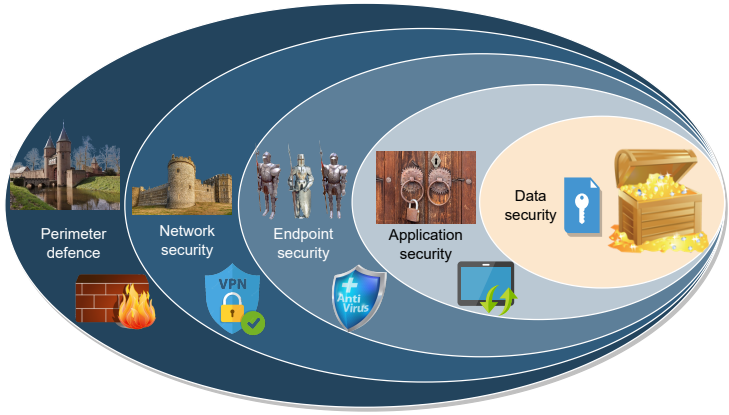


Figure 2.13: Layers of network security

Perimeter defence: The outer layer represents the perimeter defense, which involves securing the physical infrastructure of a network, such as routers, switches, servers, and other network devices. The security measures include

access controls, such as biometric identification or card readers, to prevent unauthorized access to network devices. Firewalls and intrusion detection and prevention systems designed to prevent unauthorized access to the network come under this layer. This layer acts as the first line of defense that monitors and controls the flow of traffic between the network and the outside world.

Network security: The second layer of security is network security, which involves securing the entire network infrastructure itself to protect it from external threats. Network security measures include securing data transmitted over the network, securing network devices such as routers and switches, and implementing protocols such as virtual private networks to ensure secure remote access. Network security also includes measures such as intrusion detection and prevention, traffic analysis, and packet filtering. This layer handles attacks such as DDoS attacks.

Endpoint security: The third layer is endpoint security, which involves securing individual networked devices, such as computers and mobile devices. Endpoint security measures include antivirus and anti-malware, firewalls, and other security software. It also includes measures such as device encryption, vulnerability management, and patch management that ensure that the devices run in a secure environment.

Application security: The fourth layer is called application security, which involves securing individual software applications that run on the network, such as web applications, email clients, and other software programs. Application security measures included secure coding practices, regular software updates, and the use of secure protocols such as HTTPS.

Data security: The fifth and innermost layer of security is data security, which involves protecting sensitive data that is stored, processed, or transmitted over the network. Data security measures included encryption, data masking, access controls, and monitoring. Data security also includes measures to prevent data loss or theft, such as backup and recovery systems, data classification and labeling, and incident response plans. Incident response and recovery include backup systems and data storage, redundancy and failover mechanisms, incident response teams, and disaster recovery plans regularly tested and updated to reflect changing security threats and business needs.

2.3.3 Threats in network security

There are many network security threats that organizations face on a daily basis. Some of the most significant ones are the following:

Malware: Malware is any software that is designed with the intention of causing harm or exploiting computer systems. This malicious software can manifest in various forms, such as viruses, ransomware, and Trojans.

Phishing: Phishing is a form of cyber attack where malicious actors employ counterfeit emails, webpages, or communications to deceive individuals into giving away sensitive data, such as login credentials or financial details.

DDoS attacks: Distributed denial of service (DDoS) attacks involve overwhelming a network or website with traffic from multiple sources, causing it to become unavailable or slow to respond.

Insider threats: Insider threats refer to attacks or security breaches caused by individuals within an organization. This could include employees, contractors, or partners with access to sensitive data.

Advanced persistent threats: Advanced persistent threats (APTs) are sophisticated attacks that use various techniques to gain unauthorized access to a network or system and remain undetected for long periods.

Man-in-the-middle attacks: Man-in-the-middle (MitM) attacks involve intercepting and manipulating communications between two entities with the intent of either stealing sensitive information or modifying the content of messages.

Zero-day exploits: Zero-day exploits are characterized by their exploitation of software or hardware vulnerabilities that remain undisclosed to the vendor or developer. Attackers can exploit these vulnerabilities to gain unauthorized access to systems or networks.

Botnets: A botnet is a network of compromised computers that can be controlled remotely to carry out attacks or engage in malicious activities, such as distributing spam or initiating DDoS attacks.

Social engineering: Social engineering involves tricking people into revealing sensitive information or performing actions that could compromise security. This could include phishing emails or phone calls, or even physical access to a building or system.

Password attacks: Password attacks involve using brute force or other techniques to guess or steal passwords, allowing attackers to gain unauthorized access to systems or networks.

Network security threats can affect the world in various ways, including economic, political, social, national security, and global impacts. Network security threats not only impact organizations, but they can also harm private individuals. These threats can cause significant financial losses, identity theft, emotional distress, and even physical harm to people. Therefore, it is essential to take

appropriate measures to mitigate these threats and protect against potential cyber threats.

Most attacks, such as malware and phishing, can be avoided by following certain guidelines or discipline from the user. Attacks such as zero-day exploits, password attacks, and man-in-the-middle attacks can be controlled by enhancing the encryption criteria. However, attacks such as DDoS attacks are beyond the user's control. This prompts the introduction of additional detection and prevention algorithms for mitigating such attacks. DDoS attacks remain a significant threat to organizations of all sizes, and it is critical to have robust defenses in place to protect against them.

2.4 Distributed denial of service (DDoS) attacks

DoS stands for Denial of Service attack, a type of cyber attack where an attacker tries to make a website or network resource unavailable to its intended users by flooding it with traffic or requests. A DoS attack aims to overwhelm the target system with traffic or requests so that it becomes inaccessible to legitimate users. DDoS stands for Distributed Denial of Service attack, a type of cyber attack similar to a DoS attack but involves multiple sources to flood the target system with malicious network traffic [4]. In a DDoS attack, the attacker typically uses a network of compromised devices called botnets, including computers, servers, IoT devices, and even mobile devices, to launch the attack. These devices are often infected with malware that allows the attacker to control them remotely and use them to send traffic or requests to the target system. A huge amount of unwanted traffic sent from multiple sources depletes the available resources of the victim and causes a particular service or network resource to be unavailable to legitimate users. Figure 2.14 depicts a typical DDoS attack where an attacker make use of compromised devices or botnets to take down a target.

Over time, the frequency and severity of DDoS attacks have seen a drastic increase. The first recorded instance of a DDoS attack dates back to the year 2000 when a 15-year-old student, who went by the online alias 'Mafiaboy,' targeted well-known websites like Yahoo, eBay, Amazon.com, Fifa.com, and Dell [75]. This attack, carried out using compromised university computers, even had repercussions on the stock markets. Since then, DDoS attacks have become a prevalent issue in cyberspace, with various types of attacks emerging, such as multivector attacks like Mirai Botnet [56] and Brickerbot [136]. Emerging attack types such as Carpet Bomb DDoS attacks [35] can inflict a lot more damage than existing attacks as these kinds of attacks can evade the existing detection algorithms. According to Cisco's projections, the total number of

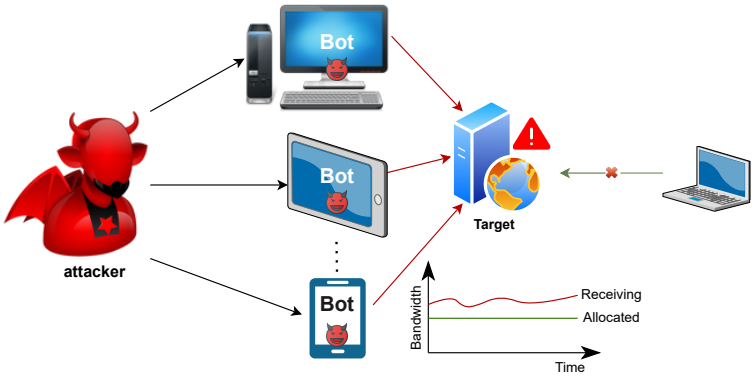


Figure 2.14: DDoS attack

DDoS attacks is estimated to reach approximately 15.4 million by the year 2023 [20]. This forecast highlights the escalating scale and severity of these cyber threats.

The magnitude of DDoS attacks has also been steadily increasing over the years. The DDoS attacks can be measured in terms of three metrics: bits per second (bps), packets per second (pps), and requests per second (rps). Attacks that target network links are measured in *bps*, attacks aimed at network equipment, or DNS servers are measured in *pps*, and the attacks against application servers are measured in *rps* [21, 18]. The largest DDoS attack ever recorded in terms of *bps* is a 3.45 Tbps attack reported by Microsoft in 2021. Prior to that, in 2017, Google encountered a significant 2.54 Tbps attack, which was the largest recorded attack at that time. Another major attack of size 2.3 Tbps was reported by Amazon web services in 2021. In terms of *pps*, the most substantial attack on record was detected by Akamai in 2020, reaching a magnitude of 809 Mpps. Most recently, in 2023, Cloudflare reported a 71 Mrps attack, the largest ever attack in terms of *rps*, surpassing a 46 Mrps attack mitigated by Google in 2022. Such large attacks cause significant financial damage as well. The availability 100GbE connectivity would further increase the attack magnitude. As per the recent reports, the financial impact of a DDoS attack is quite substantial, with organizations facing costs of approximately USD 100,000 for each hour the attack persists [23]

2.4.1 Classes of DDoS attacks

Two main classes of DDoS attacks, according to exploited vulnerability, are bandwidth depletion attacks and resource depletion attacks [92, 222]:

A. Bandwidth Depletion Attacks

These attacks focus on consuming all available bandwidth of the targeted system or network by flooding it with a high volume of traffic. The aim is to make the system unresponsive by overwhelming its capacity to handle incoming traffic. Bandwidth depletion attacks employ Botnets and amplification techniques such as DNS amplification or network flood. Some of the bandwidth depletion attacks are:

Flood attack: In a flood attack, the attacker employs bots to flood the target system with a substantial volume of IP traffic, aiming to overwhelm the target's available bandwidth. Network floods and Domain Name System (DNS) floods are examples of flood attacks. In a network flood attack, the attacker sends a large amount of network traffic to a targeted system with the goal of saturating the network bandwidth. UDP flood and ICMP flood are different types of network floods. In a DNS flood attack, the attacker sends a large number of DNS requests to a DNS server aiming to overwhelm the server and make it unable to respond to legitimate requests.

Amplification attack: The amplification attack involves the exploitation of the broadcast IP address feature present on many routers. The attacker or their agents leverage this feature to amplify the attack by sending messages to a broadcast IP address. This action prompts the routers in the network to transmit the packets to all the IP addresses within the broadcast address range. As a result, the victim's system bandwidth is diminished by the amount of malicious traffic generated.

Amplification attacks can be used to target a variety of services, including DNS, Network Time Protocol (NTP), and Simple Network Management Protocol (SNMP). DNS amplification attacks involve the exploitation of vulnerabilities in the DNS servers to generate large amounts of traffic directed toward the target of the attack. NTP amplification attacks exploit vulnerabilities in the Network Time Protocol, which is used to synchronize the clocks of computers on a network. By spoofing the source IP address of an NTP packet, the attacker can direct the response to the target of the attack, amplifying the amount of traffic directed at the target. SNMP amplification attacks exploit vulnerabilities in the Simple Network Management Protocol, which manages and monitors network devices. By sending a request to a vulnerable SNMP

server, the attacker can elicit a large response directed at the target of the attack, amplifying the amount of traffic directed at the target. E.g., NTP amplification, DNS amplification.

B. Resource Depletion Attacks

These attacks target specific resources of the targeted system, such as CPU, memory, or disk space, to make the system unresponsive. Resource depletion attacks do not consume as much bandwidth as volumetric attacks, but they can still cause significant damage to the targeted system. Resource depletion attacks can be categorized as protocol exploit attacks and malformed packet attacks.

Protocol exploit attack: A protocol exploit attack is a type of cyber attack that targets vulnerabilities or weaknesses in network protocols. Network protocols are the rules and procedures that govern how devices communicate and exchange data over a network, such as the Internet. This attack exploits a specific feature or implementation bug of protocols installed on the victim's device, causing it to use an excessive amount of resources. SYN flood is an example of a protocol exploit attack.

Malformed packet attack: This attack sends incorrectly formatted IP packets to the intended victim to cause their system to crash. The malformed packet attacks can be categorized into IP address attacks and IP packet options attacks. An IP address attack involves identical source and destination IP addresses of the packet, leading to confusion within the victim's operating system and causing system crashes. An IP packet options attack involves a malformed packet that alters the optional fields within an IP packet and sets all quality of service bits to '1.' This causes the victim's system to spend additional processing time analyzing the traffic. When multiple agents are employed, this attack may result in the victim's system crashing. Smurf attack and Ping of Death are examples of malformed packet attacks.

2.4.2 Types of DDoS attacks

There are several types of DDoS attacks [91], including:

Volumetric attacks: This type of attack floods the target with a massive amount of traffic, typically generated using amplification techniques such as amplification attack and UDP flood attack. Volumetric attacks are bandwidth depletion attacks. It is measured in bits per second.

Protocol attacks: This type of attack exploits weaknesses in the protocols used by the target, such as the TCP/IP stack, the HTTP protocol, or the DNS protocol [222]. SYN floods, ping of death, and Smurf DDoS are examples of protocol attacks. These attacks target layer 3 and 4 of the OSI reference model. Protocol attacks are resource depletion attacks. It is measured in packets per second.

Application layer attacks: This type of attack targets the application layer of the target, typically using HTTP floods, slowloris attacks, or other techniques to overload the web server or application [69]. Application layer attacks are bandwidth depletion attacks. HTTP flood attack is an example of an application layer attack. It is measured in requests per second.

Hybrid attacks: These attacks combine multiple techniques, such as volume-based and application-layer attacks, to increase their effectiveness and make them more challenging to defend against. Hybrid attacks can cause either bandwidth depletion or resource depletion, or both.

2.4.3 Defending DDoS attacks: Large flow detection

It is important for organizations to have a comprehensive DDoS defense strategy in place to protect against these types of attacks, as they can have a significant impact on business operations and reputation if left unchecked. In DDoS defense, more attention should be paid to the detection of volumetric attacks and hybrid attacks because these attacks are more difficult to defend against than protocol attacks. This is because protocol attacks commonly occur due to vulnerabilities in the system, whereas volumetric attacks can occur irrespective of the system's robustness. Moreover, volumetric attacks are easier to launch and are difficult to detect. Mitigating volumetric attacks without affecting legitimate traffic poses an even greater challenge when compared to protocol attacks. On the other hand, protocol attacks are less common. Protocol attacks can be easily detected and can be easily mitigated by patching the vulnerabilities. The recent high-frequency recurrence of volumetric DDoS attacks has highlighted the significance of volumetric DDoS defense mechanisms [32]. Hence, our focus is more towards the detection of volumetric attacks.

One approach for DDoS detection is based on the statistical analysis of traffic flow characteristics, such as flow size, duration, rate, and packet count. Detecting the network flows that consume more than the allocated bandwidth or a threshold is a commonly adopted technique to detect DDoS attacks. The network flows that use more than the allocated bandwidth are called large flows, heavy-hitters, or elephant flows. *Elephant flows* and *mouse flows* are terms used to categorize flows based on their total volume during a measurement period. A flow is

classified as an elephant flow/large flow/heavy-hitter if its total volume exceeds the high-bandwidth threshold, while a flow is categorized as a mouse flow if its total volume falls below the low-bandwidth threshold. Even though detecting elephant flows is our topic of interest, it is equally important to detect mouse flows to allocate resources efficiently in hardware. Achieving memory efficiency is one of the primary objectives of this work, which includes allocating counters of varying sizes for mouse and elephant flows due to the low storage requirements of the former. This objective is closely aligned with the detection of large flows. Section 2.5 provides a more detailed description of large flow detection.

2.5 Large flow detection

Large flows, elephant flows, or heavy-hitters are network flows that consume a significant amount of network bandwidth within a specific time period. Large flows can significantly impact network performance, potentially causing network congestion and slowdowns. In some cases, large flows may indicate malicious activity, such as a DDoS attack or an attempt to exfiltrate large amounts of data from a network. Identifying and mitigating large flows can help network administrators to maintain optimal network performance and security.

Large flow detection typically involves analyzing network traffic data to identify flows exceeding a predetermined data volume or duration threshold and thereby consuming significant bandwidth or resources. The threshold may be set based on the organization's or network administrator's specific needs and goals. After identifying a large flow, the monitoring tool may provide information about the source and destination of the flow, as well as other relevant details, such as the protocol and port being used. Based on these details, appropriate actions can be taken to manage or mitigate the large flows, such as applying traffic shaping policies or blacklisting the flows. Large flow detection algorithms typically follow flow-based monitoring, which tracks only the flow statistics of network flows, such as the number of packets or packet size in bytes, instead of capturing the individual packets as a whole. As Section 2.2 mentions, a network flow is identified as all network packets that share the same characteristics, such as source, destination, and protocol. The unique identifier for a network flow is called the flow identifier. More details about the network flow and flow identifiers are discussed in Section 2.2.

Figure 2.15 depicts a typical representation of an online flow-based large flow detection system. A parser extracts the flow identifier from the incoming packet header. The flow identifier is then passed on to the detection algorithm based on a mitigation policy or a filter. The real-time monitoring results or a predefined

set of rules may define the mitigation policy, but typically the policies are based on real-time monitoring results. A commonly used mitigation policy is to use a blacklist. The blacklist stores all the malicious flow identifiers, and if the incoming flow identifier is present in the blacklist, that packet is dropped. The packet is allowed to pass if the incoming flow identifier is not present in the blacklist. The detection module extracts the traffic features with the help of a flow measurement unit. Flow measurement refers to the collection and analysis of network flow data. In large flow detection, the total size of the packets is measured, which can be the number of packets or the byte volume.

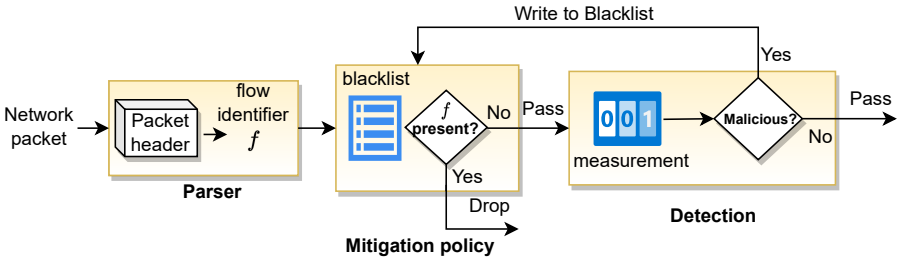


Figure 2.15: Representation of a typical Large flow detection system

The detection algorithm makes a decision based on the measured volume and detection criteria to differentiate whether the incoming flow is malicious. The measured volume of the flows is the average of the total volume over the measurement period. If the incoming flow is benign, it is allowed to pass to the network. The packet is dropped if the flow is malign, and the flow identifier is added to the blacklist. Nevertheless, the blacklisting of malicious flows is one among many mitigation policies. Other mitigation and prevention techniques, such as rate limiting and traffic filtering, are also applicable.

2.5.1 Existing research: algorithmic approaches in large flow detection

There are various approaches to detecting large flows. Based on the packet processing techniques employed in network flow measurement, the detection algorithms can be categorized as sampling-based algorithms and streaming algorithms [221]. Figure 2.16 illustrates the different methods used to identify large flows, highlighting the diverse approaches available.

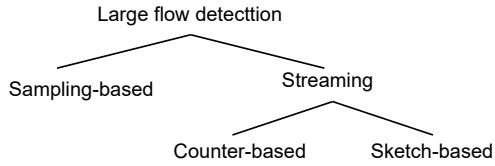


Figure 2.16: Algorthmic approaches in large flow detection

Sampling-based algorithms

The fundamental concept behind sampling-based detection is to select a random subset of packets from the incoming network traffic and analyze them to identify the heavy-hitters. The chosen subset should be large enough to capture a representative sample of the traffic, yet small enough to allow real-time analysis. Sampling-based approaches like NetFlow [85] and SFlow [219] are commonly employed in present-day routers. In these approaches, one out of every N incoming packets are captured. The sampling rate, $\frac{1}{N}$, determines how frequently packets are captured. NetFlow employs very low sampling rates, sometimes as low as 1% or 0.01% (depending on the traffic volume), to reduce processing and storage overhead [221]. sFlow employs even lower sampling rates. The sampling can be of two types, namely packet sampling and flow sampling. Packet sampling creates an exact copy of all the sampled packets, whereas flow sampling only stores the summary of all the flows in the sampled packets. Flow sampling is less resource-intensive because multiple packets can be aggregated to a single flow record. For large flow detection, flow sampling is the most suitable sampling technique.

Streaming algorithms

Streaming algorithms process every packet in an incoming data stream and operate within a limited memory requirement. There is always an accuracy-memory trade-off for streaming algorithms. However, depending on the specific needs of an application, this trade-off can be effectively utilized. Streaming algorithms can be categorized as counter-based and sketch-based based on the counting architecture employed.

Counter-based algorithms process every packet in a data stream and map each flow to respective counters. These algorithms employ associative array-based or hash table-based approaches and are widely used for network measurements. Counter-based algorithms prioritize storing large flows while disregarding smaller ones in order to minimize storage requirements. Exact-counting approaches on

hardware that are based on associative arrays allocate a single counter for each flow, necessitating expensive storage architectures such as content addressable memories (CAM) [53] and additional computational overhead. Counter-based algorithms require storing the complete flow IDs in contrast to probabilistic data structures like sketches [88], which do not keep track of the flow IDs. The approach of storing complete flow IDs is unproductive unless sufficient on-chip memory is available to accommodate all the counters, which is impractical and will not scale with the growing bandwidth of networks, especially when it is required to process flows at line rate.

Sketch-based approaches process every packet in the incoming data stream, unlike sampling-based approaches. Sketches are probabilistic data structures that help to summarize large data streams within limited (on-chip) memory [79, 88]. Sketches make use of the sub-linear space and provide a compact synopsis of the data with some loss in accuracy. Moreover, sketching algorithms can deliver better estimates than sampling/approximation-based algorithms. Nevertheless, with increasing line rates and data explosion, sketches face many challenges in keeping up with the design requirements of high-speed flow measurements. For a probabilistic architecture, accuracy remains a challenge to be addressed. The accuracy increases with higher memory allocation, but achieving higher accuracy while keeping the memory overhead to a minimum is challenging. Sketches do not keep track of the flow identifiers, making it challenging to filter the flows based on flow sizes or delete flows at a later time. Nevertheless, the inherent hardware friendliness of sketches is promising to employ sketches for flow measurement architectures on hardware. Sketches are more suitable for per-flow measurement than counter-based approaches, whereas counter-based approaches are more appropriate for top-k frequent item counting.

2.5.2 Research challenges in large flow detection

Large flow detection has been a research problem over the past two decades. The continuing significance of this topic can be attributed to the worldwide impacts of increasing cyber-attacks and the growing significance of network performance for applications that require low latency. In the modern networking environment, the ever-increasing data rates aggravate the difficulties in accurate network flow measurement and fast detection of malicious flows. Moreover, fast detection of misbehaving flows is of prime importance for detection algorithms to minimize the damages. When prioritizing hardware efficiency, the primary obstacles involve achieving low memory usage, high accuracy, and fast detection.

As discussed in the previous section, the feasibility of employing counter-based streaming algorithms for flow monitoring at line rate, even for top-k detection,

is very difficult because of the high memory footprint of these algorithms. There are several approaches proposed recently to improve the memory efficiency and performance of counter arrays for flow measurement. Exact-counting with shared or virtual counters and the use of hybrid SRAM-DRAM architectures somewhat reduce the memory overhead, but the extent of improvement is not substantial [81, 169]. With an ever-increasing data rate and insufficient fast on-chip memory available in hardware platforms such as FPGAs (even the high-end FPGA like Alveo U250 has only 54MB of internal memory), the feasibility of using exact counters for flow measurement is quite limited.

That paves the way to approximate/probabilistic flow measurement architectures. Even though probabilistic measurement techniques aid in reducing memory usage, architectures such as sketches cannot be used for eviction-based detection algorithms such as EARDet [241], Hashpipe [221], Space-saving [178], and Lossy Counting algorithm [173]. Eviction-based algorithms retain only the flows that meet the threshold criteria, while the outdated flows that exist in the array are evicted. Associative array counters are the best-suited architectures for eviction-based algorithms, as sketches cannot remove existing contents. Even though eviction-based algorithms reduce the memory requirement by storing only the large flows and evicting small flows, there are certain limitations when it comes to hardware implementations. The Space-saving algorithm is ill-fitted for hardware implementations as operations like sorting a list and finding the minimum in a list are not straightforward on hardware, negatively impacting performance when processing at line rate. Since eviction-based algorithms do not admit all the incoming flows in the summary, it would cause significant detection errors under limited memory constraints. Even though the Lossy Counting algorithm is more accurate than the Space-saving algorithm, it is considerably slower [117]. Hashpipe guarantees accuracy but delivers a very low throughput.

The optimal duration of the measurement epoch also plays a vital role in determining the memory requirement and accuracy, where a measurement epoch represents the time frame during which the data is collected or measured. If the measurement epoch is too long, it may lead to a high number of false negatives, where the detection algorithm fails to identify malicious flows. Conversely, if the measurement epoch is shorter, the memory requirement decreases, but this results in more false positives, where legitimate flows are incorrectly identified as malicious. Furthermore, different types of flow detection, such as burst-flood attack detection [216], require specific measurement epoch duration. During a Burst-flood attack, the attacker sends multiple simultaneous bursts through distinct flows that last only for a brief period of time, typically a few hundred milliseconds. These bursts are slightly larger than benign flow bursts, making them difficult to detect. Conventional sketch algorithms struggle to effectively

identify such attacks, leading to a substantial number of false positives. This inefficiency can be attributed to the duration of the measurement epoch and periodic reset prerequisites of the sketch-based algorithms, which clash with the unpredictable timing and duration of the bursts.

In fact, different kinds of large flows require different types of algorithms. Taking measurement epochs into account, we conclude that an effective detection mechanism would involve a combination of these algorithms. Moreover, considering the algorithms in terms of measurement methods (eviction-based or sketch-based), it is essential to have a flexible probabilistic measurement architecture that can be standardized for both sketch-based and eviction-based algorithms. We set this requirement as one of our prime objectives, and such an architecture is discussed in Chapter 5.

Most of the existing research is primarily concerned with the fast detection of flows that are 100 to 1000 times larger than the allocated bandwidth. Failing to promptly detect such flows can have detrimental consequences. In scenarios where the conventional large flow detection algorithms use considerably higher detection thresholds, attackers can exploit this to remain within the defined threshold limits while still overwhelming the link capacity, causing congestion. This emphasizes the critical importance of identifying flows having a flow rate slightly higher than the average flow rate, as their swift detection is of great importance for the fast detection and mitigation of malicious flows.

2.6 Existing research and challenges in implementing large flow detection algorithms in FPGAs

As discussed in Section 2.1.4, FPGAs possess some inherent advantages compared to other hardware platforms, especially in networking and security applications, positioning them to play a major role in data centers and other networking applications in recent times. Beyond their role as an accelerator for high-performance computing applications, FPGAs are used for the implementation of secure algorithms for applications such as network security and cryptography. The applications of FPGAs in data centers and networking environment showcases their exceptional potential in high-speed networking applications [84, 196, 235, 236]. However, the research focusing on large flow detection on FPGAs remains significantly limited. Existing studies predominantly use FPGAs to accelerate applications such as deep packet inspection, frequent item counting, and top-k frequent item detection. Deep packet inspection primarily finds application in intrusion detection systems,

while frequent item detection is commonly employed in large flow detection systems.

Deep packet inspection employs pattern-matching techniques to filter the anomalies in the incoming packets and plays a crucial role in detecting application-level attacks. Even though pattern-matching approaches lean towards intrusion detection systems, they also serve as a fundamental component in large flow detection systems, as elaborated in Section 2.5. The pattern-matching approach is based on a predefined set of rules, and the incoming packet is searched for a possible match with any of the signatures in the rule set. This approach is quite time-consuming when dealing with a substantial number of rules. Three main approaches used for pattern matching on FPGAs are deterministic solutions, non-deterministic finite automata (NFA), and probabilistic architectures [160]. Deterministic solutions and NFA are beneficial in rule-based pattern matching, but the usability of these approaches in large flow detection is minimal. In large flow detection, monitoring, storing, and analyzing the features of all the incoming flows is crucial. Consequently, the overall memory requirement and lookup complexity would be much higher in large flow detection than in rule-based approaches. Compared to other pattern-matching approaches, probabilistic architectures are very promising for implementing larger lookup modules and are much more suitable for large flow detection applications.

Probabilistic lookup data structures such as Bloom filters [70] and Cuckoo filters [108] are far superior in pattern-matching applications compared to content addressable memories (CAM) and ternary content addressable memories (TCAM) [53]. CAM and TCAM are the most common lookup architectures used for pattern-matching algorithms in hardware. In conventional memory, such as Random-Access Memory (RAM), the address is the input, and the RAM returns the data stored at that address. In contrast, CAM searches for a match between input and stored data and returns the address of the matching data. CAMs are commonly employed in network switches to determine the appropriate port for incoming network traffic based on the destination MAC address. A CAM stores each bit explicitly in a memory cell, similar to RAM. Unlike RAM, CAM contains additional hardware circuits that facilitate the lookup, leading to additional resource usage and access delays. The probabilistic data structure, on the other hand, provides the presence of the data but cannot retrieve the data at a later time as it does not store the data itself. However, probabilistic data structures offer constant lookup time, while the lookup time for CAM grows proportionally with size, despite CAM maintaining a fixed latency in terms of the number of clock cycles.

While CAMs and TCAMs offer a hundred percent accuracy, their drawbacks, including high cost per bit, inefficient storage, limited scalability to long input

keys, and significant memory overhead, put them on the back foot [49]. In Bloom and Cuckoo filters, the length of the string has no effect on the memory requirements, and the query time remains unaffected by the number of strings in the database. These advantages provide a substantial boost in performance when it comes to hardware implementations. A more comprehensive analysis of the existing research on probabilistic pattern-matching architectures on FPGA is given in Chapter 4.

Since network flow measurement is the foundation of most of the large flow detection algorithms, many works have been proposed to improve the performance of flow measurement and large flow detection on FPGAs. The storage requirements of per-flow measurement are much higher and it makes per-flow measurement on FPGAs difficult because of the limited memory available. Cache-based hybrid SRAM-DRAM architectures, proposed by Zadnik et al. [255], are suitable for passive flow measurements, where querying is done offline, and real-time flow measurement at line rate is not possible. The implementation by Teubner et al. [226] uses a CAM and exact counters. It is specifically suited for detecting the top-k frequent items and is too primitive for high-speed networks. In general, exact counting approaches are more suitable for top-k frequent item detection where only the most frequent k flows are stored. However, this approach reduces the accuracy of large flow detection.

Probabilistic flow measurement approaches proved to be rewarding on hardware regarding speed, memory, and computational overhead. An in-depth study of the role of sketches for traffic measurement is presented by Zhou in [260]. The effectiveness of sketch algorithms [253] and their utilization in traffic monitoring schemes [238] have already demonstrated the ability to achieve high throughput on FPGA. Lai et al. [157] propose a sketch-based DDoS detection method on the NetFPGA SUME platform, where complex and time-consuming Shannon entropy computations are pre-computed and stored in lookup tables on the FPGA, significantly reducing the latency and achieving a throughput of 40Gbps. Tong and Prasanna [227] proposed an algorithm for online heavy-hitter detection using the CM Sketch. This architecture achieves an impressive throughput of 118Gbps on a Virtex-7 platform by leveraging pipelining and parallel computing engines to accelerate the sketch. Heavy-hitter detection on FPGA using a CM-CU sketch is proposed by Saavedra [208]. The CM-CU algorithm offers the advantage of low memory requirement and requires only a 2-stage pipeline for the update operation. This approach is more advantageous than the one presented in [229] because the deep pipeline in the latter requires complex hazard detection and a data forwarding unit. Some of the other related research targeting hardware focuses on accelerating sketches to improve throughput [230, 158, 237, 209]. The throughput of the work by Lai [158] and Wellem [237] is limited to 10 Gbps on the NetFPGA platform. Saavedra et

al. [209] propose using on-chip Block RAM (BRAM) and efficient pipelined hash computations on an FPGA. However, the external DDR3 RAM used in their approach does not meet the throughput requirements of high-speed networks.

There are a few publications targeted specifically for Terabit Ethernet on FPGA. Zazo et al. [256] present a heavy-hitter detection architecture on 100GbE links on a single FPGA. The detection architecture employs Count Sketch (CS) and leverages the 100G Ethernet subsystem available on AMD Xilinx Ultrascale devices. Recent works such as LOFT [217] for identifying overused flows, ALBUS [216] for detecting burst flow attacks, and Jaqen [165] for detecting volumetric DDoS attacks also emphasize the capability of hardware-based sketches to process data at line rate for Terabit Ethernet.

Challenges

Implementing large flow detection algorithms on reconfigurable hardware poses several challenges, particularly concerning the real-time monitoring of high-speed networks. The challenges associated with the hardware implementation of such algorithms can be one or more of the following:

Memory overhead: In high-speed networks, storing flow characteristics requires a substantial amount of memory. To keep up with the line rate, the computations and the memory accesses involved in these computations must be instantaneous. For instance, AMD Xilinx UltraScale+ integrated 100G Ethernet subsystem sends/receives 512 bits per clock cycle. Assuming a minimum packet of 64 bytes, we need to process a packet in every clock cycle. Hence, it is important to keep the memory access delays to a minimum by employing the on-chip memory for storage. Offloading the storage to off-chip memory is not suitable for real-time monitoring, and the available on-chip memory in FPGAs is limited. Moreover, the conventional associative array counter architectures are not ideal for minimizing the memory overhead.

Even with sketch-based flow measurements, it remains challenging to limit memory requirements within the on-chip memory of FPGAs, as the accuracy is proportional to memory. Tong et al. [230] propose CM Sketch and K-ary Sketch-based implementations for heavy-hitter detection, achieving a throughput over 150 Gbps. Nevertheless, the high memory utilization may prevent the design from increasing the width of the sketch over a certain limit. From another related work by Kiefer et al. [146], it can be understood that exploiting a high level of parallelism to achieve a high throughput also increases the memory as well as resource overhead. Instead of per-flow measurements where each and every flow is stored, top-k large flow detection is more advantageous, but it considerably increases the false positives and false negatives compared to per-

flow measurements. In this thesis, we address the challenges of constraining the memory requirements within on-chip memory while maintaining high accuracy for flow measurements. Chapter 5 provides a detailed discussion of our hardware-optimized flow measurement architectures.

Computational overhead: High throughput is the prime requirement of high-speed networks, and the high parallelism in FPGAs helps to match the throughput requirements. However, the computational overhead of existing large flow detection algorithms has a negative impact on throughput when implementing them in hardware [231, 162, 132, 221, 253]. Here, the sketch acceleration can result in high computational overhead but may still deliver a low throughput. Emphasizing minimal memory usage may also increase the computational complexity. Some existing approaches, such as Countertree [81], Counterbraids [169], and Brick [133], effectively minimize the memory requirements, but at the cost of increased latency because of their computational complexity. One of the highly-accurate algorithms, LOFT [217], also requires a very small memory footprint but has much higher computational and implementation complexity on hardware. Moreover, most probabilistic measurement algorithms are less hardware-efficient because of the presence of complex computational modules such as logarithmic computation [162, 99]. It is very evident from the related work that there is always a trade-off between memory footprint, throughput, and computational overhead in flow monitoring algorithms. All these factors must be taken into consideration while designing hardware-efficient algorithms for large flow detection.

Execution model: There exists only a handful of works on large flow detection that target FPGAs. This also points to the fact that nearly all of the existing algorithms target processors/platforms that support sequential execution, which makes it difficult for them to exploit the parallelism of FPGAs. Translating such software-centric algorithms to hardware is a cumbersome task and results in additional computational and memory overhead that adversely affects the execution speed as well. The algorithms that target FPGAs must be designed in such a way that the memory requirements must be constrained to the limited on-chip memory. Enhancing throughput requires exploiting design parallelism and easier implementation of pipelining operations. To achieve this, a hardware-software co-design approach is the need of the hour, and it requires hardware-efficient components for the design of detection algorithms.

In summary, a hardware-oriented design for large flow detection should have the following needs:

- Need 1: limited to on-chip memory
- Need 2: low latency
- Need 3: less computationally intensive

- Need 4: support for parallel execution

To fulfill *Need 1*, we use probabilistic data structures and approximate computing techniques to restrict the memory requirements. We apply Loop-unrolling techniques to satisfy the requirements of *Need 2*, which reduces the latency of critical operations such as lookup and flow measurements. To accomplish *Need 3*, we employ non-cryptographic hash functions specifically designed to achieve low logical depth, thereby reducing the computational complexity in time-critical components. Leveraging FPGA parallelism in all our implementations, we ensure fulfillment of *Need 4*, enhancing operating speed and throughput. These techniques that are employed to fulfill these requirements for designing hardware-efficient architectures for large flow detection are discussed in detail in the subsequent chapters.

Chapter 3

Hardware-efficient non-cryptographic hashing for networking

This chapter discusses the hardware-oriented design of hash functions, specifically targeting networking and security applications. This chapter is organized as follows. Section 3.1 provides a general introduction to hash functions. Section 3.2 presents the background and relevant work related to non-cryptographic hash functions. The design of the non-cryptographic hash Xoodoo-NC is presented in Section 3.3, followed by the design of non-cryptographic hash functions based on lightweight ciphers in Section 3.4. Finally, Section 3.5 concludes this chapter.

CONTENT SOURCE

- Sateesan, A., Biesmans, J., Claesen, T., Vliegen, J. and Mentens, N., "Optimized algorithms and architectures for fast non-cryptographic hash functions in hardware". Microprocessors and Microsystems, 2023, 98, p.104782. (Contributions: Main author)
- Sateesan, A., Vliegen, J., Daemen, J., and Mentens, N., Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. Microprocessors and Microsystems, 2022, 93, p.104619. (Contributions: Main author)

3.1 Introduction to hash functions

According to NIST [90], a hash function maps a bit string of arbitrary length to a fixed-length bit string. Irrespective of the length of the message, the output of the hash function will always be fixed. Even an infinitesimal change in the input should result in a completely different output, and the resulting output should be unpredictable. Examples of some of the well-known hash functions in use are SHA [45], BLAKE2 [58], and MD5 [204]. Figure 3.1 shows the inputs and outputs of an MD5 hash function, demonstrating how even a slight change in the input leads to a significant change in the output. Unlike encryption algorithms, hash functions are one-way functions, meaning it is not possible to recover the original message from the digest. Modern cryptography relies on hash functions to ensure information security by verifying transactions, messages, and digital signatures. Hash functions are vital components in probabilistic data structures, which are commonly employed in networking and security applications. The subsequent sections will delve into more comprehensive discussions about hash functions, covering their properties, applications, types, and how they are applied in networking applications.

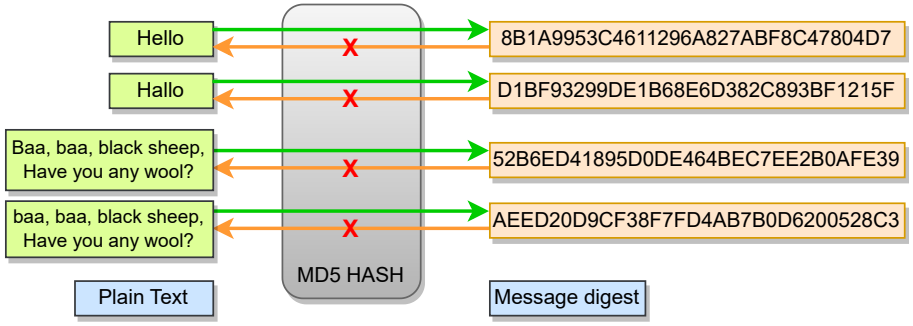


Figure 3.1: Change in output with respect to the change in input of a hash function

3.1.1 Properties of hash functions

A typical hash function is expected to exhibit certain functional properties such as determinism, uniformity, low computational overhead, and must adhere to avalanche properties. For cryptographic applications, hash functions must possess cryptographic properties such as collision resistance, pre-image resistance, and second pre-image resistance. These properties also allow us to classify

hash functions into cryptographic and non-cryptographic hash functions. The important properties of hash functions are listed here.

Deterministic: The hash function should always produce the same output for a given input.

Collision resistance: The collision resistance property ensures that it is hard to find two different inputs that produce the same hash output.

Uniformity: The output of the hash function should be uniformly distributed. Uniformity ensures that the hash collisions are minimal.

Pre-image resistance: Given the hash value h, it should be computationally hard to find the input m such that $h = hash(m)$. The pre-image resistance ensures that the hash function is irreversible and truly a one-way function.

Second pre-image resitance: Given an input m_1, it should be difficult to find another input m_2 such that $hash(m_1) = hash(m_2)$.

Avalanche properties: A good hash function should satisfy the avalanche properties, which means that a small change in the input should result in a significant change in the output. This ensures that the output has sufficient randomness. This property is essential for security applications, as it makes it difficult to predict the output of a hash function.

3.1.2 Cryptographic and non-cryptographic hash functions

Hash functions can be categorized as cryptographic and non-cryptographic hash functions. Depending on their security properties, cryptographic and non-cryptographic hash functions differ in the guarantees they offer. While non-cryptographic hash functions exhibit properties like determinism, collision resistance, and avalanche behavior, they have comparatively weaker cryptographic properties. Cryptographic properties such as pre-image resistance and second pre-image resistance are also not required for non-cryptographic hash functions. However, non-cryptographic hash functions provide significant performance advantages over cryptographic hash functions, surpassing even the fastest cryptographic hash functions. Cryptographic hash functions are typically preferred in applications that demand stringent security measures, like digital signatures, password storage, and authentication. Conversely, non-cryptographic hash functions are favored for applications that prioritize faster execution, such as hash tables, approximate membership query data structures, and checksums.

3.1.3 Applications of hash functions

Some of the popular applications of hash functions in various domains are listed below.

Data integrity: Ensuring that the data has not been tampered with is a challenging task when it comes to larger files or any considerably large form of data. Comparison of similarity between extremely large files or data sets would require significant human/machine effort. Hash functions can be used to prevent any potential tampering or corruption of data during transmission or storage. Hash functions can generate a fixed-sized hash value, such as 256 bits, of any large file/data, making it easier to compare the hash functions of data to confirm the data integrity.

Message authentication: Attacks such as 'Man-in-the-middle' can alter the contents of the messages that are exchanged between two devices. One way to guarantee the authenticity of the messages is by using digital signatures. This involves encrypting the hash value of the message using the private key of the sender to produce a digital signature. The receiver can verify the authenticity of the message by extracting the hash value from the digital signature using the corresponding public key and comparing it with the hash value of the received message.

Password storage: Hash functions are used to securely store passwords by generating a fixed-length hash value from a user's password. This hash value is then stored in a database instead of the original password. In the event of a database breach, the attacker cannot retrieve the original passwords due to the irreversible nature of hash functions.

Key generation: Key generation algorithms utilize hash functions to generate cryptographic keys that are both unique and secure. These keys serve various purposes, such as encryption or authentication.

Indexing and searching: Hash functions are employed in databases or data storage systems like hash tables to efficiently index and search data. They generate a distinct key or fingerprint for each data item, enabling streamlined storage and expedited retrieval of information. These hash-based storage systems reduce storage requirements and ensure faster retrieval of data.

For applications that demand some levels of security, such as ensuring data integrity, message authentication, password storage, and key generation, cryptographic hash functions are required. Conversely, in applications like indexing and searching, speed takes precedence over security considerations, making non-cryptographic hash functions the preferred option.

3.1.4 Hash functions for network security applications

Hash functions are the principal component of approximate/probabilistic data structures. Approximate membership query/lookup data structures provide fast lookup and low memory overhead compared to dictionaries and associative arrays but with a tolerable error. Advantages such as speed and low memory footprint of probabilistic data structures like Bloom filters [148] and sketches [87] make them favorable for fast membership queries and network flow measurement for networking applications. This way, hash functions play a crucial role in network security applications.

In order to serve the security guarantees of secure applications, cryptographic hash functions offer high levels of security. However, certain security properties of cryptographic hash functions such as preimage resistance, second preimage resistance, and collision resistance [177] are of minor importance for probabilistic data structures employed in network security applications. Since the execution speed of the hash computations directly impacts the throughput of the probabilistic data structures, the speed of the hash function becomes a crucial factor. Employing cryptographic hash functions in these data structures would make it difficult to process packets at line rate in high-speed networks. Non-cryptographic hash functions, on the other hand, are faster and more suitable for these applications, such as Bloom filters, hash tables, and sketches. The prime requirement of non-cryptographic hash functions employed in Bloom filters, hash tables, and sketches is that they must be fast, uniformly distributed, and must possess excellent avalanche properties [70, 87, 200, 104]. The latter indicates how well a hash function succeeds in minimizing the correlation between the input text and the generated digest.

Non-cryptographic hash functions in network security applications

The properties of the non-cryptographic hash functions, such as speed and resource-efficiency, make them better suited for approximate/probabilistic membership query data structures compared to cryptographic hash functions. Hence, non-cryptographic hash functions are preferred for these applications, provided they meet the avalanche criteria to ensure no correlation between bit-fields and possess low logical depth for optimal performance. However, recent studies have demonstrated that non-cryptographic hash functions account for a significant portion of the overall resource demands of hardware-based probabilistic data structures [151] in networking applications. High resource and computational overhead of hash functions adversely affect the latency and throughput more often than not, limiting the real-time data processing capabilities in high-speed networks. Therefore, it is crucial to develop

efficient non-cryptographic hash designs specifically tailored for network security applications.

In this chapter, we explore the design of fast non-cryptographic hash functions on FPGA and ASIC in order to serve the membership query and network flow measurement applications for network security. We propose simplifying the existing fast cryptographic ciphers/permutations to function as non-cryptographic hash functions. We propose to use reduced-round versions of cryptographic permutations and symmetric-key ciphers to minimize the computational delay and logical depth while maintaining the non-cryptographic properties. Reducing the number of rounds helps to achieve higher throughput and low latency on hardware. While reducing the number of rounds, it is ensured that the desired avalanche properties are satisfied. The hash function takes a 96-bit flow identifier as input, which consists of a combination of source and destination IP addresses and ports from the IPv4 packet.

3.2 Related work on non-cryptographic hash functions

Most well-known non-cryptographic hash functions possess the required properties, such as uniformity and determinism, and exhibit satisfactory avalanche properties. However, these hash functions are designed to run on conventional software platforms and result in inferior performance on hardware in terms of implementation complexity and speed, for instance, FNV-1a [113] and Murmur3 [104] hash functions. The hardware suitability of these hash functions, like many other non-cryptographic hash functions, is limited as they target efficient execution in software [212]. FNV-1a and Murmur3 employ multiplication as the core operation, which is an expensive operation in hardware in terms of resource utilization and latency. FNV-1a and Jenkins hash [140] process 8 bits per cycle, and Murmur processes 32 bits per cycle, making them well-suited for execution on microprocessors with 8-bit or 32-bit architectures. When implemented in hardware, however, it requires a large number of registers to store the intermediate values and uses multipliers of larger sizes. These requirements have a negative effect on resource utilization, operating frequency, and execution delay, especially when generating large output hash sizes. Moreover, both FNV and Murmur exhibit high latency with larger inputs because of their smaller input block sizes of 8 bits and 32 bits, respectively.

In addition, many of these hash functions have limitations on the size of their output. For instance, SipHash [57] and XORHash [76] are restricted to 16 bits.

The limited output sizes make it difficult to serve our purpose of using a single hash function with a large output size and sample the output to generate multiple hash values. Increasing the hash output size will result in lowering the speed further. Some non-cryptographic hash functions, such as XORHash, exhibit higher operating frequency on FPGA than FNV-1a and Murmur3. However, the overall execution delay for larger input sizes is much higher for high-speed applications such as Terabit Ethernet applications. Additionally, the output hash size of XORHash is also limited to 16 bits. Grochol and Sekanina proposed various versions of NSGAHash [121], which are also capable of achieving a higher operating frequency due to their low logical depth. However, detailed avalanche analysis shows that there exists a very high correlation between the bit-fields [128], as NSGAHash was designed using evolutionary computation based on collision resistance for a specific dataset. Such a design employing collision resistance as the design criterion is highly data-dependent [104] and causes a predictable correlation between the bit-fields. Another possible option is to employ CRC codes as a non-cryptographic hash function because of their excellent collision resistance and their usability in data storage applications. Moreover, CRC can generate larger output sizes, such as 32, 64, or 128. However, the high cost in terms of resources and latency, as presented in the recent works [220, 54], makes them less suitable for large flow detection algorithms.

Due to the high logical depth and computational overhead of cryptographic hash functions, much research has focused on designing lightweight cryptographic hash functions. Cryptographic hash functions based on lightweight symmetric-key algorithms and cryptographic permutations address the issues regarding high computational overhead and poor resource efficiency. These lightweight algorithms provide both area efficiency and security. Nevertheless, these hash functions are typically slower than non-cryptographic hash functions as they require a relatively large number of rounds to satisfy all security requirements. Many recent works on lightweight ciphers employ an FPGA as the target platform for executing these cryptographic algorithms [50, 159, 55, 143]. Most of these implementations are proven to be efficient in terms of resource utilization and power consumption. However, a significant drawback that persists in these implementations is the high latency they exhibit.

The implementation of the Photon lightweight hash function presented in [55] exhibits a minimum cycle count of 12 and a maximum cycle count of 1680. For Spongent hash, the minimum cycle count is 45, and the maximum cycle count is 1980 [159]. Based on these implementations, it can be inferred that a prevalent characteristic among most lightweight cryptographic hash functions is their high latency. High latency is not ideal in high-speed network security applications where processing at line rate is a matter of utmost concern. While reducing the number of rounds in such ciphers compromises their security properties,

it also can lead to faster execution times. As long as the required avalanche properties are satisfied, reduced round ciphers can function as non-cryptographic hash functions, as maintaining stringent security properties is not essential for non-cryptographic hash functions. Hence, reduced-round versions of lightweight cryptographic hash functions that satisfy the avalanche properties are excellent replacements for hardware-friendly non-cryptographic hash functions.

Universal classes of hash functions [77] are commonly employed in network applications but at the expense of increased latency. One approach to mitigating latency in applications like probabilistic data structures involves employing a single hash function with a sufficiently large output size and splitting or sampling the output to generate multiple hash values. An illustration of mapping a 96-input to a memory size of 2^{12} bits is shown in Figure 3.2. When faced with the task of mapping a 96-bit input to a memory space of 2^{12} bits, the inherent challenge arises from attempting to associate 2^{96} potential outcomes with a memory capacity of only 2^{12} bits. Consequently, in conventional approaches such as universal hashing, the issue of collisions becomes significant, necessitating the use of multiple hash functions to alleviate this problem. However, by employing a hash function characterized by perfect avalanche properties that can map a 96-bit input to a 96-bit output space, the probability of collisions can be reduced to near-zero levels. Theoretically, one-to-one functions such as permutations can produce zero collisions, and a hash function with one-to-one mapping and excellent avalanche scores is a good approximation of permutations. This 96-bit hash output can then be effectively utilized to address a memory of size 2^{12} bits by dividing it into 12-bit hash values, as illustrated in Figure 3.2.

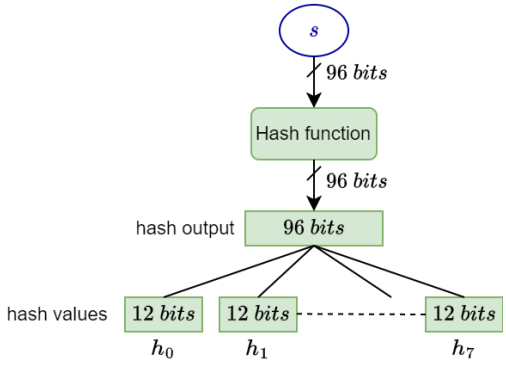


Figure 3.2: Splitting a single hash function to generate hash values

By ensuring the hash function is uniformly distributed, the bit-fields exhibit no correlation, and the collisions are minimal, this technique of using a single

hash function has minimal impact on the false positive rate of probabilistic data structures. Our works on Bloom filters and sketches [212, 215] have shown that this technique can significantly improve the speed, enabling compatibility with Terabit networks without compromising false positive rates or estimation errors. Nevertheless, choosing a suitable non-cryptographic hash function that satisfies the aforementioned requirements is still cumbersome.

In this chapter, we discuss the design of non-cryptographic (NC) hash functions in two parts. The first part (Section 3.3) explains the design and performance evaluation of Xoodoo-NC, which is the NC version of the Xoodoo cryptographic permutation. Subsequently, in the second part (Sections 3.4), we extend this approach to implement a number of non-cryptographic hash functions that are based on reduced-round cryptographic ciphers. These hash functions are evaluated with respect to their avalanche properties, speed, and hardware resource utilization.

3.3 Design and analysis of Xoodoo-NC

To discover a hash function that is suitable for hardware implementation with low logical depth and resource utilization, we initiate our search by considering Xoodoo, a cryptographic permutation introduced by Daemen et al. in [89]. A Xoodoo round employs only shift and boolean operations, which require only negligible computational ability on hardware. While Xoodoo uses a 384-bit state, the non-cryptographic version Xoodoo-NC operates on a 96-bit state, which perfectly fits our requirements to process incoming network flow IDs of 96 bits. Output values can be any multiple of 96 bits. A more detailed explanation of Xoodoo-NC is provided in the subsequent paragraphs.

The original Xoodoo permutation is parameterized by the number of rounds n_r, and it iteratively applies the round function R_i to the state A. The state A is depicted in Figure 3.3 and has a size of 48 bytes, divided into three 2-dimensional planes, which are indexed by y. Each plane has four 1-dimensional lanes, which are indexed by x. Each lane consists of 32 bits in size, indexed by z. A collection of three parallel stacked lanes in A is called a sheet, and any three parallel stacked bits in A are called a column. Each bit in the state can thus be referred to with the coordinates (x, y, z).

The round function R_i of Xoodoo consists of 5 sequential steps: a mixing step θ, a plane shifting step ρ_{west}, a step for the addition of round constants ι, a non-linear layer χ, and a second plane shifting step ρ_{east}. For more in-depth details on these steps and the round constants, we refer to [89].

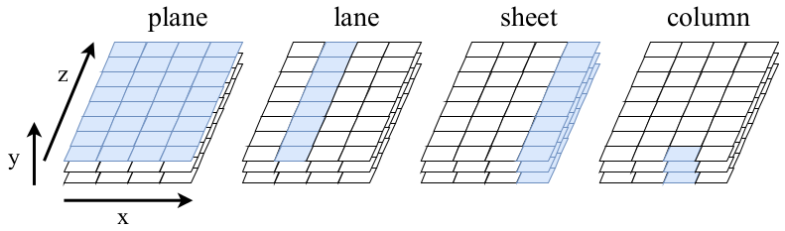


Figure 3.3: Graphical representation of the Xoodoo state

Considering a flow ID size of 96 bits, it is required to have a hash function that accepts a 96-bit input. Therefore, we reduce the state A of the Xoodoo permutation to 96 bits, corresponding to one sheet. This fixes the x coordinate, allowing the y coordinate to vary from 0 to 2 and the z coordinate from 0 to 31. The original definitions of lanes and columns still hold. Because of the reduction of the size of the state, the Xoodoo round function is slightly adapted. The cyclic shift operations involving the x and z coordinates are replaced by operations involving only z. The other operations and the round constant C_i remain the same as in the original algorithm. The resulting Xoodoo-NC permutation is specified in Alg. 1, and the notational conventions of Xoodoo-NC are specified in Table 3.1.

3.3.1 Avalanche behavior of Xoodoo-NC

The avalanche characteristics of the Xoodoo permutation are summarized in Table 8 of the original article [89]. The avalanche behavior of Xoodoo-NC is also determined here using the same avalanche metrics presented in the original article for Xoodoo. These metrics include: avalanche dependence (D_{av}), avalanche weight ($\overline{w}_{av}$), and avalanche entropy (H_{av}), which are defined below.

Avalanche dependence: For a single-bit change in the input, the number of bits in the output that may flip defines the avalanche dependence. The avalanche dependence is defined as:

$$D_{av} = n - \sum_{i} g(p[i]) \tag{3.1}$$

Here n is the number of bits in the output, p is the probability vector, and $p[i]$ represents the probability that the bit i of the output flips for a single bit change in the input. $g(p) = 1$ if $p = 0$ and $g(p) = 0$ otherwise.

Algorithm 1: Xoodoo-NC[n_r]

Parameters: Number of rounds n_r
for *Round index* i *from* $1 - n_r$ **to** 0 **do**
  $A = R_i(A)$; // A is 96-bit input state (1 sheet)
end
  Round R_i:
    θ:
      $P \leftarrow A_0 \bigoplus A_1 \bigoplus A_2$
      $E \leftarrow P \lll 5 \bigoplus P \lll 14$
      $A_y \leftarrow A_y \bigoplus E$ for $y \in \{0, 1, 2\}$
    ρ_{west}:
      $A_2 \leftarrow A_2 \lll 11$
    ι:
      $A_0 \leftarrow A_0 \bigoplus C_i$
    χ:
      $B_0 \leftarrow \overline{A_1}.A_2$
      $B_1 \leftarrow \overline{A_2}.A_0$
      $B_2 \leftarrow \overline{A_0}.A_1$
      $A_y \leftarrow A_y \bigoplus B_y$ for $y \in \{0, 1, 2\}$
    ρ_{east}:
      $A_1 \leftarrow A_1 \lll 1$
      $A_2 \leftarrow A_2 \lll 8$

Table 3.1: Notational conventions of Xoodoo-NC

A_y	Lane y of state A. In Xoodoo-NC, A is one sheet.
$A_y \lll v$	Cyclic shift of A_y moving bit in z to position $(z + v)$
$A_y \bigoplus B_y$	Bitwise sum (XOR) of planes A_y and B_y
$A_y.B_y$	Bitwise product (AND) of planes A_y and B_y
$\overline{A_y}$	Bitwise complement of lane A_y

The Avalanche dependence is satisfied when $D_{av} = n$ for all outputs as a result of a single bit change at the input.

Avalanche weight: The avalanche weight generalizes the avalanche criterion and is defined as the expected Hamming weight of the output difference for a single bit change in the input. It is defined as:

$$\overline{w}_{av} = \sum_i p[i] \tag{3.2}$$

The condition is satisfied when the avalanche weight is equal to 50% of the number of output bits i.e., $w_{av} \approx \frac{n}{2}$, for all inputs with a single bit change (Hamming weight = 1).

Avalanche entropy: The avalanche entropy represents the uncertainty of whether the bits in the output flip or not for a single-bit change in the input. It is defined as:

$$H_{av} = \sum_{i}(-p[i].log_2(p[i]) - (1-p[i]).log_2(1-p[i])) \tag{3.3}$$

The H_{av} generalizes the strict avalanche criterion. The avalanche entropy is said to be satisfactory if it fulfils the condition $H_{av} \approx n$ for all inputs with a single bit change.

Table 3.2: Avalanche Scores for Xoodoo-NC

Rounds	D_{av}	$\overline{w}_{av}$	H_{av}
2	84	35.408	80.332
2.5	96	47.324	95.864
3	96	47.309	95.867
3.5	96	47.922	95.996

The avalanche metrics are calculated for single-bit differences at the input. In our computation, only the worst-case values are reported. The computed results of D_{av}, $\overline{w}_{av}$, and H_{av} according to equations 3.1, 3.2, and 3.3 are presented in Table 3.2. Xoodoo-NC provides full bit-dependence and quasi-strict avalanche weight after 2.5 rounds. Half a round of Xoodoo is formed by ρ_{east}, θ, and ρ_{west}, which is in the phase reversed sequence. This half-round is also called the linear layer, denoted by λ [89]. Xoodoo-NC exhibits $\approx 90\%$ dependence even after two rounds. We consider 2.5 rounds sufficient for our intended application and generate a 96-bit Xoodoo-NC hash output after 2.5 rounds. To obtain an output of 192 bits, one additional round is executed, and the output of the additional round is concatenated with the current round's output. Likewise, an output size of any multiple of 96 bits can be obtained by increasing the number of rounds and concatenating the results.

3.3.2 Hardware evaluation of Xoodoo-NC

The hardware block diagram of Xoodoo-NC is shown in Figure 3.4. The core of the Xoodoo permutation is the round function, which consists of the five steps shown in Algorithm 1: θ, ρ_{west}, ι, χ, and ρ_{east}. The round steps are applied sequentially n_r times on the input state function A. To obtain the output as multiples of 96, additional rounds, equal to a concatenation factor (C_f), are executed. If the C_f is 1, one additional round is executed to obtain a 192-bit

output. This way, the output of Xoodoo-NC can be any multiple of 96 bits, but we stick to two different output sizes in our evaluation: 96 and 192 bits.

The rounds are implemented in a fully unrolled architecture without registers in between the rounds, i.e., the 2 or 3 rounds of Xoodoo-NC are entirely implemented in combinational logic. The detailed hardware block diagram of the round function is shown in Figure 3.5. The 96-bit input A consists of 3 lanes of size 32 bits, namely A_0, A_1, and A_2 (cf. Alg. 1). The round constant C_i used in ι is a single 32-bit lane, indexed from $1 - n_r$ to 0. The 3-lane output of the round goes to the input of the next round.

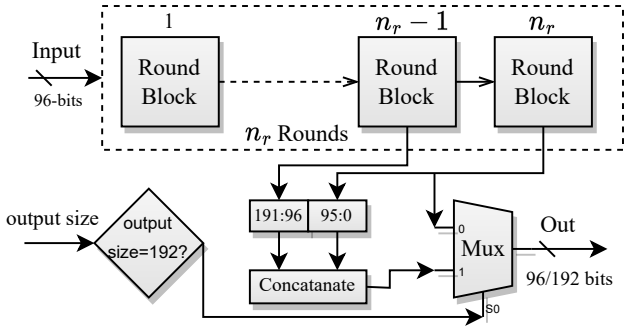


Figure 3.4: Xoodoo-NC block diagram

Table 3.3: Comparison of hash implementations on a Xilinx Virtex Ultrascale+

Function	Block size (In/Out bits)	LUT	FF	BRAM	DSP	Frequency	Latency (ns)	Throughput (Gbps)
FNV-1a [113]	96/32	38	70	0	2	211.01 MHz	61.6	1.56
	96/64	76	161	0	7	160.23 MHz	81.1	1.18
Murmur3 [104]	96/32	574	277	0	24	153.87 MHz	65.0	1.48
	96/64	566	341	0	38	120.58 MHz	82.9	1.16
Xoodoo-NC 2 rounds	96/96	256	194	0	0	666.67 MHz	1.5	64
Xoodoo-NC 3 rounds	96/192	323	290	0	0	416.67 MHz	2.4	40

When it comes to low latency and the need for a larger output hash size, Xoodoo-NC has proven to outperform FNV-1a and Murmur3. Table 3.3 summarises the hardware implementation results of FNV-1a, Murmur3, and Xoodoo-NC on a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVB2104-2-i) using Vivado 2017.4 design tool [210]. Optimized versions of FNV-1a and Murmur3 hash functions are implemented with output hash sizes of 32 and 64 bits. The throughput is calculated using the equation $\frac{Input\ block\ size}{latency\ in\ cycles} \times f_{max}$, where f_{max} is the maximum operating frequency. The table clearly shows that despite

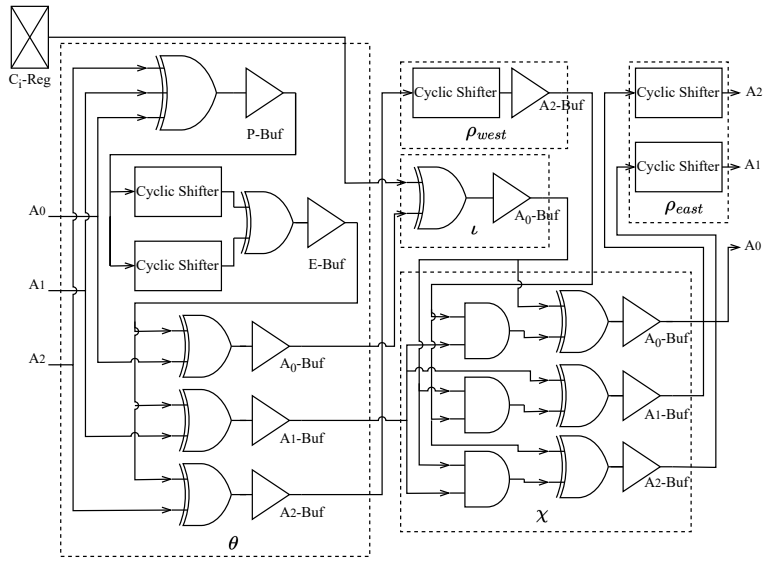


Figure 3.5: Xoodoo-NC Round function

having a larger output block size, Xoodoo-NC has significantly improved the frequency of operation and latency compared to both Murmur3 and FNV-1a. Murmur3 and FNV-1a also leverage DSP slices to improve performance as they include multiplication operations. Our preliminary research showed that the operating frequency decreases when DSP blocks are employed in our hash functions, leading us to exclude DSP blocks from our design.

3.4 Implementation and analysis of cryptographic cipher-based NC hash functions

In this section, we focus on reduced-round, reduced-logic implementations of non-cryptographic hash functions derived from existing lightweight ciphers. For our implementation and analysis, we arbitrarily choose eight symmetric-key ciphers. The symmetric key ciphers are Pyjamask [118], GIFT [60], SKINNY [65], AES [98], SPECK [63], PHOTON [61], Sparkle [64], and TinyJAMBU [242]. TinyJAMBU is a stream cipher, whereas all the others are block ciphers.

3.4.1 Algorithmic details of the lightweight cryptographic ciphers

This section provides a concise summary of the algorithmic details for the selected ciphers. It is to be noted that any special modifications made to the round functions in the first or last round of these algorithms are not discussed or implemented here. Only the regular round function of the ciphers is considered. We forward the interested reader to the specifications in the original articles for a more comprehensive description. Some of the ciphers have a block size of 96 bits, which aligns with our objective of designing 96-bit non-cryptographic hash functions. PHOTON and Sparkle employ a 256-bit block size, but for our design requirements, the block size is reduced to 144 and 128 bits while preserving the avalanche properties. For the ciphers having 128-bit or 144-bit block sizes, the avalanche properties displayed in Section 3.4.2 are calculated for the entire 128-bit/144-bit inputs. In the hardware implementation, the input size is taken as 96 bits, and the remaining bits are tied to zero. The output hash size remains the same.

SPECK

The SPECK block cipher family holds ten different variations of the SPECK algorithm, having block sizes of 32, 48, 64, 96, and 128 with varying key sizes ranging from 64 to 256. This work focuses only on the cipher with a block size and key size of 96 bits. SPECK is an add-rotate-xor cipher, relying exclusively on these three operations to calculate the ciphertext. SPECK with a block size and key size of 96 bits consists of 28 rounds. During the execution of rounds, the key and the plaintext are split into two parts of 48 bits. The most significant bits of the plaintext (PT) and the key (K) are stored in PT_1 and K_1, respectively, while the least significant bits are stored in PT_2 and K_2.

Round function: Each round performs three operations in SPECK: Rotation, Addition, and XOR. Initially, PT_1 is rotated 8 bits to the right, followed by an addition with PT_2. Subsequently, PT_1 is XORed with K_2. For PT_2, the operations commence after the addition with PT_1, involving a 3-bit left rotation followed by an XOR with PT_1. A visual representation of this process for a single round can be observed in Figure 3.6. After each round, the output is connected to the input of the next round.

Key schedule: In SPECK, the key varies with each round, while the round function operates similarly to the encryption round for the plaintext. The only distinction lies in the XOR operation, where the second operand is the round number, starting from 0 (the first round) and incrementing by 1 in every round.

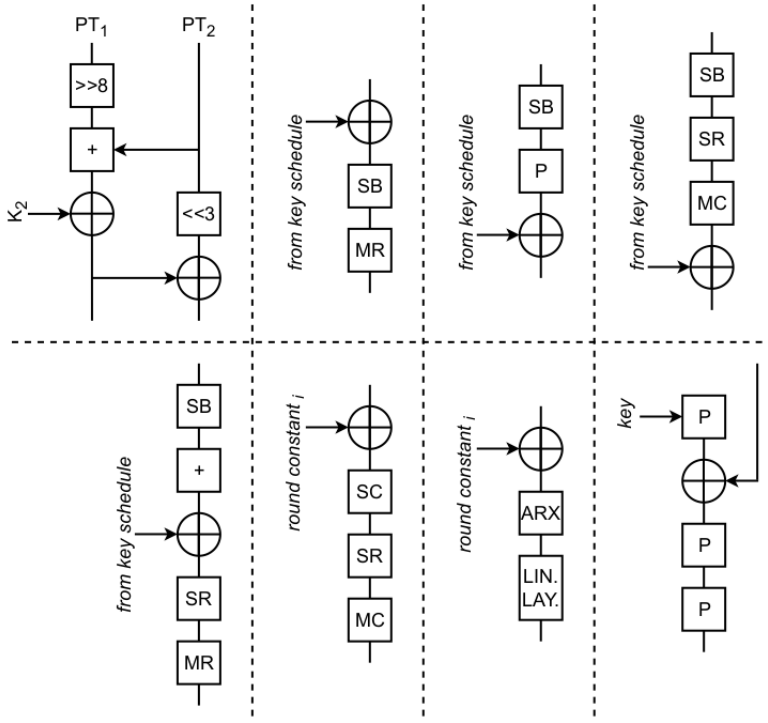


Figure 3.6: Functional representation of a single round of the eight chosen ciphers: SPECK, Pyjamask, GIFT, AES, SKINNY, PHOTON, Sparkle, and TinyJAMBU.

Pyjamask

The Pyjamask block cipher family [118] comprises two algorithms that can handle block sizes of either 96 bits or 128 bits. These algorithms utilize a 128-bit key, undergo 14 rounds, and rely on a Substitution-Permutation Network (SPN) structure to transform the plaintext into the ciphertext. The plaintext is structured in a bit-by-bit left-right, top-down structure, where each row consists of 32 bits. Each row R_i starts from $32 \times i$ and increments by 1, where $i \in 0, 1, 2, 3$. Depending on the block size of the chosen algorithm, 3 or 4 rows are used. The cell which holds the lowest index represents the most significant bit of the plaintext, and the cell with the highest index represents the least significant bit. The key is represented in the same way as the plaintext but with four rows, each containing 32 bits. Once the plaintext enters the algorithm, it will be referred to as the internal state.

Round function: Each round is composed of three operations, which happen in sequential order: AddRoundKey, SubBytes, and MixRows. AddRoundKey first XORs n bits of the key with the internal state. n is 128 for both Pyjamask-128 and Pyjamask-96. However, in the case of the latter, only the 96 most significant bits are addressed. The result of the XOR is passed through an Sbox. Finally, the internal state is multiplied with a constant matrix.

Key schedule: Throughout the algorithm, different keys are used in each AddRoundKey. These keys, known as subkeys, are derived from the original secret key. To receive these subkeys, three operations are executed in each round on the current round key: MixColumns, MixRows, and Constant Addition. Because Pyjamask-96 and Pyjamask-128 use 128-bit keys, the same operations are used to create the subkeys.

GIFT

The GIFT block cipher [60] family contains two algorithms: GIFT-64 and GIFT-128. Both use a key size of 128 bits, while the former uses a block size of 64 bits and the latter of 128 bits. In this paper, GIFT-128 is used. Like Pyjamask, GIFT is an SPN cipher containing 40 rounds.

Round function: Each round of GIFT is based on three operations, namely SubCells, PermBits, and AddRoundKey. Similar to SubBytes in Pyjamask, the SubCells operation compares each 4-bit column in the cipher state to an Sbox and is replaced by the appropriate value. The PermBits operation shuffles all the bits in the internal state according to a specific permutation. Finally, the AddRoundKey operation adds the roundkey to the internal state through an XOR.

Key Schedule: GIFT follows a simple key schedule operation to minimize the area in hardware. The key is split into blocks K_i of 32 bits each as in the AddRoundKey operation, where $i \in \{0, 1, 2, 3\}$. K_0, K_1, and K_2 are then shifted toward the least significant bits of the key state, while K_3 now holds the most significant bit. K_3 is then split again into two 16-bit blocks, $K_{3,1}$ and $K_{3,2}$. These two blocks are then right-rotated over 2 and 12 positions, respectively.

AES

AES is the most popular and widely recognized cipher. It has a block size of 128 bits. The key size can be 128, 192, or 256 bits, determining the number of rounds, 10, 12, and 14, respectively. This work focuses on the algorithm with a key size of 128 bits. Similar to GIFT, the input plaintext is structured in a

top-down, left-right manner. Similarly, the key also adheres to this structured arrangement.

Round function: Each round consists of 4 operations: AddRoundKey, SubBytes, ShiftRows, and MixColumns. AddRoundKey performs a simple bitwise XOR of the key with the internal State. Each round uses a different round key. These round keys are calculated in the key schedule. In the SubBytes step, each byte in the internal State is replaced through a Sbox lookup. After the Sbox, the rows of the State are shifted cyclically in a specific manner. Finally, in the MixColumns step, matrix multiplication is done. The State is updated with the resulting product.

Key schedule: The key schedule generates a new round key, starting from the initial key. Every key is split into four equally sized parts. One of these parts is rotated and subsequently passes another Sbox. The result of the Sbox is then XORed with a round constant and another part of the incoming key. The last two parts are finally XORed in turn with the previous result.

SKINNY

The SKINNY block cipher family holds six algorithms, namely SKINNY-64-64, SKINNY-64-128, SKINNY-64-196, SKINNY-128-128, SKINNY-128-256, and SKINNY-128-384. The first numerical value in the name is the block size used in the algorithm, and the second value is the key size. In this paper, we focus only on SKINNY-128-128. The plaintext and key are initialized similarly to AES but in a left-right top-down manner. The number of rounds of the SKINNY cipher varies with the algorithm used. For SKINNY-128-128, the number of rounds required is 40.

Round function: Each round consists of 5 operations: SubBytes, AddConstants, AddRoundKey, ShiftRows, and MixColumns. The SubBytes operation follows the same operation of SubBytes in AES, the only difference being the values in the Sbox. In the AddConstants operation, the round constants are XORed to the first column of the internal State. The round constant is a 6-bit number and is generated using an LFSR, and the round constant changes in every round. The key State is added to the internal State in the AddRoundKey operation. Unlike the AddRoundKey function in AES, only the eight most significant bytes of the key are XORed to the eight most significant bytes of the internal State. The ShiftRows function works the same way as the ShiftRows function in AES, the only difference being that the rows are shifted in the opposite direction. The MixColumns function is the final round operation. Similar to MixColumns in Pyjamask and AES, each column of the internal State is multiplied by a constant matrix M.

Key schedule: Just like other ciphers, the key is updated in every round into different subkeys. In SKINNY, the key updates are similar to the key schedule in GIFT.

PHOTON

We use the underlying PHOTON_{256} [123] permutation in PHOTON-Beetle to construct a non-cryptographic hash function. The number of rounds is 12, and the input state to the permutation is represented as an 8x8 matrix, with each element consisting of 4 bits, leading to an input size of 256 bits. However, to meet our requirements, we trim down the input state from 8x8 to 6x6 matrix, and the corresponding elements from the Sbox, RoundConstant matrix, and MixColumn matrix are ignored. The MixColumn matrix of size 8x8 is also trimmed down to a 6x6 matrix, and the remaining constant elements are ignored. The Round Constant matrix of size 8x12 is trimmed down to a size of 6x12.

Permutation (Round function): Each round of the permutations consists of four operations: AddConstant, SubCells, ShiftRows, and MixColumn. The AddConstant operation is very similar to the AddConstant/AddRoundKey operations of the abovementioned algorithms. In each round, a 4-bit constant from the Round Constant matrix of size 8x12 is XORed with all the elements in the first column of the input state matrix. Similar to the SubBytes operation stated in other algorithms, the SubCells operation replaces the elements in the input state with the values from the Sbox. The ShiftRows and MixColumn operations are also similar, where the elements are shifted cyclically during the ShiftRows operation and multiplied with constant matrix M during the MixColumn operation. The PHOTON permutation does not use a key, but uses pre-stored round constants in each round.

Sparkle

The Sparkle family consists of 3 permutations: SPARKLE256_{n_s}, SPARKLE384_{n_s}, and SPARKLE512_{n_s}, each having block sizes of 256, 384, and 512 bits, respectively. n_s denotes the number of permutation steps and is defined for any $n_s \in N$. Sparkle uses *slim* and *big* instances, and these instances differ only in terms of the number of steps of the permutations, where n_s=7 for the *slim* instance and $n_s = 10$ for the *big* instance. n_b defines the number of branches where each branch represents a block of 64 bits. n_b must be even, and Sparkle uses 4, 6, or 8 branches, and 8 is the maximum number of branches that can be used. We employ the *slim* instance of SPARKLE256 permutation, but to

reduce the output size of the hash, we take n_b=2, which keeps the number of output bits to 128.

Permutation (Round function): The Sparkle permutation is a classical Substitution-Permutation Network (SPN) construction in which the S-boxes are different in each branch. The 64-bit block cipher *Alzette* plays the role of the S-box, which is iterated in each step of the permutation. The constants (keys) used in Alzette are branch dependent. Alzette (ARX-box layer) is followed by a *linear layer* in each step. A step-dependant constant is XORed to the state at the beginning of each step prior to the ARX-box layer and linear layer.

Key schedule: The Sparkle permutation does not use a key, but the 64-bit block cipher Alzette, employed as the ARX-box layer, uses a 32-bit key. The keys are pre-stored as round constants. They are different in each step and dependent on the number of branches.

TinyJAMBU

TinyJAMBU is specified for three different key sizes: 128, 192, and 256 bits. Each variant uses a 96-bit nonce, 128-bit state, and 64-bit authentication tag. TinyJambu employs a keyed permutation P_n, where n defines the number of rounds. In each round of the permutation, a 128-bit non-linear feedback shift register (NFSR) is used to update the state. The permutation uses two different rounds, N_1 and N_2, for state update during different stages, where N1 is 640 (P_{640}) and N2 is 1024 (P_{1024}). Since TinyJAMBU is a stream cipher, the avalanche properties cannot be satisfied when using the plaintext block as the input to the hash function where the plaintext is XORed only once to the generated keystream. To make use of TinyJAMBU as a non-cryptographic hash function, the *Associated Data (AD)* input of TinyJAMBU is used as the input to the hash, and the plaintext value is tied to 0. The authentication tag or Message Authentication Code (MAC) is used as the output, and it can be any multiple of 32 bits.

Round function: The TinyJambu-128 keystream generation and encryption round consists of 4 stages: initialization, associated data processing, encryption, and finalization. The initialization stage consists of a key setup and a nonce setup phase. The key setup sets the 128-bit State to zero, and the State is updated with the keyed permutation using N2 rounds. The nonce setup consists of three rounds. In each round, a constant *FrameBitsIV* is XORed with the State, followed by a state update using the keyed permutation with N1 rounds, and an XOR of the 32-bit nonce with the State. The associated data (AD) is our hash input, and there are $\frac{adlen}{32}$ steps in AD processing, where *adlen* is the length of AD in bits. In each step of AD processing, a constant *FrameBitsAD* is

XORed with the State, followed by a state update using the keyed permutation with N1 rounds, and finished by XORing the 32 bits of AD with the State. Two different steps are followed for full blocks (32 bits) and (possible) partial block (if the last block is not a full block) of AD. In the encryption stage, there are $\frac{mlen}{32}$ steps, where $mlen$ is the length of the plaintext in bits.

The encryption stage is similar to AD processing, with the difference being the constant *FrameBitsPC*. The ciphertext is generated by XORing the plaintext with a part of the state. In our implementation, we do not generate the ciphertext, and the plaintext value is taken as 0. However, the length of the plaintext determines whether the encryption stage needs to be executed. Hence, keeping the $mlen$ to zero ignores the encryption stage, whereas keeping $mlen$ to non-zero helps achieve the avalanche properties with fewer rounds. The finalization or MAC stage involves XORing the state with the constant *FrameBitsFinalization* followed by a state update using the keyed permutation with N1 rounds, and finally assigning state[64:95] to the first 32 bits of the tag. This step is repeated with N2 rounds of permutation to generate the next 32 bits of the tag. Since the tag is taken as our hash output, we have repeated this step once more to generate the 96-bit tag output. To optimize the implementation, 128 rounds of the permutation are performed in parallel, and the values of N1 and N2 are taken as 5 and 8, respectively, instead of 640 and 1024.

Key schedule: TinyJAMBU employs a 128-bit key for updating the state by applying the keyed permutation P_n. Each message must be associated with a unique key-nonce combination.

3.4.2 Analysis of avalanche properties

The previously mentioned ciphers have proven to be secure but not fast enough for high-speed networks. Our objective is to utilize these ciphers as hash functions by reducing the number of rounds, while also examining their avalanche properties [111] to ensure they are satisfactory. The use of keys does not impact the avalanche properties but allows us to easily modify the hash function by altering the key value. The avalanche performance can be assessed by analyzing to which extent the output value changes for a slight change in the input of the hash function. If the hash function exhibits poor avalanche performance, it indicates inadequate randomization, making it vulnerable to potential attackers.

In our analysis, the worst-case values are considered to ensure that the avalanche metrics are satisfactory in any given condition. To have high precision for the avalanche metrics, we set the number of iterations M as large as 250'000. The expected standard deviation of each element in the probability vector p is determined to be $\frac{1}{\sqrt{M}}$ [89]. In each iteration, all n bits are flipped one by

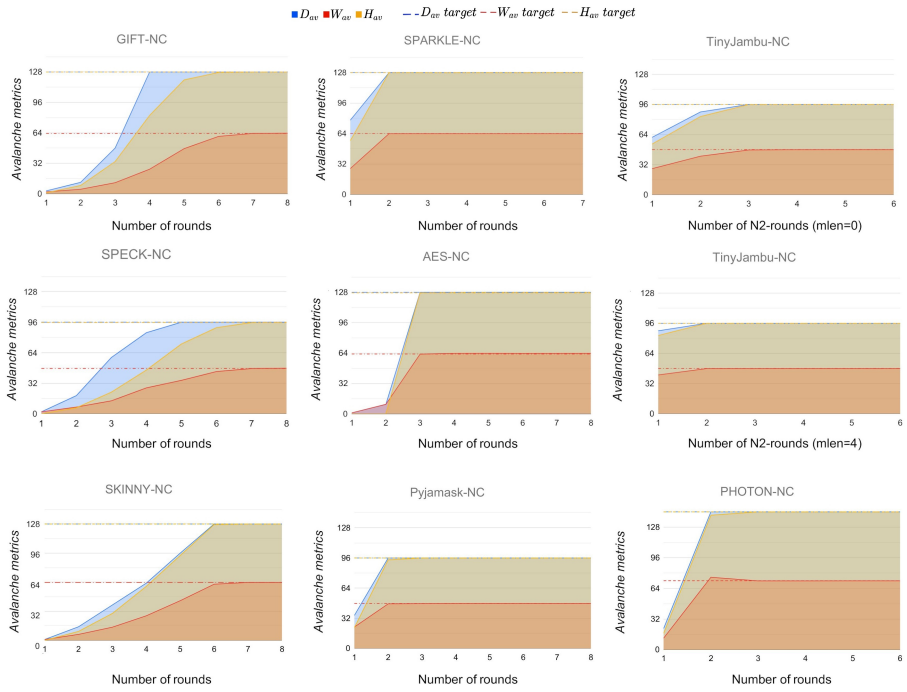


Figure 3.7: The avalanche metrics for SPECK-NC, Pyjamask-NC, GIFT-NC, AES-NC, SKINNY-NC, PHOTON-NC, Sparkle-NC, and TinyJAMBU-NC respectively. The areas show the obtained metric while the horizontal lines indicate the target. The legend indicates which colors are the Avalanche Dependence (D_{av}), the Avalanche Weight (w_{av}), and the Avalanche Entropy (H_{av}).

one, and the avalanche metrics are calculated. This approach ensures that the avalanche metrics obtained are stable and precise.

The plot of the avalanche metrics of each cipher as a function of the number of rounds is shown in Figure 3.7. We append 'NC' (Non-Cryptographic) to the cipher name to indicate the reduced-round versions of the ciphers as non-cryptographic hash functions. While SPECK has 28 rounds, SPECK-NC only requires seven rounds to achieve satisfactory avalanche properties. Similarly, Pyjamask typically requires 14 rounds, but the avalanche metrics plot shows that Pyjamask can meet the avalanche properties with only 2 or 3 rounds. GIFT-NC requires only 6 to 7 rounds to achieve the avalanche property requirements, as shown in the figure, in contrast to the actual 40 rounds of GIFT. The number of

rounds to have sufficient avalanche properties for AES-128 can also be reduced from 10 to 3 rounds in AES-NC. Similar to GIFT, SKINNY-128-128 also employs 40 rounds but requires only six rounds to meet the avalanche criteria in SKINNY-NC. The cryptographic PHOTON permutation requires 12 rounds, but a mere three rounds are sufficient to meet the avalanche requirements. Impressively, even with just two rounds, D_{av} of PHOTON is 144, and the values of W_{av} and H_{av} are satisfactory with 75.64 and 140.53, respectively. The number of rounds for Sparkle can be reduced from 7 to 2 to meet the avalanche properties.

TinyJAMBU is a stream cipher, and the avalanche calculations slightly differ from the calculations for the block ciphers. Even though we have tied the plaintext value to 0, the length of the plaintext, *mlen*, plays an important role in the output generation. If *mlen* is equal to 0 bytes, the encryption stage, as mentioned in Section 3.4.1, is ignored. If *mlen* is greater than zero, the encryption stage is entered. Hence we have plotted the avalanche properties for $mlen = 0$ and $mlen = 4$ bytes. Instead of using N_1 and N_2 to define each round as mentioned in Section 3.4.1, we have accumulated all the rounds, each round consisting of 128 rounds of the permutation performed in parallel. This work defines one round of TinyJAMBU as a 128-round permutation performed in parallel. $mlen = 8$ or $mlen = 12$ is not plotted because only one round of N_1 and N_2 is enough to achieve the required avalanche properties. Based on our analysis, incrementing N_1 has a negligible impact on improving the avalanche properties. Therefore, we have only incremented N_2 while calculating the avalanche properties. The chart shows only the incremental rounds of N_2. The cryptographic version of TinyJAMBU requires a total of 80 rounds ($N_1 = 5, N_2 = 8$) for *mlen* equal to 12 bytes, whereas only 12 rounds for $mlen = 8$ are required to satisfy the avalanche criteria.

The findings of this analysis indicate that we can use reduced-round versions of cryptographic ciphers as non-cryptographic hash functions. The number of rounds can be reduced by 70-85% while maintaining excellent avalanche properties. With a reduced number of rounds, the latency can be improved and the hardware resource requirements can be kept to a minimum. A summary of the avalanche metrics analysis as a function of the number of rounds is shown in Table 3.4. For TinyJAMBU, the accumulated rounds of 128-round permutations performed in parallel are given for r and r_{av} in the Table. Normalized values of w_{av} and H_{av} with respect to D_{av} are also presented for ease of comparison.

Table 3.4: The number of rounds needed for meeting the avalanche criteria

Cipher	D_{av}	w_{av}	H_{av}	r	r_{av}	Normalized w_{av}	Normalized H_{av}
SPECK-NC (96-bit)	96	47.45	95.68	28	7	49.42%	99.67%
Pyjamask-NC (96-bit)	96	47.73	95.96	14	3	49.71%	99.95%
TinyJAMBU-NC (96-bit)	96	47.58	95.89	80	12*	49.56%	99.89%
GIFT-NC (128)	128	63.39	127.95	40	7	49.52%	99.96%
AES-NC (128-bit)	128	63.0	127.43	10	3	49.21%	99.55%
SKINNY-NC (128-bit)	128	63.72	127.95	40	7	49.78%	99.96%
Sparkle-NC (128-bit)	128	63.86	127.99	7	2	49.89%	99.99%
PHOTON-NC (144-bit)	144	71.84	143.99	12	3	49.89%	99.99%

r: Number of rounds in the original cipher

r_{av}: Number of rounds to meet avalanche criteria

*Accumulated rounds for $N_1 = 1, N_2 = 1, mlen = 8$

3.4.3 Hardware Evaluation

The evaluation of the non-cryptographic hash functions on FPGA and ASIC are provided in the following sections.

Experimental setup

The non-cryptographic hash functions are implemented on an FPGA and synthesized for ASIC to evaluate the resource requirements and operating speed. The evaluation is performed on both Zynq and Virtex Ultrascale+ FPGAs using Xilinx Vivado 2020.2 design tool. The results reported in the tables exclusively pertain to the implementation outcomes on the Virtex Ultrascale+ XCVU7P-FLVB2104-2-i device. On ASIC, the non-cryptographic hash functions are synthesized for TSMC 65nm.

The general block diagram of the hardware implementation of the reduced-round ciphers as hash functions is shown in Figure 3.8. This figure illustrates the unrolled implementations with only a register before and after the rounds. Pipelining is not applied in the implementation, and the whole execution of all the rounds is completed in this single clock cycle. This configuration ensures the worst-case operating frequency. Nevertheless, if the operating frequency has precedence over single clock cycle latency, it is possible to employ multiple parallel multicycle paths to achieve a higher operating frequency. The plaintext is the input to the hash function, and the resulting ciphertext is the hash output. While the inclusion of a key is not essential, its inclusion does not impact the avalanche properties. In textbook hash functions, the digest should always be the same, given the same input. However, certain applications benefit from having multiple distinct hash values for the same input, such as in Bloom filters. Adding a key to the block cipher allows the generation of multiple independent

hash values using the same hash function but with different keys. In hardware implementations without the key, we eliminated all the computational logic involving the key and manually removed all the hardware logic responsible for processing the key.

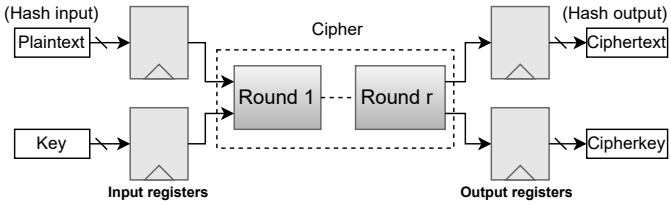


Figure 3.8: General hardware setup of hash implementation

Results and analysis on FPGA

The performance of the implemented non-cryptographic hash functions is evaluated mainly in terms of maximum operating frequency, latency, resource requirements, and throughput. The number of rounds r for each cipher has been chosen corresponding to the number of rounds required to meet the sufficient avalanche properties (see Table 3.4). Furthermore, a comparison is performed between the results obtained and several established non-cryptographic hash functions.

Latency and throughput: To analyze the timing, the worst negative slack is plotted against varying clock periods and is shown in Figures 3.9 to 3.16. Each cipher's timing results are plotted with and without the key. In general, the implementation with the key is slightly slower than that without the key. This is because of the fact that all the calculations involving the key are completely eliminated in the implementation without the key. It is also evident from the plots that the performance is also dependent on the FPGAs, as the latency is lower for Virtex Ultrascale+. Table 3.5 presents the maximum possible frequency for each hardware implementation. Among the ciphers, GIFT-NC and PHOTON-NC show the best operating frequency and prove to be faster than the other ciphers. With and without using the key, the variations in latency of GIFT-NC, Pyjamask-NC, and SKINNY-NC are negligible. However, the inclusion of the key in TinyJAMBU adds a 33% increase in latency, whereas SPECK-NC and AES-NC add around 17% latency, which is caused by additional computational steps and the delay involving the key.

The throughput of the hash function is dependent on the latency and input block size. To calculate the throughput in this context, we use the equation

$\frac{Block\ size}{latency\ in\ cycles \times T_{min}}$, where T_{min} is the minimum clock period. The throughput is of the order of ten thousand as the latency is only a single clock cycle.

Resource requirements: Table 3.6 illustrates the utilization of resources for achieving the best possible latency. GIFT-NC proves to be the most resource-efficient of all the compared ciphers when the key is included. SPECK-NC utilizes the lowest resources when the key is not included. On the other hand, AES-NC and SKINNY-NC are the most expensive in terms of hardware resources while offering moderate operating frequency. Although SPECK-NC exhibits efficient resource utilization without the key, it requires ≈3× more resources when the key is added. GIFT shows the least variation in resources, whether the key is added or not.

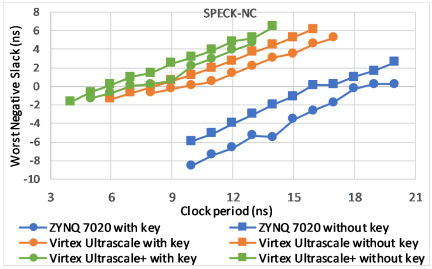


Figure 3.9: Timing analysis of SPECK-NC

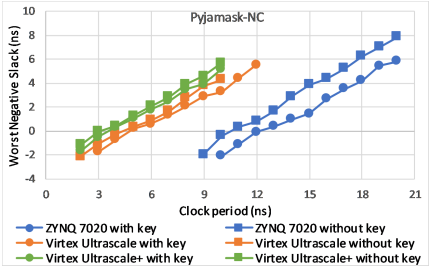


Figure 3.10: Timing analysis of Pyjamask-NC

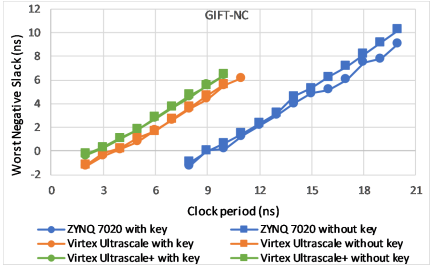


Figure 3.11: Timing analysis of GIFT-NC

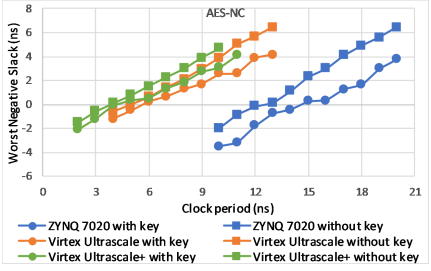


Figure 3.12: Timing analysis of AES-NC

Comparison with related work: This work focuses on reduced-round, reduced-logic implementations of non-cryptographic hash functions based on existing ciphers. Hence, a straightforward comparison with the existing full-fledged implementations of the same ciphers on FPGA is not fair. For example, the throughput of the FPGA implementation of AES-128 and SPECK-96

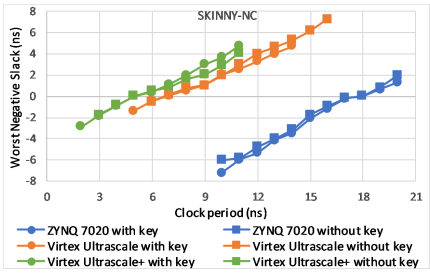


Figure 3.13: Timing analysis of SKINNY-NC

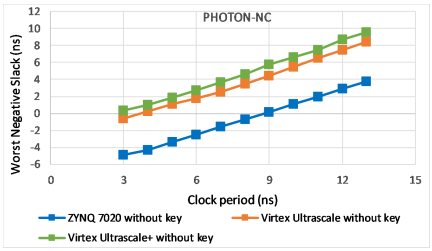


Figure 3.14: Timing analysis of PHOTON-NC

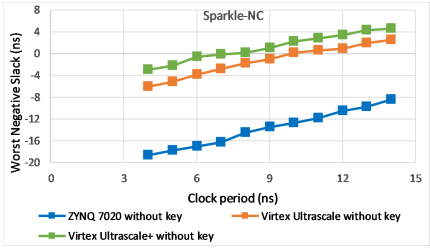


Figure 3.15: Timing analysis of Sparkle-NC

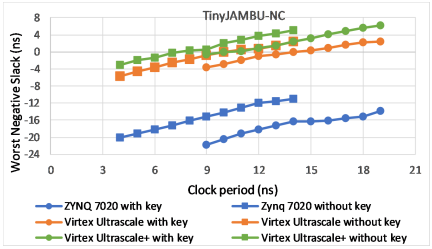


Figure 3.16: Timing analysis of TinyJAMBU-NC

Table 3.5: Maximum operating frequency of hash implementations

Hash Function	Max. Frequency (without key)	Max. Frequency (with key)
SPECK-NC	171.6 MHz	144.8 MHz
Pyjamask-NC	279.4 MHz	271.6 MHz
GIFT-NC	369.7 MHz	365.8 MHz
AES-NC	255.9 MHz	214.8 MHz
SKINNY-NC	202.4 MHz	202.5 MHz
PHOTON-NC	384.6 MHz	NA
Sparkle-NC	133.3 MHz	NA
TinyJAMBU-NC	138.9 MHz	104.2 MHz

NA-Not applicable

given by Diehl et al. [95] are 119 Mbps and 1622 Mbps, respectively, which is 206× and 10× lower than our reduced-round implementations. Similarly, the high-throughput implementation of SKINNY-128 on FPGA [65] has a Throughput/LUT of 3.06 Mbps, which is three times lower than our reduced-round implementation. To have a fair comparison, we have also implemented FNV-1a and Murmur3 hash on FPGA and compared them with the reduced-

Table 3.6: Resource utilization

Hash function	LUTs (without key)	Flip Flops (without key)	LUTs (with key)	Flip Flops (with key)
SPECK-NC	432	192	1273	384
Pyjamask-NC	811	448	1615	448
GIFT-NC	546	512	665	512
AES-NC	2225	512	3402	512
SKINNY-NC	2176	512	2348	512
PHOTON-NC	1338	436	NA	NA
Sparkle-NC	811	388	NA	NA
TinyJAMBU-NC	1521	387	1919	515

NA-Not applicable

round ciphers. The comparison of some of the fast non-cryptographic hash functions with our implementations is shown in Table 3.7.

Table 3.7: Comparison of maximum frequencies, throughput (Tp), throughput per LUT (Tp/LUT), and delay with related work

Design	Block Size (In/Out)	Maximum Frequency	Tp (Mbps)	Tp/LUT (Mbps)	Latency (ns)	Device
Murmur3	96/64	120.6 MHz	2573	4.54	24.87	VU+
FNV-1a	96/128	122.9 MHz	925	1.63	130.08	VU+
SipHash [121]	96/16	182.8 MHz	1463	1.38	21.88	Virtex 7
XORHash [121]	96/16	**627.3 MHz**	2868	9.86	11.13	Virtex 7
NSGAHash7 [121]	96/16	184.1 MHz	1473	18.41	21.72	Virtex 7
Xoodoo-NC (ours)	96/192	363.6 MHz	34'906	**112.96**	2.75	VU+
SPECK-NC (ours)	96/96	171.6 MHz	16'473	38.13	5.82	VU+
Pyjamask-NC (ours)	96/96	279.4 MHz	26,822	32.70	3.58	VU+
GIFT-NC (ours)	96/128	369.7 MHz	35'491	65.00	2.70	VU+
AES-NC (ours)	96/128	255.9 MHz	24'566	11.04	3.91	VU+
SKINNY-NC (ours)	96/128	202.4 MHz	19'440	8.93	4.93	VU+
PHOTON-NC (ours)	96/144	384.6 MHz	**36'922**	27.59	**2.6**	VU+
Sparkle-NC (ours)	96/128	133.3 MHz	12'797	15.78	7.5	VU+
TinyJAMBU-NC (ours)	96/96	138.9 MHz	13'334	8.77	7.2	VU+

VU+ - Virtex Ultrascale+

As presented in Table 3.7, our implementations of non-cryptographic hash functions demonstrate significantly higher throughput compared to previously existing ones. Although the available implementations of SipHash, XORHash, and NSGAHash7 are performed on a comparatively low-end FPGA compared to our implementations, the difference in latency and throughput are quite significant. XORHash [121] shows the highest operating frequency of all related works. However, the throughput compared to Sparkle-NC is still 4.5× lower, and Sparkle-NC has the lowest throughput among our reduced round implementations. On the other hand, PHOTON-NC showcases the highest throughput, which is 12.9× greater than XORHash and slightly surpasses the throughputs of GIFT-NC and Xoodoo-NC. The number of rounds required for

Xodoo-NC is four to produce a 192-bit output, and four rounds are sufficient to satisfy the avalanche properties. Most of the operations in PHOTON-NC are 4-bit operations which prove to be advantageous for PHOTON-NC on hardware. Besides the throughput per LUT, which is already provided in Table 3.7, we introduce another metric that considers both the latency and FPGA resources. This metric, calculated as $1/(latency * LUTs)$, is depicted in Figure 3.17 together with the $throughput/LUT$. Higher values of these two parameters indicate better performance of the hash function.

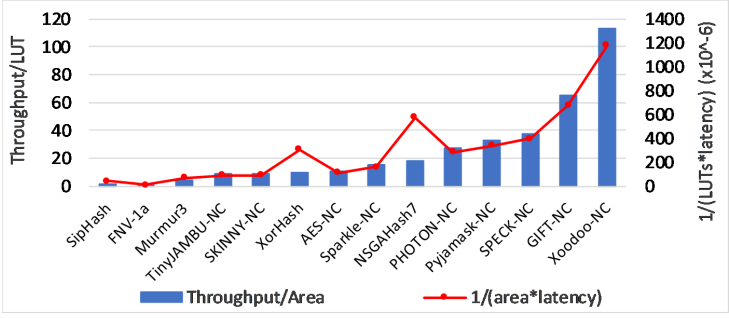


Figure 3.17: Throughput and latency with respect to resource utilization

Xoodoo-NC exhibits the best throughput per LUT, which is 1.7× higher than GIFT-NC and 4× higher than PHOTON-NC. Hence for applications where area efficiency plays an important role, Xoodoo-NC should be preferred over GIFT-NC or PHOTON-NC. Another observation when evaluating throughput per area is that Murmur3, FNV-1a, and NSGAHash7 employ 29, 38, and 3 DSP blocks, respectively, in their implementations, whereas our reduced-round ciphers do not use any DSP slices. PHOTON-NC has the lowest execution delay among all the implementations. The area-latency performance of hash functions shows similar characteristics as throughput-area for all hash functions except for NSGAHash7 and XorHash. Both NSGAHash7 and XorHash show increased area-latency performance, thanks to the relatively lower area-to-latency ratio, while area-latency performance of all other hash functions closely aligns with their throughput-area characteristics. The other versions of NSGAHash deliver much higher operating frequency and superior area-latency performance. However, NSGAHash does not satisfy the avalanche criteria as mentioned in Sect. 3.2.

Results and analysis on ASIC

The algorithms' ASIC performance was assessed through synthesis to evaluate their effectiveness for the 65nm TSMC manufacturing node. The typical-condition timing constraints of nominal voltage and 25 degree Celsius are used. The synthesis tool used is Genus 20.10 from Cadence. The first synthesis has been done to find the maximum frequency, and timing and area are reported. Then, synthesis was done at 100 MHz to find the minimum area. It is to be noted that the synthesis tool will always try to optimize for the area. Consequently, an algorithm synthesized for 100 MHz will have a smaller size compared to the one synthesized for 500 MHz. The synthesis tool has a number of standard cells available, some of which are smaller with lower drive strength, while others are larger with higher drive strength.

All algorithms are unrolled, and all rounds are executed in a single clock cycle. This way, the algorithm with the highest maximal frequency also has the lowest delay. For the implementations in this work that have an optional key, synthesis has been done on both versions, with and without the key. The algorithms Xoodoo-NC, PHOTON-NC, and Sparkle-NC do not use a key.

Figure 3.18 shows the maximum frequency of the algorithm implementation, and Figure 3.19 depicts the required area to achieve this frequency. Hash functions like GIFT-NC, SKINNY-NC, and Pyjamask-NC display minimal differences in hardware logic for keyed and non-keyed versions, resulting in only a slight disparity in operating frequencies. As the synthesis tool optimizes the design for area, there could arise minor variations in operating frequency compared to the FPGA implementations. Consequently, the GIFT-NC and SKINNY-NC implementations without key exhibit slightly lower operating frequencies compared to their keyed counterparts. It is evident that among the tested algorithms, Xoodoo-NC exhibits the highest frequency and the lowest area, establishing it as the optimal choice.

Figure 3.20 shows the area results after synthesis for the TSMC 65nm technology. The figure illustrates that AES-NC represents the largest implementation among the evaluated designs, while Xoodoo-NC is the smallest. Additionally, GIFT, Pyjamask, and PHOTON exhibit promising characteristics on ASIC, as these algorithms offer high operating frequencies and relatively low area requirements.

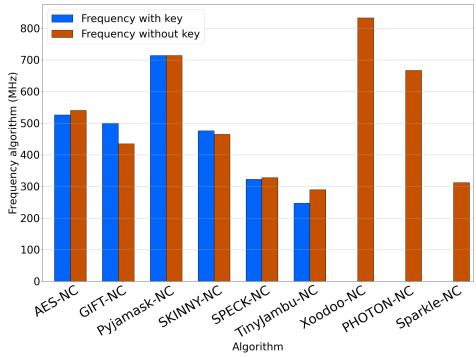


Figure 3.18: Maximum frequency of the ASIC implementations

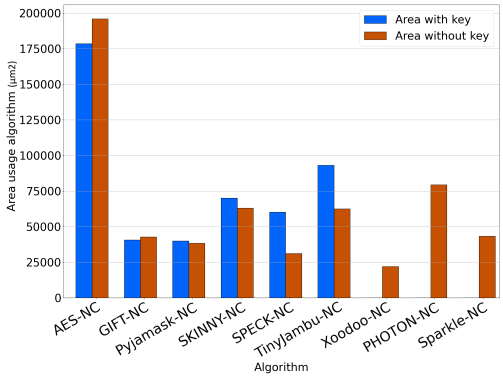


Figure 3.19: ASIC area at maximum frequency

3.5 Conclusion

This chapter investigates the use of reduced-round versions of existing lightweight ciphers and cryptographic permutations to serve as non-cryptographic hash functions, specifically for networking applications. In order to do so, we simulate the avalanche properties of each cipher as a function of the number of rounds. Based on the minimum number of rounds required to satisfy the avalanche requirements, we evaluate the resource utilization and latency on an FPGA and ASIC, assuming completely unrolled and non-pipelined architectures. We required a hash function capable of generating output with a single clock cycle latency and maintaining a high operating frequency. Implementing loop-unrolling and a non-pipelined approach helped in achieving the desired single-

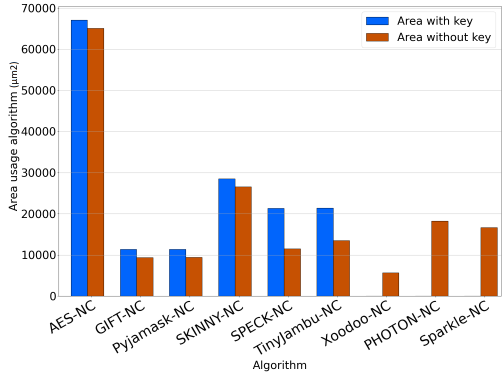


Figure 3.20: Area of the ASIC implementations at 100 MHz

clock cycle latency. Additionally, reducing the number of rounds contributed to achieving low logical depth and a high operating frequency.

We show that the newly proposed Xoodoo-NC and PHOTON-NC outperform all previously proposed hash functions in terms of throughput and latency on FPGA. Xoodoo-NC is also the most efficient algorithm on ASIC in all aspects. The other non-cryptographic hash functions we proposed are also valuable, allowing a pool of different non-cryptographic hash functions with desirable throughput and latency. This is important for applications needing multiple different non-cryptographic hash functions in high-speed lookup or counting architectures. This work also encourages the exploration of additional research methodologies for developing high-speed non-cryptographic hash functions. Our recent collaborative work, which leveraged evolutionary computation in the development of non-cryptographic hash functions [128], yielded promising outcomes and offers valuable insights into future research.

Chapter 4

Fast lookup architectures on FPGA and the analysis of their hardware-compatibility

This chapter discusses the design and implementation of fast membership query/lookup architectures for network security applications, especially large flow detection, on FPGA. This chapter centers around implementing hardware-efficient data structures for probabilistic/approximate membership queries. This chapter is organized as follows. Section 4.1 introduces the topic, followed by the design of optimized Bloom filter architectures in Section 4.2. A detailed analysis of the hardware friendliness of various probabilistic architectures is provided in Section 4.3. Section 4.4 concludes this chapter.

CONTENT SOURCE

- Sateesan, A., Vliegen, J., Daemen, J., and Mentens, N., Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. Microprocessors and Microsystems, 2022, 93, p.104619.(Contributions: Main author)
- Sateesan, A., Vliegen, J. and Mentens, N., "An analysis of the hardware-friendliness of AMQ data structures for network security". In Security, Privacy, and Applied Cryptography Engineering: 12th International Conference, SPACE 2022, pp. 287-313. (Contributions: Main author)

4.1 Introduction

High data rates in networks induce hefty storage and computation costs. This issue becomes even more important in networking applications where data need to be processed at line rate, i.e., the rate at which data are transmitted in the network. Storing large amounts of data within a limited on-chip memory of FPGAs and answering queries in a short time play an important role in high-speed processing applications like network security. This serves as the driving force behind our research, which focuses on designing and implementing high-speed probabilistic lookup/membership query architectures on FPGA.

Exact versus approximate membership query data structures: Membership queries or dictionary lookups have always been a challenging problem in networking applications. This is even more so when the number of unique elements to be stored, commonly referred to as the *cardinality*, is very large. Network applications such as packet forwarding, traffic flow monitoring and security, and deep packet inspection require efficient lookup algorithms to store and analyze network flows. Both exact and approximate membership query data structures are commonly employed for lookup applications. Exact membership query (EMQ) data structures store the complete input data/key. In contrast, approximate membership query (AMQ) data structures either do not store the key or store only a fingerprint of the key. Associative arrays, such as CAM, are the most commonly used exact membership query data structures. Bloom filters, Cuckoo filters, hash tables (in which false positives can be eliminated through chaining mechanisms), and Cuckoo hash tables are examples of AMQ data structures. Most existing AMQ data structures are improved/optimized versions of Cuckoo filters or Bloom filters. The input data or the key is a flow identifier (flow ID) in network security applications. In this work, we take the size of the flow ID as 96 bits.

EMQ data structures have no false positives or false negatives. In contrast, AMQ data structures exhibit a low false positive rate. Nevertheless, EMQ data structures perform worse than AMQ data structures when it comes to chip area, energy consumption, and operating speed, as presented by George Varghese in [115]. Since these are critical implementation properties for network routers [71], AMQ data structures are the preferred lookup mechanism when the address of the element is not needed, and a limited number of false positives are acceptable.

AMQ data structures in networking applications: In a typical application, as shown in Figure 4.1, AMQ data structures provide an estimate of whether an element (x), also called a key, is present in the data structure (S). In such a system, probabilistic lookups can be beneficial in speeding up the off-chip

memory searches, as off-chip memory access occurs only when the AMQ data structure yields a positive output. AMQ data structures can be employed as a standalone unit when the sole requirement is to ascertain the presence of the key x. However, in cases where both the presence and value of a key need to be retrieved, AMQ data structures can be used to implement key-value storage mechanisms.

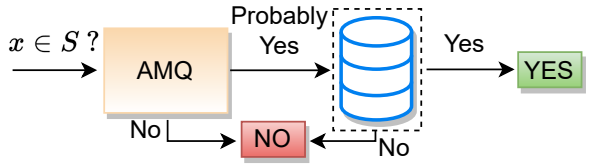


Figure 4.1: Role of an AMQ data structure for lookup applications

AMQ data structures are widely used for lookups in networking and database applications with strict speed requirements, limited memory, and power constraints [224]. In large flow detection systems, as discussed in Section 2.5 of Chapter 2, the lookup (AMQ) module inspects if the parsed flow identifier or flow ID (f) of the incoming network flow is present in the blacklist or not. A simple representation of such a system is shown in Figure 4.2. The incoming flow f is discarded if it is present in the blacklist. If f is not present, a detection module employing techniques like pattern matching, probabilistic algorithms, or machine learning, examines the flow. In this context, it is crucial to consider the false positive rate (FPR) of the AMQ lookup structure, as false positives can incorrectly classify legitimate flows as malicious. Therefore, the FPR of the data structure becomes equally important as speed. An example of a DDoS detection system relying on a setup similar to the one depicted in Figure 4.2 is proposed by Scherrer et al. [217].

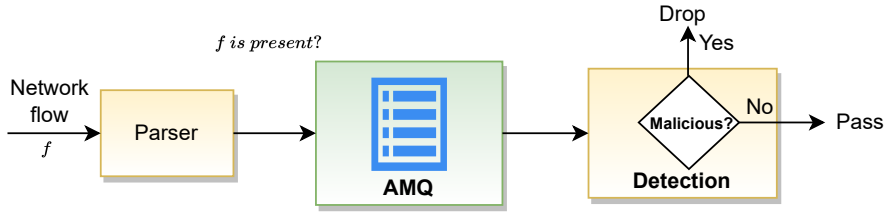


Figure 4.2: Role of AMQ data structures for lookups in network attack detection systems

AMQ data structures exhibit optimal performance when the data set is static, and the cardinality is known upfront. A data set is said to be static when

the data remains unchanged after the data collection. Our analysis primarily focuses on single data sets where cardinality is already known or predictable, which resonates with the characteristics of network security algorithms that rely on a specified measurement epoch. However, it does not mean that the data cannot be added to data structures at runtime. New incoming data can be added to the data set and data structure, but the existing data cannot be changed once the data collection is complete. Data structures based on this criterion are the prime requirement for large flow detection mechanisms, where measuring the network flow is stipulated to a specific measurement epoch. In network applications, the flow ID is the key for the lookup, and the size of the flow ID is fixed. In this work, the flow ID is taken as a combination of source and destination IP addresses and ports, totaling 96 bits. We ignore the protocol field as the protocol is already being used in filtering the network flows in our large flow detection architecture. This implies that the input size/key size to the lookup or key-value storage mechanism is 96 bits.

Challenges in the implementation of membership query data structures

Membership query or lookup is one of the primary concerns when it comes to hardware platforms. On software platforms, a dictionary is the easiest solution for storing data as key-value pairs. However, a data structure similar to dictionaries would be highly inefficient on hardware. CAMs are the most effective solution for lookup data structures without sacrificing accuracy, but they are too expensive for resource-constrained devices [212]. Additionally, the operating frequency of a CAM decreases with increasing size, posing a challenge when large amounts of data need to be processed at line rate.

Even though probabilistic architectures come with several advantages, such as lower memory requirements and faster data retrieval, there are concerns regarding accuracy and hardware-friendliness. When it comes to processing data at line rate on hardware, a number of challenges have to be addressed to optimize the accuracy, lookup speed, and memory utilization. Data structures, such as hash tables employed with linear probing or chaining as collision-resistant mechanisms, would be difficult to implement on hardware. The difficulty arises from the changing table size and the limited feasibility of pipelining. Dynamic insertion is another requirement for network security applications, where a full set of flow IDs are not available at the time of construction, and new flow IDs need to be added during runtime. Therefore, static architectures are generally not preferred unless all the required keys or rules to be stored are already available, as with regular expression matching. It is to be noted that static architectures do not correspond to static data sets. A static architecture requires the whole data set during the construction, and data cannot be added

at a later time. In our context, a static data set allows data to be added until the data collection is complete or the end of the measurement period, but the collected data cannot be modified at a later time. For applications such as large flow detection, the architecture should be dynamic, and the data set should be static. Bloom filter is one such architecture and is very promising for fast lookup applications due to its inherent collision resistance, higher lookup speed, low memory requirement, and hardware friendliness. In the following section, we focus on optimizing the Bloom filter for fast lookup in hardware platforms.

4.2 Optimized Bloom filter architectures for lookups

The positive attributes of Bloom filters for hardware implementations motivate us to design optimized Bloom filter architectures for network security applications. Although Bloom filters possess a native hardware-friendliness, there still exists bottlenecks that can adversely affect the throughput, such as the complexity of hash functions and memory access delays. In this chapter, we address these bottlenecks to improve the performance of Bloom filters for network security applications, especially for large flow detection. Since the speed of a Bloom filter heavily depends on the speed of the hash functions that process the incoming data, we first concentrated on the design and implementation of high-speed, hardware-friendly hash functions as discussed in Chapter 3. We use the proposed Xoodoo-NC in the Bloom-1 architecture, proposed by Qiao et al. in [199], and our proposed Parallel Bloom Filter architecture inspired by the design of Dharmapurikar et al. in [93]. We show that our Bloom filter implementations based on Xoodoo-NC are more efficient with respect to occupied FPGA resources and computational delay in comparison to Bloom filters based on the existing FNV-1a [112] hash function and CAM-based lookup architectures.

4.2.1 Bloom filter

The Bloom filter is an approximate membership query data structure used to determine whether an element is in a set of elements. The filter has two possible outcomes: 1) the element might be in the set, or 2) the element is not in the set. Although the first option can trigger false positives, the second option rules out false negatives. In 1970, Howard Bloom presented space/time trade-offs for lookup algorithms in data structures based on hash coding with allowable errors [71]. The author's name has been linked to the presented algorithms ever since.

A standard Bloom filter (SBF) that searches for an input x in a set S, consists of a bit vector of length m, which is a compacted representation of the elements that are present in S, and k hash functions, which are used to map n inputs to the bit vector. An example is shown in Figure 4.3, where m is 14 and k is 3. Before the SBF can be used, the bit vector is first loaded with zeroes. Then, all k hash functions are applied to each element s in the set S, i.e. $H_{k_i}(s)$ is calculated, with $i \in 1, \ldots, k$. For each hash function, the resulting hash value is used as an index in the bit vector, and the corresponding bit is set to 1. This process is repeated for each hash function and subsequently for each possible $s \in S$. In Figure 4.3, this means that 3 bits out of 14 are set to 1 for each loaded element s. In the figure, this process is illustrated for two elements s_1 and s_2. After these steps, the set S has been successfully loaded into the Bloom filter.

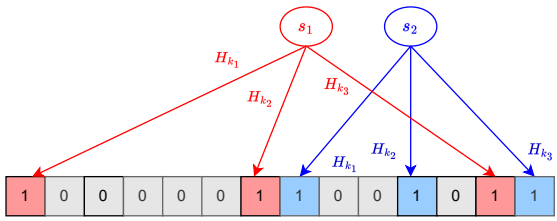


Figure 4.3: Standard Bloom filter (SBF) representation

To query the SBF using input x, all hash digests $H_{k_i}(x)$ on x are calculated. Analogous to the initialization phase, the hash values are used as indices in the bit vector. If all the corresponding bits in the bit vector are set to 1, the SBF returns $x \in S$. Otherwise, $x \notin S$ is returned. As stated above, a membership query can have a false positive result. This happens when, during the initialization of the Bloom filter, the hash values of different elements in the set lead to a 1 being written to the same position in the bit vector. The false positive rate of a standard Bloom filter is presented in equation 4.1 as presented by Mitzenmacher and Upfal [182]. Unlike CAMs, Bloom filters do not explicitly store each element in a memory cell but rather store the bit vector that represents all elements that are present in the data structure. This leads to more economical memory resource utilization.

$$fpr = (1 - e^{-kn/m})^k \tag{4.1}$$

From the analysis of existing probabilistic/approximate data structures, it can be inferred that the hardware-friendliness of the Bloom filter is better than the competing data structures. The execution speed of Bloom filters is one of the bottlenecks for applications such as large flow detection. Bloom filter consists

of k hash functions, and the overall update/query delay is proportional to the execution delay of the hash functions. To reduce the false positive rate, it is required to increase the memory size m. The factors that determine the number of hash functions (k) required are the size of the memory (m), the number of elements to be added (n), and the desired false positive rate (ϵ). The optimal value of k to minimize the false positive rate depends on the m/n ratio, and k can be computed as $k = \frac{m}{n} ln2$. Therefore, it can be deduced that an increase in memory will lead to a higher number of hash functions, or vice versa, to achieve the minimum possible false positive rate. Moreover, as the value of k increases, the number of memory accesses also increases, adding to the total delay. Hence, it is required to reduce the complexities in hash computation and memory access delays. In this subsection, we seek an optimized solution to solve the above-mentioned drawbacks of Bloom filters by exploiting the parallelism of FPGAs.

On a general note, if achieving the minimum false positive rate is not necessary, we can reduce the number of hash functions to meet the desired false positive rate, thereby minimizing the required hash bits and computational complexity. The minimum value of k to attain the desired false positive rate ϵ can be calculated as $k = -\frac{ln(\epsilon)}{ln(2)}$. A plot of false positive rate against varying values of k is depicted in Figure 4.4. It should also be noted from the plot that as the value of k goes beyond the optimal value, there is another rise in the false positive rate. Additionally, the false positive rate also depends on the n/m ratio, as deduced from equation 4.1. Increasing the value of n while keeping m constant will increase the false positive rate, explaining the correlation between the increment in false positives and the filter approaching full capacity.

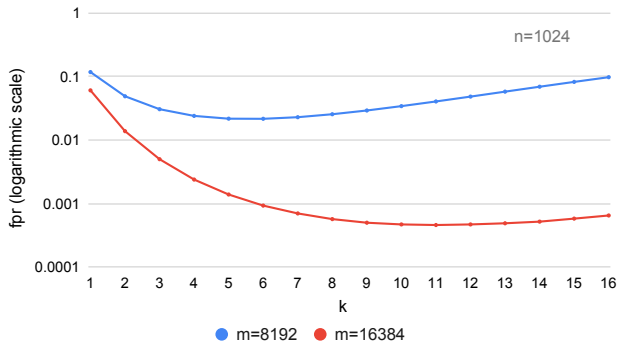


Figure 4.4: The false positive rate of Bloom filter against varying number of hash functions

4.2.2 Related work

A wide range of related research is available, elaborating on the design and optimization of Bloom filters for diverse applications. When designing hardware architectures for Bloom filters, multiple factors must be taken into account, including the lookup delay, false positive rate, and operating frequency.

As one of the main factors affecting the lookup delay of a Bloom filter is the execution speed of the hash functions, a number of fast non-cryptographic hash functions with good avalanche properties have been proposed, such as Murmur3 and FNV-1a. Murmur3 is the result of a general study of non-cryptographic hash functions by Estebanez et al. [105]. FNV-1a was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee by Fowler and Vo and later improved by Landon Curt Noll [112] to reduce the computational delay. However, as discussed in Section 3.2, commonly employed hash functions like FNV-1a and Murmur3 are inefficient for Bloom filters due to small input block sizes and high computational complexity. As our main objective is to employ a single hash function that can produce a hash output in a single clock cycle for implementing Bloom filters, we proposed Xoodoo-NC, a fast hardware-oriented non-cryptographic hash function with good avalanche properties, which outperforms all other solutions in terms of latency in hardware. The design and implementation of Xoodoo-NC are discussed in detail in Chapter 3.

Besides optimizing the hash functions, there are other methods that contribute to the fast generation of the lookup addresses in Bloom filters. Examples are the one-hashing Bloom filter (OHBF) [168], the less hashing Bloom filter (LHBF) [149], and the ultra-fast Bloom filter (UFBF) [167]. OHBF and UFBF follow similar techniques, using a single base hash function and k modulo operations to obtain k addresses. LHBF employs two base hash functions and a linear equation to generate additional addresses from the base hash functions. Although these techniques can reduce the generation delay of the lookup addresses, the overall delay of a query will still be dominated by the delay of the memory accesses when an SBF array [71] is used as depicted in Figure 4.3. This is because each read/write operation will require one clock cycle when the on-chip RAM in an FPGA is used for the storage of the Bloom filter. Hence, it is important to not only reduce the delay for the generation of the lookup addresses, but to also consider Bloom filter architectures that reduce the number of memory accesses.

In an SBF, the number of memory accesses is directly proportional to the number of hash functions (k). In order to reduce the false positive rate, the number of hash functions and the size of the data structure (m) must be increased. However, this has a negative impact on the execution speed. Parallel Bloom filter (PBF),

as proposed by Dharmapurikar et al. [94], is a classical solution to overcome this issue. There are other approaches like multi-partitioning counting Bloom filters (MPCBF) [106], OMASS [181] and One-memory-access Bloom filters (Bloom-1) [199], where it takes only one memory access per query. MPCBF, OMASS, and Bloom-1 employ similar approaches based on partitioning memory blocks to limit the number of memory accesses to one. In this work, we do not focus on counting Bloom filters which store a count value for each memory cell to enable the deletion of elements, which leads to increased memory usage. Consequently, MPCBF is not considered for our implementation. In OMASS, each element is represented in multiple sets with different hash mappings, reducing the false positive rate at the expense of additional memory usage. Among the considered algorithms, we choose Bloom-1 [199] for implementation as it requires less memory storage than OMASS.

Several studies have explored the utilization of Bloom filters in hardware. Dharmapurikar et al. [93] present an implementation of parallel Bloom filters on FPGA for string matching. The implementation is fast in terms of the number of memory accesses with acceptable resource utilization. However, it has a very low operating frequency of 73.5 MHz, making it unsuitable for high-speed networks. In this work, an improved version of PBF is implemented, which offers a high operating frequency. In [232], a rolling hash function-based Bloom filter is presented and implemented on a high-end FPGA for fast streaming data. However, the architecture assumes a byte-based interface, restricting its compatibility to Gigabit Ethernet networks. When moving towards Terabit Ethernet, the incoming data need to be processed in larger parallel blocks. The work of Kaya et al. [145] and of Lyons et al. [172] focus mainly on the power reduction of Bloom filter implementations. A counting Bloom filter implementation proposed by [126] for network intrusion detection seems to provide operating frequencies above 200 MHz for an input data size of up to 8 bytes. Despite the notable achievements presented in these publications, Terabit Ethernet networks demand an even higher degree of parallelization and/or operating frequency.

In this work, we propose to use the existing Bloom-1 architecture and the modified parallel Bloom filter architecture, both of which require only one memory access per query. We also propose to use Xoodoo-NC, an extremely fast non-cryptographic hash function that outputs a hash value large enough to be split into chunks that can directly be used in the Bloom filter architecture. We implement speed-optimized hardware architectures that outperform all other FPGA-based Bloom filter implementations as well as the architectures based on the CAM IP cores in Xilinx FPGAs.

4.2.3 Bloom filter algorithms for analysis

This section presents the architectures of Bloom filter algorithms, Bloom-1 and parallel Bloom filter, chosen for implementation on hardware.

Bloom-1 architecture

The number of memory accesses in SBF is proportional to the number of hash functions k, as k locations in the bit vector need to be read out. In order to reduce the number of read-outs, a more efficient structure Bloom-1 is proposed by Qiao et al. [199]. Bloom-1 is a fast Bloom filter architecture requiring only one read-out for each query. Bloom-1 uses hash functions to map an input element s and requires a total of $log_2(l) + k \times log_2(w)$ hash bits. The first $log_2(l)$ bits are the output of the hash function $H_l(s)$ and are used as an address in a data structure that has l memory locations, as shown in Figure 4.5. We call each word in the data structure a membership word. The next k values of $log_2(w)$ bits each are the outputs of the hash functions $H_{k_i}(s)$, with $i \in 1, \ldots, k$. They are used as addresses that point to bit locations in the membership word. The size of a membership word is w, and a 1 is written at the bit locations that correspond to the addresses $H_{k_i}(s)$. To query an input x, a similar procedure is followed to read out k bits (of which the locations are determined by $H_{k_i}(x)$, with $i \in 1, \ldots, k$) from the membership word at the address determined by $H_l(x)$. If all k bits are 1, the Bloom-1 filter returns $x \in S$. Otherwise, $x \notin S$ is returned.

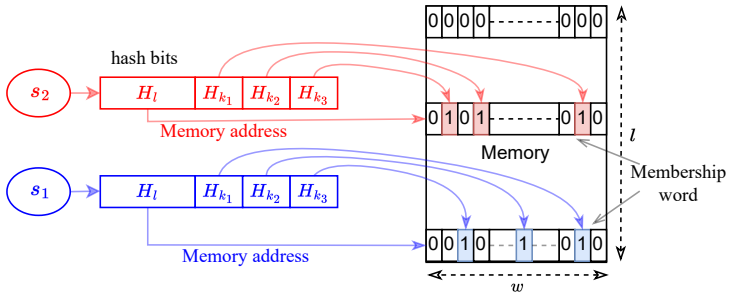


Figure 4.5: Bloom-1 representation

In Bloom-1, w usually corresponds to the data bus width in a computer system, i.e., $w = 32$ or 64. The size of the entire Bloom-1 filter is denoted as $m = l \times w$. The advantage of Bloom-1 is that the number of required hash bits is significantly lower when k is large, compared to other Bloom filters [199]. The total number

of hash bits for Bloom-1 is $log_2(l) + k \times log_2(w)$, compared to $k \times log_2(m)$ for SBF. Additionally, the w-bit membership word can be read out at once, while the k 1-bit values in SBF require k read-outs. This significantly increases the speed of Bloom-1 compared to SBF.

Qiao et al. give an approximation of the false positive rate (fpr) for Bloom-1 in [199]. A comment published by Reviriego et al. in [202] refines the approximation, resulting in Eq. 4.2. Besides the Bloom-1 parameters, l, w, and k, the number of entries, n, has an influence on the false positive rate.

$$fpr = \sum_{x=0}^{n} \left(\binom{n}{k} \cdot \left(\frac{1}{l}\right)^{x} \cdot \left(1 - \frac{1}{l}\right)^{n-x} \cdot \left(\frac{w!}{w^{k(x+1)}} \sum_{i=1}^{w} \sum_{j=1}^{i} (-1)^{i-j} \frac{j^{kx} i^{k}}{(w-i)!j!(i-j)!} \right) \right) \tag{4.2}$$

Parallel Bloom filter (PBF) architecture

We propose a modified architecture named Parallel Bloom Filter to keep the overall latency of the Bloom filter to a single cycle. The parallel Bloom filter (PBF) implementation approach is inspired by the work by Dharmapurikar [93]. Our approach follows the SBF model but employs multiple smaller memory blocks to form SBFs in parallel to increase the lookup speed. When the number of hash functions k is set to 2 for each SBF, a Bi-SBF is formed, which requires two memory accesses per update/query. The Bi-SBF is termed a mini-Bloom filter in the original literature. The authors of [93] suggest that a dual-port BRAM can be employed in Bi-SBF to perform two read/write operations at once, hence keeping the latency at one cycle. However, to avoid write conflicts, assigning one port for reading and the other for writing is better, especially if the memory is being shared.

The Bi-SBFs can be arranged in parallel for two reasons, either to reduce the false positive rate or to increase the storage capacity. In order to reduce the false positive rate, the value of k and m has to be increased, keeping the value of n constant, which is achieved by stacking the Bi-SBF blocks in parallel. Such Bi-SBFs stacked together in parallel form a PBF, as shown in Figure 4.6 (a). To form a parallel Bloom filter with k hash functions, $k/2$ SBFs are stacked together. The authors of [93] indicate that the false positive rate of a Bi-SBF is $(\frac{1}{2})^2$, considering a constant n/m ratio of 1419/4096, based on Eq. 4.1. The overall false positive rate of the PBF is then $(fpr(\text{Bi-SBF}))^{k/2}$.

In order to increase the storage capacity, Dharmapurikar et al. [93] suggest stacking the PBFs together to form a Large Bloom Filter (LBF). The string storage capacity of an LBF formed by arranging p PBFs together while keeping

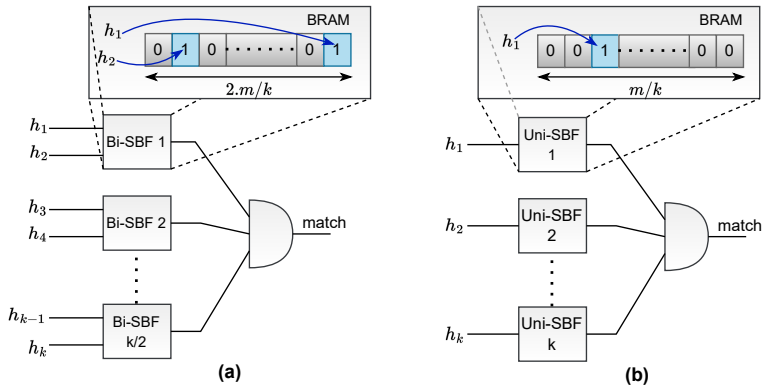


Figure 4.6: Parallel Bloom filter logic (a) using Bi-SBF and (b) using Uni-SBF

the same false positive rate is $n \times p$, where n is the string storage capacity of a single PBF. If the overall memory size of a PBF is m, then p such filters in parallel constitute a Bloom filter of memory size $p \times m$. When an element is stored, it is placed in one of the PBFs, and the selection of a specific PBF is determined using a hash value. It is important to note that this approach introduces additional operational logic, which has an adverse impact on resource requirements and speed. Hence, we prefer to change the memory size of the Bi-SBFs accordingly for the required false positive rate and storage capacity. To increase the storage capacity, m is increased while keeping the same number of SBFs in parallel. The query/configure operations are similar to standard Bloom filters, with every block operating in parallel. An element e is said to be present in the set only if all the parallel filters show a match. In our implementation, the false positive rate of the Bi-SBF is determined by Eq. 4.1, and the n/m ratio changes depending on the false positive rate or string storage requirements. The false positive rate of a Bi-SBF is $(1 - e^{-kn/m})^2$ and the equation $(fpr(\text{Bi-SBF}))^{k/2}$ still holds for the false positive rate of PBF. The total memory size of each Bi-SBF is $2 \times m/k$, where m is the total memory size of the Bloom filter.

To make the PBFs faster, the number of hash functions for each SBF can be set to 1, resulting in a Uni-SBF. The memory size of each Uni-SBF is m/k. The false positive rate of each such Uni-SBF adheres to Eq. 4.1 and is $(1 - e^{-kn/m})$. The overall false positive rate of the PBF is given by $(fpr(\text{Uni-SBF}))^k$. The PBF based on the Uni-SBF architecture shown in Figure 4.6 (b) has the same false positive rate as PBF using Bi-SBF but requires fewer hash bits. The PBF based on Uni-SBF can employ a dual port BRAM with dedicated ports for read and write operations, which can reduce the lookup latency to only one cycle

with all SBFs operating in parallel. It should be noted that there is always a trade-off, and we have the option to trade logic or memory bits/ports to reduce the false positive rate. By capitalizing on this trade-off, we can guarantee that the design is tailored for a specific FPGA platform that we target.

4.2.4 Experimental setting and basis of comparison

Our analysis focuses on Terabit Ethernet. FPGA platforms that support Terabit Ethernet are the 200Gbps NFB-200G2QL board [186] and Xilinx Alveo U250 [24], which processes network packets at a line rate of 512 bits per clock cycle. It is evident that implementing network security applications on Terabit Ethernet FPGA platforms requires processing many data bits in parallel. An example of a network encryption implementation that adheres to the strong operating frequency and parallel processing requirements of Terabit FPGA platforms is given in [174].

The considered flow ID in this work contains the source and destination IP addresses and the source and the destination ports from the packet header. The size of the flow ID is 96 bits, which is the case in an IPv4 network. We compare six different lookup data structures in terms of cycle count, clock frequency, FPGA resources, and false positive rate:

- CAM block generated by the design tools of Xilinx,
- speed-optimized custom CAM consisting of manually combined smaller CAM blocks,
- Bloom filter based on the PBF architecture and the FNV-1a hash function,
- Bloom filter based on the PBF architecture and the newly proposed Xoodoo-NC hash function.
- Bloom filter based on the Bloom-1 architecture and the FNV-1a hash function,
- Bloom filter based on the Bloom-1 architecture and the newly proposed Xoodoo-NC hash function.

Each of the six architectures has a storage capacity of 1024 data words, with each word consisting of 96 bits. For the first two architectures, which are CAM-based, this means that exactly 1024 x 96 memory cells are occupied. For the last four architectures, which are based on Bloom filters, we explore different options for the Bloom filter size, false positive rate, and the number and output size of the hash functions. The number of entries stored in the

Bloom filters is equal to 1024 to ensure a fair comparison with the CAM-based implementations.

The CAM architecture is directly generated by the design tools of Xilinx. It utilizes embedded RAM (Block RAM or BRAM) with a size of 1024 x 96 bits, in combination with a limited amount of FPGA logic in the form of lookup tables (LUT) and flip-flops (FF). A lookup in the directly generated CAM takes two clock cycles. The second CAM architecture consists of 128 CAM blocks of size 24 x 32 bits. The Xilinx design tools again generate these smaller CAM blocks, and the LUTs and FFs for combining and addressing these blocks are added manually. This leads to an architecture with a higher operating frequency but with a larger resource occupation compared to the directly generated CAM architecture. A lookup in the manually modified CAM takes three clock cycles. Despite requiring an additional clock cycle, the high operating frequency of modified CAM makes it suitable for high-speed applications. Figure 4.7 illustrates the operating frequency and resource utilization of the CAM-based architectures implemented as a reference for comparison. While the modified CAM architecture may exhibit higher latency in smaller CAMs, the situation changes as the CAM size increases. The maximum operating frequency of the directly generated CAM decreases drastically for larger CAMs, leading to a much higher latency in terms of nanoseconds when compared to modified CAM architecture, as discussed in Section 4.2.6.

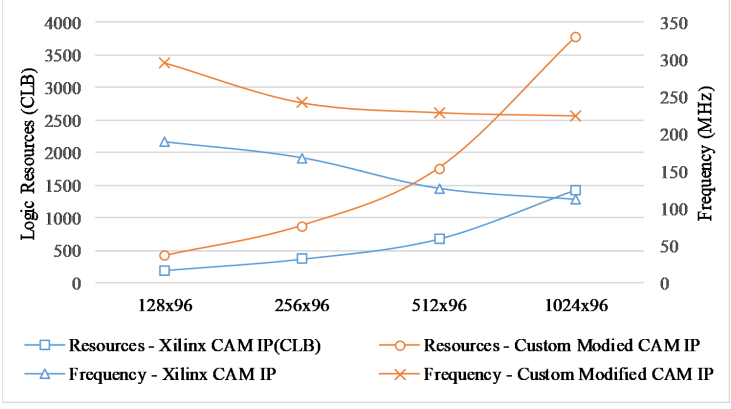


Figure 4.7: Maximum operating frequency and resource utilization of the CAM directly implemented with the Xilinx tools (referred to as Xilinx CAM IP) and the speed-optimized CAM architecture (Custom Modified CAM IP)

For all experiments, the Vivado 2017.4 design tool of Xilinx is used, and the UltraScale+ XCVU7P-FLVC2104-1-E FPGA is targeted. Only for the

generation of the CAM blocks the ISE 14.7 design tool of Xilinx is used because CAM generation is no longer supported in Vivado. Subsequently, the generated code from ISE is imported into Vivado to be used in the UltraScale+ FPGA, which is not supported by ISE 14.7.

4.2.5 Hardware architectures

The hardware implementations of the Bloom filter architectures that are evaluated in this work are discussed in this section. Our approach involves the utilization of a single non-cryptographic hash function, which produces an output split into $k + 1$ parts, i.e., one part of size $log_2(l)$ and k parts of size $log_2(w)$, as explained in Section 4.2.3. Two non-cryptographic hash functions with an input size of 96 bits are evaluated, namely FNV-1a and Xoodoo-NC. In the following paragraphs, we elaborate on the implementation details of the hash functions and the Bloom filter implementations.

Hash functions

FNV-1a FNV-1a can provide output hash sizes ranging from 32 to 1024 bits. It uses only two parameters - a non-zero FNV Offset basis, and an FNV Prime. Both parameters depend on the hash output size. FNV-1a processes one byte at a time, and the algorithm consists of iterations of two sequential operations - XOR and multiplication.

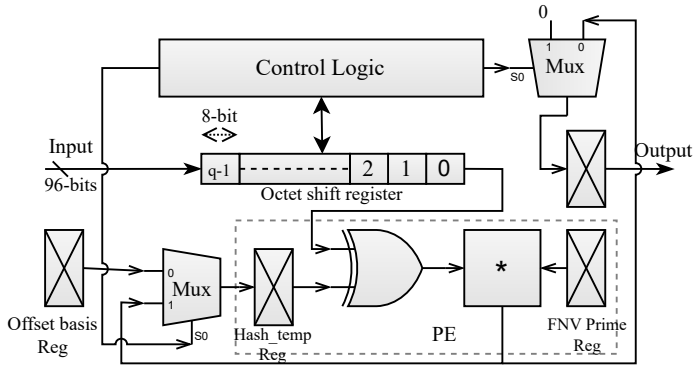


Figure 4.8: FNV-1a - serialized hardware architecture block diagram

The hardware block diagram of FNV-1a is shown in Figure 4.8. The input can be of any size, but only 8 bits are processed per cycle. The input is stored

in an octet shift register, which shifts a single byte per cycle. The size of the octet shift register is equal to the size of the incoming message, namely q bytes. The size of the flow ID is 96 bits, corresponding to $q = 12$. The hash_temp register stores the intermediate hash value. Every clock cycle, a byte is passed to the processing element (PE), which performs the XOR operation either with the intermediate hash value or the FNV Offset basis if the passed byte is the first byte of the input. Subsequently, multiplication with the FNV Prime is calculated. The output is obtained after processing the final byte of the input.

The main limitation of a simple FNV-1a architecture is its ability to process only one byte per clock cycle. The input needs to be buffered until the processing is completed. This is a major concern for high-speed networks like Terabit Ethernet networks, where a complete packet can be received in each clock cycle. For IPv4 packets, the size of the flow ID is taken as 12 bytes in this work, and it needs to be processed every clock cycle. Another concern in the hardware implementation is related to the multiplication operation, which is a core operation in the processing element (PE). If the input size and the output hash size need to be increased, the width of the multiplier must also be increased. This, in turn, adds to the complexity of the hardware design and the critical path. Pipelining somewhat helps to mitigate this issue and to improve the overall throughput (after an initial delay equal to q clock cycles).

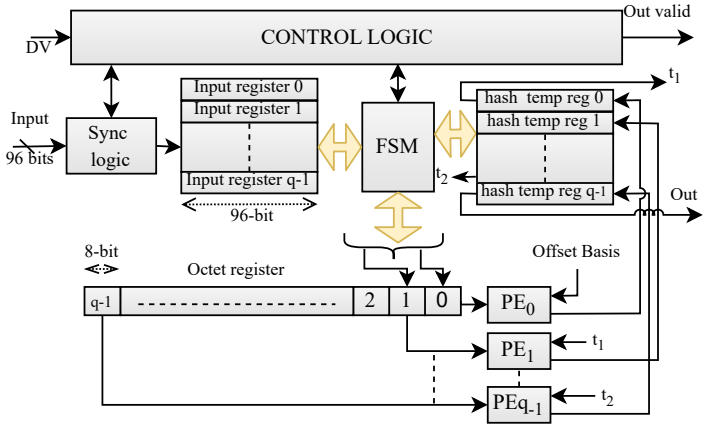


Figure 4.9: FNV-1a - Pipelined hardware architecture block diagram

The hardware architecture of the pipelined version is shown in Figure 4.9. The core of the design is a free-running Finite State Machine (FSM), which processes q bytes in each clock cycle. The input buffers store the incoming q-byte inputs. In total, q input buffers are implemented that each store a q-bit input word.

In each clock cycle, the bytes from the input buffers (starting from register 0) are shifted to the octet register. The octet register is an array of 8-bit registers. For every input, the first byte is stored in the 0th register, the second byte in the 1st register, and so on. Each PE takes two inputs, one byte from the octet register and the output from the previous PE, except for PE_0. PE_0 takes the FNV Offset basis, and the 0th byte of the octet register as inputs. The output of each PE is connected to an output buffer that stores the intermediate hash value and is connected to the input of the next PE in the sequence. The pipelined version of FNV-1a can process a 96-bit input in every clock cycle and has an output latency of 12 clock cycles.

Xoodoo-NC The architectural features and hardware implementation of Xoodoo-NC are discussed in detail in Section 3.3 of Chapter 3.

Bloom filters

Bloom-1: The hardware architecture of the Bloom-1 filter adheres to the principles outlined in Section 4.2.3 and is illustrated in Figure 4.10. Bloom-1 implementation employs only BRAMs for storage. The hash bits are generated with a single hash function, Xoodoo-NC. The output of Xoodoo-NC is split into $k+1$ parts for determining the location of the membership word and addressing the specific bits within the membership word. The memory is read or written accordingly depending on the *query/insert* instruction. The overall execution delay is determined by two factors: the BRAM read-write delay, which is one clock cycle, and the delay of the hash generation. Xoodoo-NC is executed in a single clock cycle, while the non-pipelined (serialized) and pipelined versions of FNV-1a take 14 and 2 clock cycles, respectively.

Parallel Bloom Filter PBFs are constructed by stacking Bi-SBFs or Uni-SBFs in parallel, as discussed in Section 4.2.3. The hardware architecture of PBF using Bi-SBFs is shown in Figure 4.11. Only on-chip BRAMs are employed to implement the memory. Dual port BRAMs are employed with dedicated ports for read and write operations. Xoodoo-NC is used to generate the hash values. The SBFs are operated in parallel such that the read/write latency is always equal to the latency of a single SBF. When using Bi-SBF, the latency is three cycles, while Uni-SBF offers a lower latency of two cycles. The memory sizes of Bi-SBF and Uni-SBF are $2m/k$ and m/k, respectively, where m is the overall size of the Bloom filter and k is the number of hash functions. Dividing the memory block into multiple smaller blocks in PBF helps to minimize the routing delays associated with larger BRAM blocks.

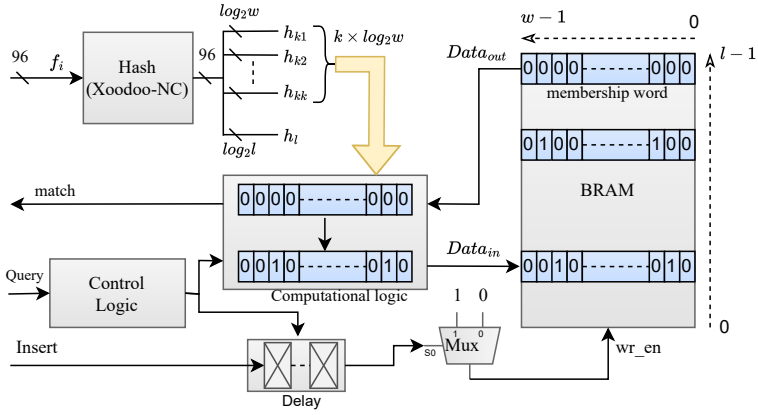


Figure 4.10: Bloom-1 hardware architecture

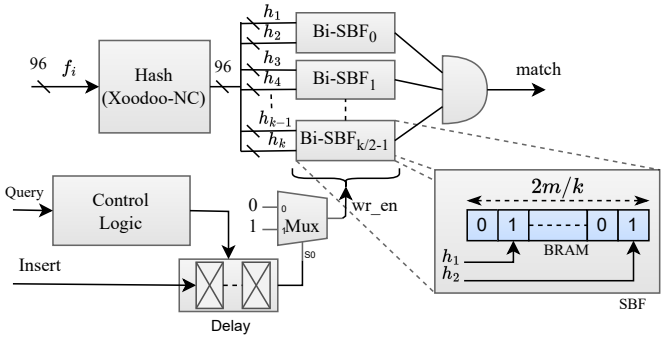


Figure 4.11: Parallel Bloom Filter hardware architecture

4.2.6 Hardware evaluation

In this section, we present the results of the experiments conducted on a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVC2104-1-E) using the Vivado 2017.4 design tool. For the Bloom filter implementations, we employ FNV-1a with 64-bit and 128-bit hash outputs, and Xoodoo-NC with 96-bit and 192-bit hash outputs [213]. The input size is always 96 bits. For FNV-1a, both the non-pipelined and the pipelined architectures are evaluated. The Bloom-1 parameters l, w, and $m = l * w$ are taken as 4096, 64, and 262144, respectively. For the PBF implementations, the values of m and k are varied to match the fpr of Bloom-1. The size of the CAMs that we compare with is 1024x96 bits,

where a depth of 1024 indicates the number of elements to be added, and the width of 96 bits represents the input size, which is the size of the flow ID.

Effect of hash functions on Bloom-1

The resource usage in terms of FPGA logic and DSP units in the Bloom-1 architecture while using different hash functions is depicted in Figure 4.12. The Xoodoo-NC-based implementations outperform the FNV-1a-based implementations, especially for larger values of k. The FNV-1a implementations utilize the multipliers inside the DSP units, whereas no DSP units are employed by the implementations based on Xoodoo-NC. As the value of k increases, a drastic increase in logic utilization can be seen in FNV-based Bloom filter implementations compared to Xoodoo-NC-based implementations. Additionally, the pipelined version of FNV-1a has a significantly higher consumption of logic resources and DSP units than the serial FNV-1a. Figure 4.13 demonstrates the operating frequency and execution delay of the Bloom-1 architectures. The Xoodoo-NC-based Bloom filter architectures exhibit a much higher operating frequency and considerably lower execution delay than the FNV-1a-based architectures.

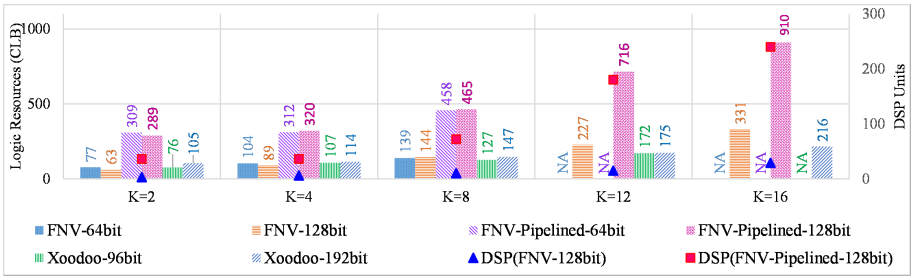


Figure 4.12: Resource Utilization of Bloom-1: Logic resources and DSP units

The outcomes obtained from the implementations of Bloom filters are presented in Table 4.1, and Figures 4.12 and 4.13. Two versions of the Bloom-1 architectures are considered in the table: those with $k = 2$ and $k = 12$. To calculate the false positive rate (fpr) using Eq. 4.2, the number of entries in the Bloom filter is assumed to be 1024, such that a fair comparison can be performed with the CAMs whose depth is taken as 1024. Going from $k = 2$ to $k = 12$ leads to an increased resource occupation but a reduced false positive rate for both Xoodoo-NC and FNV-1a-based Bloom filters. Nevertheless, the resource occupation of the Xoodoo-NC-based Bloom filter remains considerably lower than all the other considered architectures for all values of k.

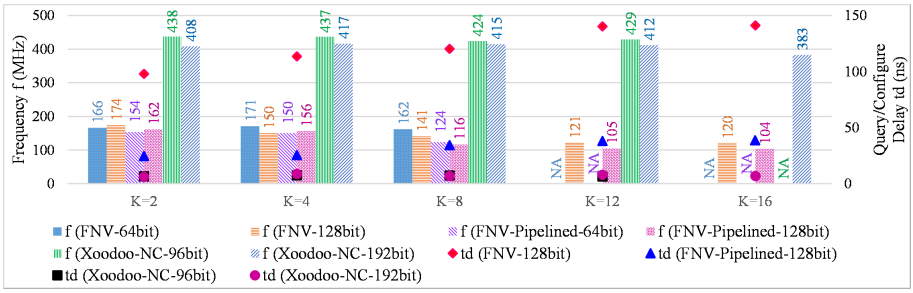


Figure 4.13: Maximum Operating Frequency and Delay of Bloom-1

Table 4.1: Comparison of Bloom-1, PBF, and CAM implementations on a Xilinx Virtex Ultrascale+

Bloom filter	k	fpr	LUTs	FFs	BRAM	LUTRAM	DSP	Max freq [MHz]	Cycles for lookup	Lookup Delay [ns]
Bloom-1 (Xoodoo-NC)	2	0.0002976	267	137	7.5	0	0	**435**	**2**	**4.6**
Bloom-1 (FNV-1a)	2	0.0002976	261	228	7.5	0	3	174	16	92.2
Bloom-1 (FNV-1a) Pipelined	2	0.0002976	649	1507	7.5	6	36	152	4	24.8
Bloom-1 (Xoodoo-NC)	12	$2.61\text{x}10^{-7}$	707	197	7.5	0	0	429	**2**	4.7
Bloom-1 (FNV-1a)	12	$2.61\text{x}10^{-7}$	1366	339	7.5	0	15	121	16	132.0
Bloom-1 (FNV-1a Pipelined)	12	$2.61\text{x}10^{-7}$	2706	1562	7.5	6	180	105	4	38.1
PBF (Uni-SBF,Xoodoo-NC)	12	$0.14\text{x}10^{-7}$	338	220	6	0	0	**434**	2	**4.6**
PBF (Bi-SBF,Xoodoo-NC)	12	$0.14\text{x}10^{-7}$	324	148	6	0	0	389	3	7.7
Direct CAM		0.0	6731	1145	320	1536	0	113	2	17.7
Custom Modified CAM		0.0	20'797	988	384	1536	0	225	3	13.3

Bloom filter versus CAM

Table 4.1 shows the comparison of the Bloom filter variants with CAM-based lookup architectures. The number of entries to be stored is taken as 1024. The Xilinx CAM blocks mentioned in [249] and [144] require 40×-50× more BRAMs than the Bloom-1 architecture. Furthermore, they necessitate approximately ≈20× more LUTs than the PBF architecture. The CAM block, with a size of 1024x96, generated using Xilinx design tools, operates at a frequency of only 112.75 MHz. The search delay of CAM increases as the size and the number of entries increase. Conversely, the search delay of Bloom filters is barely affected by the size of the filter and the number of entries. This is due to the constant number of locations to be read in the Bloom filter, which is equal to the number of hash functions employed. While the custom-modified CAM architecture achieves a higher operating frequency of 225 MHz, it consumes more FPGA resources, as shown in Figure 4.7 and Table 4.1. Although the latency is increased by one clock cycle to query the custom-modified CAM, the total lookup delay is lower compared to the directly generated CAM. The lookup delay of the Bloom filters, especially the Xoodoo-NC-based architectures,

is significantly lower than the lookup delay of CAM due to their substantially higher operating frequency.

Analysis of Bloom filter variants

Upon analyzing various implementations of Bloom filters, it becomes evident that there exists a constant trade-off between the false positive rate and memory utilization. To explore this trade-off, we examine two architectures: Bloom-1 and the PBF, proposed in Section 4.2.3. Table 4.2 shows the comparison between Bloom-1, PBF, and SBF. When comparing Bloom-1 and PBF, it is noticeable from Table 4.2 and Table 4.1 that PBF achieves a superior false positive rate with lower memory requirements, albeit at the expense of a higher number of hash bits. The Bloom-1 architecture with 12 hash functions requires only 84 hash bits, whereas PBF using Uni-SBF requires 144 bits to achieve a better *fpr*. However, Bloom-1 requires ≈ 5 times more memory than PBF. Comparing SBF and PBF, it is evident that SBF demands more memory while maintaining an equal number of hash bits. PBF using Uni-SBF exhibits less hash bit requirements and higher frequency compared to PBF using Bi-SBF. The number of cycles for lookup given in Tables 4.1 and 4.2 includes the number of clock cycles required for both hashing and memory accesses.

Table 4.2: Comparison of the properties of the Bloom filter architectures considered in this study

Bloom Filter	*fpr*	*k*	*n*	*m*	Hash bits	No. of cycles for lookup
Bloom-1	$2.61\text{x}10^{-7}$	12	1024	262144	84	2
Keeping k constant while achieving similar or better *fpr* as Bloom-1						
PBF (Bi-SBF)	$0.14\text{x}10^{-7}$	12	1024	49152	156	3
PBF (Uni-SBF)	$0.14\text{x}10^{-7}$	12	1024	49152	144	2
SBF	$8.74\text{x}10^{-7}$	12	1024	32768	192	13
Keeping hash bits constant while achieving similar or better*fpr* as Bloom-1						
PBF (Bi-SBF)	$0.49\text{x}10^{-7}$	6	1024	98304	90	3
PBF (Uni-SBF)	$0.49\text{x}10^{-7}$	6	1024	98304	84	2
SBF	$0.82\text{x}10^{-7}$	5	1024	131072	85	6

Power analysis

Power consumption is another important factor to be addressed. Our implementation results underline that CAM architectures are power-hungry compared to Bloom filters. The static power consumption of the FPGA occupies the lion's share of the total power consumption for all implementations. In terms of static power consumption, there is a minimal difference between

Bloom-1 and CAM implementations, with CAM exhibiting a maximum increase of 0.6% compared to Bloom-1. As for dynamic power consumption, Bloom-1 only contributes less than 1% of the total power consumed in FNV-1a and Xoodoo-NC based implementations. However, the dynamic power consumption of Bloom-1 employing the pipelined version of FNV-1a can rise up to 12% of the total power consumption. Notably, the dynamic power consumption of Bloom-1 based on FNV-1a and Xoodoo-NC is negligibly small compared to CAMs as shown in Figure 4.14. Although the dynamic power consumption of pipelined FNV-1a based Bloom-1 increases with an increasing value of k, it is still significantly lower than the power consumption of a CAM. CAMs, due to their parallel lookup operations, exhibit a dynamic power consumption of 33% of the total power when having a size of 1024, which is more significant compared to <1% of Xoodoo-NC-based Bloom-1 implementation. Another noteworthy point is that our modified CAM implementation consumes less power compared to the Xilinx IP CAM.

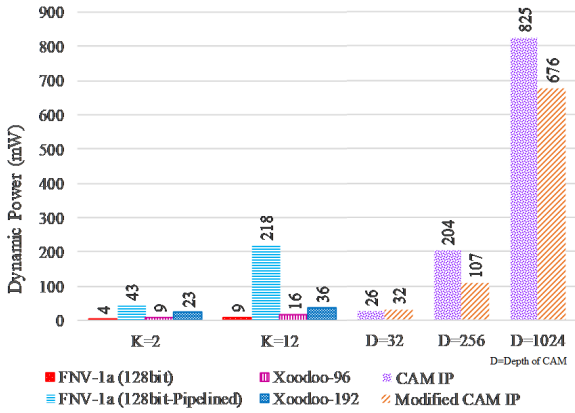


Figure 4.14: Dynamic Power Consumption of Bloom-1 (Using FNV & Xoodoo-NC), and CAM

Comparison with related work

A quantitative comparison with related work is shown in Table 4.3 for n=1024. It is clear that our Xoodoo-NC-based Bloom filter architectures with a 96-bit input significantly outperform previously proposed FPGA architectures. Our design drastically improves resource occupation and speed. Although the total lookup delay for the compared architectures is not provided, the delay solely associated with memory accesses exceeds the delay of a single query in

Table 4.3: Comparison with related work for $n = 1024$

Design	m	k	input size	LUTs	FFs	BRAMs	Frequency	fpr	Memory accesses	FPGA
PBF[93]	4096	10	10 byte	1495	1297	5	74 MHz	$8.89\text{x}10^{-5}$	2	Virtex 2000E
[126](SBF)	16384	8	8 byte	1058	1058	4	201 MHz	$5.74\text{x}10^{-4}$	4	Virtex-4
[126](CBF)	16384	8	8 byte	1188	1188	4	202 MHz	$5.74\text{x}10^{-4}$	4	Virtex-4
Ours, PBF (Bi-SBF)	49152	12	12 byte	338	220	6	389 MHz	$\mathbf{0.14}\text{x}10^{-7}$	2	VU+
Ours, PBF (Uni-SBF)	49152	12	12 byte	324	148	6	**434 MHz**	$\mathbf{0.14}\text{x}10^{-7}$	**1**	VU+
Ours, Bloom-1 [199] (Using xoodoo-NC)	262144	12	12 byte	707	197	7.5	429 MHz	$2.61\text{x}10^{-7}$	**1**	VU+

VU+ - Virtex Ultrascale+

our implementation based on Xoodoo-NC. Note, however, that the considered related work uses older FPGAs with 4-input LUTs, while our FPGA employs 6-input LUTs. Nonetheless, our reported count of LUTs is only approximately half of what is reported in the related studies, and the number of FFs ranges from one-fifth to one-tenth of their reported values. The difference in operating frequency can partially be attributed to the difference in silicon technology nodes between older and newer FPGAs. Nevertheless, achieving a frequency of 434 MHz for k=12 and an input size of 12 bytes is remarkable, thanks to the low logical depth of the Xoodoo-NC hash function employed in our approach.

Future-proofing

The present work targeted IPv4 as the network-layer protocol. However, the presented architectures are also adaptable for IPv6. The Bloom filter algorithm is based on the hashed fingerprint, and the algorithm is not affected by the size of the input (flow ID). The false positive rate of the Bloom filter is only a function of the number of hash functions, the number of elements to be inserted, and the total memory size. Therefore, it is not affected by the size of the input or hash collisions, irrespective of the network-layer protocol.

To handle IPv6 flows, the only adaptation needed is modifying the hash function to handle IPv6 flow IDs. This can be achieved by increasing the input size of Xoodoo-NC. The original Xoodoo permutation has a 384-bit input state, and in Xoodoo-NC, we used only a 96-bit input state as our requirement was only 96-bits. We can modify the Xoodoo-NC hash to accept either 192-bit (2 sheets of the Xoodoo state) or 288-bit (3 sheets) as the input state. Consequently, we must re-compute the number of rounds required to meet the avalanche criteria. The latency can still be a single clock cycle, and the computational resources will be slightly higher than Xoodoo-NC with a 96-bit input state. This adjustment remains more advantageous than alternative non-cryptographic hashes like FNV-1a and Murmurhash because as the input size increases, the latency and resource utilization of these hashes increase drastically.

4.3 A thorough analysis of the hardware friendliness of AMQ architectures

In order to fulfill the requirements of a fast lookup architecture on hardware for network security applications, we analyze existing probabilistic architectures to identify a suitable probabilistic alternative for CAMs. This analysis also helps to evaluate the performance of our proposed Bloom filter architectures compared to other AMQ data structures. The parameters for the data structures, such as input key size, are selected with a specific focus on network security. Our approach is to evaluate the suitability of transforming these algorithms into a hardware architecture, with an emphasis on achieving a speed-up on FPGAs. The efficiency in software platforms of these algorithms does not necessarily give us an idea of which algorithms perform the best in hardware. Given the abundance of available data structures, it is impractical to analyze each one comprehensively within the scope of this work. Our main focus is on memory requirement, speed, and cost of hashing and most of the well-performing data structures are derived from Bloom filters, Cuckoo filters, or hash tables. Therefore, we focus on the efficiency of these selected basic data structures in hardware. We implement the modified and optimized architectures of these data structures for evaluation [214]. Our goal is to evaluate the hardware friendliness of these data structures based on criteria such as lookup latency, operating frequency, resource consumption, and suitability for pipelining.

4.3.1 An insight into various AMQ data structures

Numerous data structures based on approximate membership queries have been proposed, with hash tables serving as the starting point. The best-known AMQ data structures are hash tables, Bloom filters [70], and Cuckoo filters [108]. Most of the prominent AMQ structures are either derived or optimized versions of these data structures. It is worth noting that hash tables can be transformed into exact lookup mechanisms when linear probing or chaining techniques are applied. Besides the classification based on Bloom filter, Cuckoo filter, and hash table approaches, there are other ways to categorize these data structures. For example, they can be classified as fingerprint-based/non-fingerprint-based as well as static/dynamic architectures. Fingerprint-based data structures store either a short digest of the key or the entire key itself, whereas non-fingerprint-based data structures do not store the key/digest. Static architectures require the whole set of keys to be available at the time of construction, whereas dynamic architectures can support insertion and deletion of items on the fly. The term 'dynamic' may also be used to represent the data structures where

resizing the table at runtime is possible. However, in this work, we do not consider the dynamic resizing of the table at runtime. This is because a complete reconstruction or rehashing of the data structure is required for dynamic resizing, which introduces overhead and hampers fair comparisons, particularly when considering hardware implementations.

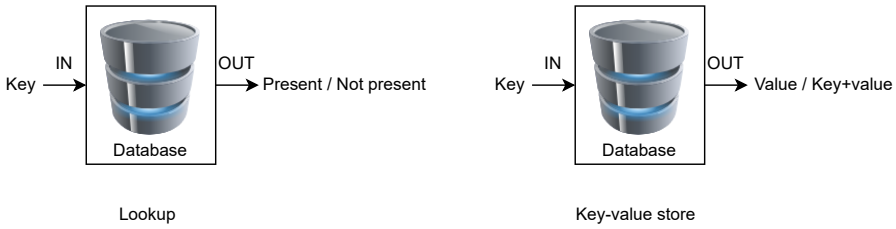


Figure 4.15: Functionalities of lookup and key-value store data structures

Since we categorize all data structures into three types: hash table-based, Bloom filter-based, and Cuckoo filter-based, the remainder of this section elaborates on these three types. The applications on focus are lookup and key-value stores. The difference between lookup data structures and key-value store data structures is depicted in Figure 4.15. Table 4.4 shows some of the prominent features of the basic data structures, which are relevant in network security applications. The table gives an average of the features from all three types. It is noted that there might be exceptions to these generalizations.

Table 4.4: Features of basic AMQ data structures

Data structure	Stores key	Stores fingerprint	Supports deletion	Supports lookups	Supports key-value store	Unlimited insertions
Hash table	✓	✗	✓	✓	✓	✗
Bloom filter	✗	✗	✗	✓	✗	✓
Cuckoo filter	✗	✓	✓	✓	✗	✗

Hash table and its variants

Hash tables are the simplest and most conventional way to implement a lookup or key-value storage architecture. To add an item x to a hash table with m locations, a hash function $h(x)$ is used to map the item to the table, where the key, value, or key-value pair are to be stored. A simple representation of the hash table is shown in Figure 4.16(a). Before we dive deep into the details, we should clarify the difference between a hash table and a hash map. In a hash table, the key is mapped to a location in the table using a hash function, and

the key is stored in that location. The hash map follows the same process but stores a key-value pair instead of a key. In this work, we use the term 'hash table' invariably for representing both hash table and hash map. Additionally, we refer to hash tables, which are not associated with any collision resistance mechanisms in the form of linear probing or chaining.

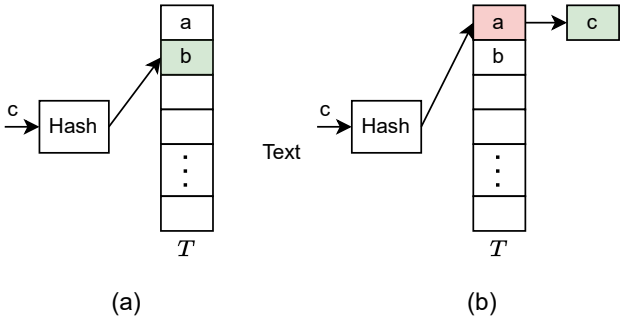


Figure 4.16: Representation of hash table. (a) A hash table with no collision during an addition, (b) Chaining in hash table during a collision.

Compared to exact associative array architectures such as CAMs, a hash table requires less memory for storing the same set of elements. However, hash tables are prone to collisions. Hash collisions cause two different keys to be mapped to the same location in the table, causing data loss. The load factor α of hash table, $\alpha = \frac{n}{m}$, must be kept to a low value to reduce the collisions, where n and m are the total number of items to be added and the number of buckets in the hash table, respectively. The load factor indicates the extent to which the hash table is occupied or filled, and a bucket is a hash-indexed location in the table that could store one or more entries. The probability of experiencing at least a single collision in a hash table is $\frac{m!}{(m-n)!m^n}$, and the total average number of collisions is $\approx \frac{n^2}{2m}$. This implies that there will be an average of $\approx$50% collisions if $n = m$. Hence, α must be lower than 0.5 to keep the collisions to a desired value. However, keeping α to a lower value would eventually cause under-utilization of memory, where a large amount of the allocated memory goes unused. The memory efficiency of a hash table can be optimized when the number of elements to be stored, the cardinality, is known upfront.

There are numerous approaches to minimize the effect of hash collisions, such as chaining, linear/quadratic probing, and double hashing. Chaining involves storing collided items as linked lists, while linear probing sequentially searches through table locations to find an empty slot. Double hashing utilizes two hash functions, with the second one acting as an offset to probe for an empty slot in case of collision. Figure 4.16(b) depicts an illustration of chaining, where a

new bucket is appended to accommodate the new element to the hash-indexed location where the collision occurred. However, chaining and linear probing are not really suited for hardware as the size of the table can increase indefinitely. Furthermore, these techniques worsen the time complexity of hash tables, as they may require probing through all the inserted elements in the worst-case scenario.

Robin Hood hashing: Robin hood hashing [78] is a hashing technique to compensate for collisions in hash tables. The principle of Robin Hood hashing is to keep the keys that are subjected to collisions close to the originally hashed slots. It uses probe sequence lengths (PSL) to find a slot during insertion. PSL is the number of probes an item makes before it finds a slot, and the PSL has to be stored along with the key. During a collision, probing starts, and if a key in a non-empty slot has a lower PSL than the key being inserted, they are swapped. This probing and swapping continue until an empty slot is found for the swapped keys. An illustration of how Robinhood hashing counteracts the hash collisions is depicted in Figure 4.17.

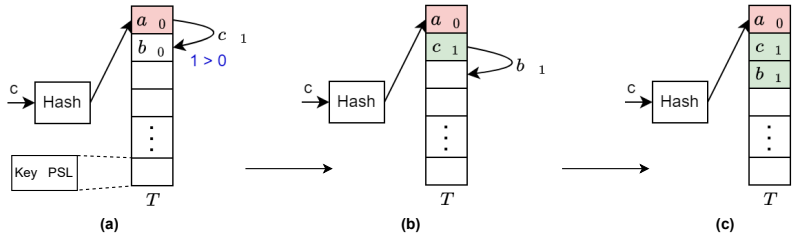


Figure 4.17: Illustration of Robinhood hashing during a collision. (a) an incoming key c is mapped to the same slot where key a is stored, indicating a collision. As both c and a has equal PSL of 0, c probes the next slot. PSL of c is increased to 1., (b) PSL of $c >$ PSL of b, and swapped. b probes the next slot, and PSL of b becomes 1., (c) b finds an empty slot and is added to that slot.

The most prominent variants of Robin Hood hashing are the Quotient filter (QF) [66] and the Counting Quotient Filter (CQF) [193]. QF is practically a linear probing hash table to reduce collisions, and QF does not store the whole key but a short fingerprint of the key in a chained manner. Both QF and CQF support deletion and resizing, and both exhibit better cache locality. Each fingerprint is divided into quotient and remainder, where the remainder is stored in the location indexed by the quotient. The collisions cause the stored remainders to be shifted to the subsequent slots linearly. In CQF, some of the remaining slots are dedicated to counters, which improve the performance of QFs on skewed datasets. Vector quotient filter (VQF) [194], which uses power-of-two-choices hashing, offers better throughput across load factors and

space efficiency over QF. Nevertheless, VQF still lags behind when compared to the Cuckoo filter. QF also occupies a bit more memory, about 20% more, than Bloom filters. Similar to hash tables, linear probing in quotient filters makes it unsuitable for hardware due to implementation complexity, dynamic resizing, and unpredictability of the number of cycles required for insertion. Shifting the elements linearly in memory is also not preferred on hardware as only one location can be read/written in a single cycle. Moreover, the performance declines as the occupancy of the QF becomes high, especially after 60% occupancy.

d-choice hashing and d-left hashing: Chained hash tables embrace an approach to finding a bucket randomly and linking the collided keys to the same bucket. This could adversely affect the update/query time complexity as the length of the chain in a random bucket becomes longer. d-choice hashing [203] introduced load balancing in the hash table of size m by applying d hash functions, and the key is inserted into the bucket which has the lowest load. At the instance of a tie when all the hashed buckets have the same load, a bucket is chosen randomly for insertion. d-left hashing [183] improved the load balancing by using d separate hash tables, with each table associated with a single hash function. In 2-left hashing, there are two tables of size $m/2$, and the key is hashed to both tables. The key is inserted into the left table if the hashed bucket in the left table has the lowest load or if there is a tie. The occurrence of ties is low in d-left hashing as the left hash table always has equal or more load compared to the right table. A graphical representation of 2-choice hashing and 2-left hashing is shown in Figure 4.18 (a) and Figure 4.18(b), respectively.

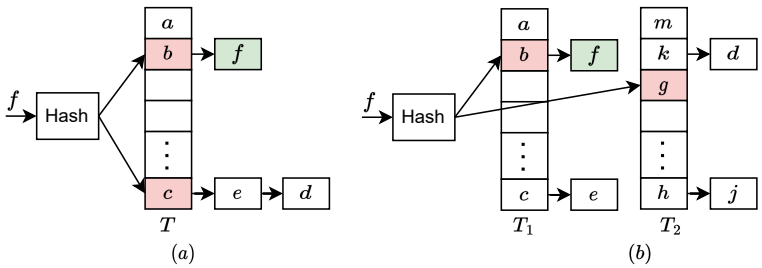


Figure 4.18: Illustration of 2-choice and 2-left hashing during a collision. (a) 2-choice hashing: the incoming key f is inserted into the bucket having the lowest load, (b) 2-left hashing: the incoming key f is inserted into the left table as there is equal load in both buckets.

Cuckoo hashing: Most hash tables encounter drawbacks such as extended worst-case lookup times when collision avoidance techniques are employed. In linear probing and double hashing, the worst-case lookup time is $O(log\ n)$,

where n is the number of elements inserted. In chained hashing, the worst-case lookup time is $O(log\ n/log\ log\ n)$, while two-way chaining offers a worst-case lookup time of $O(log\ log\ n)$. The best way to eliminate such long lookup times is to use perfect hashing where there are no collisions, but that is more of a hypothetical scenario. A hashing technique that can provide a constant lookup time of $O(1)$ is Cuckoo hashing [192]. Cuckoo hashing-based data structures provide better collision resistance than hash tables while offering a constant lookup time of $O(1)$.

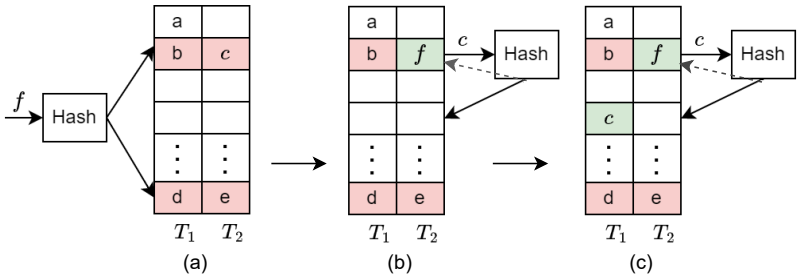


Figure 4.19: Illustration of Cuckoo hashing during a collision. (a) Incoming key f is hashed to two buckets, and none are empty., (b) f is swapped with the key c from a randomly chosen cell. c rehashed to find the indexed locations of c., (c) A cell in the hashed bucket is empty and c is placed in that cell.

A Cuckoo hash table (CHT) employs two tables and two hash functions, with the key being stored in only one of the tables. A graphical representation of CHT is shown in Figure 4.19. The red cells in the table indicate collisions, and the green cells indicate empty cells. When collisions occur during insertion, the existing key in the table is swapped with the incoming key, and the swapped key is rehashed and moved to the hash-indexed location if it is vacant. If the rehashed location is not empty, this process of swapping the existing keys continues until an empty location is found or a maximum allocated iteration value is reached. This swapping is otherwise termed 'kicking.' If insertion fails after reaching the maximum iteration value, the CHT needs to be resized, and all the elements should be rehashed using a new hash function. CHT provides a faster query time and better space occupancy, but the memory requirement is still high as it is required to store the complete key in the table. One way to overcome this problem is partial-key Cuckoo hashing, which is the basic principle of Cuckoo filters. A more detailed description of partial-key Cuckoo hashing is provided in Section 4.3.1.

Peacock hashing: Peacock hash table (PHT) is another form of linked/chained hash tables. Peacock hashing [154] is probably the most hardware-friendly solution to chained hash tables. PHT employs multiple hash tables where there

is a main table followed by multiple backup tables. The size of subsequent backup tables is scaled down by a scaling factor. All the tables use different hash functions. A graphical representation of PHT is shown in Figure 4.20. The incoming keys are directed to the larger tables first, and if there are collisions, the keys are sent to backup tables. The key is discarded if a predefined probing value is reached and still no empty slot is found. In order to make the querying faster, a fast filter-preferably a Bloom filter-is associated with each of the backup tables in the on-chip memory, and the hash tables are stored in the off-chip memory. Despite being a chained hash table, the number of probes due to collision is limited to the number of tables. The worst-case update and query time complexity is $O(log\ log\ m)$, where m is the size of the main table.

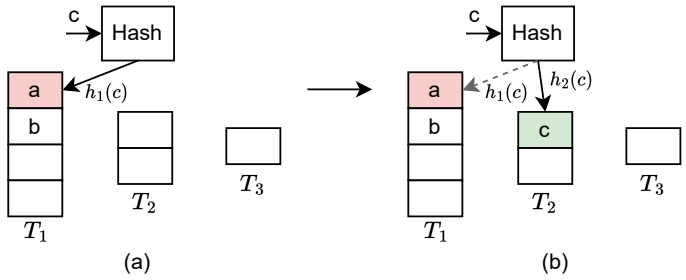
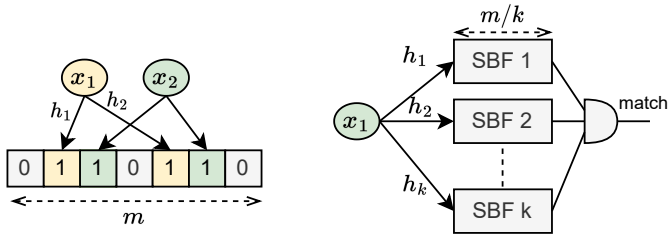


Figure 4.20: Illustration of Peacock hashing during a collision. (a) An incoming key c is hashed to the largest table T_1, and there is a collision., (b) c is hashed to the subsequent table T_2, and the mapped cell is empty. c is added to the empty cell.

Bloom filter and its variants

A Bloom filter [70] is a space-efficient probabilistic data structure that is commonly employed to perform constant-time membership queries in a set S of n elements, $\{x_1, x_2, ..., x_n\}$, such that $S \subseteq U$ where U is a universal set. Proposed by Howard Bloom in 1970, standard Bloom filters became an integral part of most applications where conventional membership queries turned out to be impractical given that the amount of data to be handled is large. The structure and operation of Bloom filter is detailed in Section 4.2.1 Compared to hash tables, the use of multiple hash functions in Bloom filters somewhat eliminates the requirement of hash-collision avoidance mechanisms. Bloom filters are simple, easy to construct, and memory-efficient, but still have some limitations such as no support for deletion. Many different variants of Bloom filters have been proposed to address these limitations [116]. There exist more than 50 variants of Bloom filters and a lot more optimized Bloom filter

architectures, but there is always a trade-off between memory, accuracy, and speed among these variants. Most of the variants are a workaround to eliminate the limitations of Bloom filters.



(a) Standard Bloom filter (b) Parallel Bloom filter

Figure 4.21: Bloom filters

Support for deletion: As Bloom filters cannot support deletion, Counting Bloom filter (CBF) [109] is introduced to enable deletions in an SBF, but at the cost of higher space utilization. A CBF follows the same structure as an SBF, but a counter replaces each single-bit slot in the BF to keep track of insertions. Whenever an item is inserted, the hash-indexed counters are incremented by one, and during deletion, the corresponding counters are decremented by one. Numerous variants and optimizations to CBF have been proposed in recent times to improve CBF. Deletable Bloom filter (DIBF) [206] tries to address the higher memory requirement of CBF while offering support for deletion. DIBF divides the bit array of size m into r regions and keeps an r-bits-sized bitmap to encode whether or not a region is collision-free. Each bit in the bitmap represents the collision status of each region. However, the trade-off is a higher FPR as the size of each region is a small fraction of m. Moreover, the deletion becomes impossible when every region has at least one collision, and DIBF acts like a normal BF with a worsened FPR. Spectral Bloom filters [86] and Space-code Bloom filters [152] also support deletions by following a similar approach as CBF but targeting multi-sets.

Dynamicity: SBF offers the advantage of supporting unlimited insertions at the cost of a higher FPR. However, it does not support dynamic resizing, where the existing filter can be resized on the run while retaining the same FPR. Dynamic Bloom filter (DBF) [122] and Scalable Bloom filter (SCBF) [52] propose dynamic resizing of the filter, adapting to dynamic datasets. Dynamic and scalable Bloom filters follow the same data structure consisting of a series of small SBFs appended sequentially, the difference being the sizes of the incremental SBFs. DBF has the same size m for all the SBFs, whereas the ith SBF of SCBF has a size equal to $m \times a^{i-1}$, where a is a positive integer. Both

approaches are slower and have a lower FPR for the same amount of memory compared to SBF. SCBF has a lower FPR than DBF. DBF is faster than SCBF as it uses homogeneous SBFs, while SCBF employs heterogeneous SBFs.

Enhanced FPR: SBF, being a simple data structure, encounters difficulties in enhancing the FPR without any trade-offs. Retouched Bloom filter (RBF) [96] improves the FPR of the Bloom filter by trading for some false negatives. The removal of false positives in RBF is achieved by clearing the corresponding bits.

Speed optimizations: The size of the Bloom filter is larger compared to the size of the cache line, causing cache misses. The issue of cache misses, along with the poor data locality of Bloom filters, is addressed by Blocked Bloom filter (BBF) [197]. A BBF has b small-sized standard Bloom filters, each with a size less than or equal to the cache line. An SBF block is chosen using a hash function, and each item is mapped to that SBF using k hash values. While this approach improves speed, it comes at the expense of a higher FPR due to the possibility of quickly filling the small SBF. Consequently, the filter size needs to be increased to compensate for this limitation. The optimization in BBF is to improve the software's runtime performance on a hardware platform, but not a hardware-oriented design that may leverage the same performance when translated onto hardware. Another technique to enhance the speed is to reduce the hash computations. Through double-hashing Bloom filter, Kirsch et al. [150] have shown that only two independent hash functions are enough to generate all the required hash values in the form of $h_1(x) + i * h_2(x)$ without any increase in the asymptotic FPR, where $h_1(x)$ and $h_2(x)$ are hash values of the item x, and i is an arbitrary value. Some of the optimizations on Bloom filters focus on reducing the latency to a single memory access cycle in the likes of Bloom-1 [198] and our proposed PBF.

Cuckoo filter and its variants

A CHT stores the keys in the hash-indexed buckets, contributing to a larger memory footprint when the key size is large. When it is required to perform processing at line rate, confining the lookup architectures within the on-chip memory is of utmost importance. Partial key Cuckoo hashing [163] helps resolve this issue by storing only a compact fingerprint or digest of the key. Cuckoo filter (CF) [108] employs partial key Cuckoo hashing as its main operation. The insertion, query, and deletion procedures are very similar to Cuckoo hashing, the difference being the computation of the second hash index. The second hash index in CF is computed by hashing the fingerprint and then XORing the fingerprint with the first hash index.

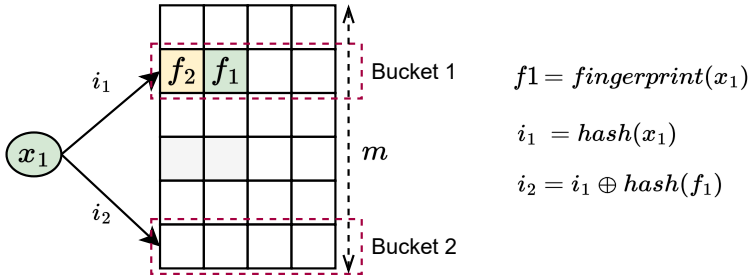


Figure 4.22: A representation of Cuckoo filter

Figure 4.22 depicts a representation of the Cuckoo filter. It is composed of a memory block with m locations where each location is termed a bucket. Each bucket is indexed by a hash value and has b entries. For optimal space efficiency, the value of b is set to 4. Every item is mapped to two buckets using hash values. The indices of the bucket B_i for each item x_i are i_1 and i_2, where $i_1 = hash(x_i)$ and $i_2 = i_1 \oplus hash(f)$. The fingerprint of x_i is denoted as f and is generated using another hash function. During an insertion process, if either of the two buckets B_1 or B_2 has an empty slot, the item is inserted into that slot. If none of the slots are empty, a bucket and entry are chosen randomly, and the existing item is swapped with the incoming item, similar to Cuckoo hashing. This process of swapping the item is called kicking. The bucket index of the swapped item f_s is then computed using $i \oplus hash(f_s)$, where i is the existing location of f_s. The swapped item f_s is then attempted to be added to the new hash-indexed slot. If that slot is not empty, this process of kicking continues until it finds a new slot or the maximum value of probing is reached. The Cuckoo filter can deny an insertion if the table is full or the maximum kicking iterations has reached. In such cases, resizing and rehashing are required, which is impractical when the processing is at line rate.

Compared to Bloom filters, the Cuckoo filter is faster, more space efficient, and supports deletions. Irrespective of all the advantages, there are some drawbacks that are very critical when implementing it on hardware. The complexity of implementation on hardware is one drawback. Moreover, as the load factor of the filter increases, the insertion length becomes indefinite due to the kicking process. The number of kicks could reach the maximum value of the iteration, which is set to an arbitrary value of 500 by Fan et al. [108]. Another disadvantage is the insertion limit, where no further insertions are possible once the filter reaches its full capacity. Insertion of duplicate items can adversely affect the false positive rate and restrict the number of unique items that can be added to the filter. However, denying the insertion of duplicates is not feasible because

two different items with the same fingerprint can be mapped to a single bucket due to collisions, and denying the insertion would cause the removal of the only existing fingerprint during deletion.

Many different variants of Cuckoo filters have been proposed to enhance the performance of Cuckoo filters on various aspects such as space efficiency, FPR, and speed. Nevertheless, there is always a trade-off between these aspects more often than not. Some of the important variants which are worth mentioning are included here.

Enhancing speed: Morton filter [74] is a compressed version of the Cuckoo filter in which data storage is more dense. Morton filter can achieve an improved memory access time by efficiently utilizing the cache, which also helps achieve a higher insertion, lookup, and deletion throughput on ARM architecture compared to CF. However, reduced support for various fingerprint sizes limits the application range of the Morton filter. Vertical CF [114] also reduces the insertion time of the CF by increasing the number of buckets for each item. Vacuum filter [233] is space efficient and faster than Bloom and CF while achieving the same FPR as CF. Vacuum filters follow the same data structure as CF but with better data locality. The improved data locality is achieved by dividing the table into multiple chunks, similar to a Blocked BF, and using two different insertion algorithms based on the number of entries to be stored. This division also helps to keep the table size not a power of two in contrast to the Cuckoo filter. This is efficient when the number of elements to be stored is not a power of two. Nevertheless, keeping the size of the filter non-powers of two makes things more difficult on hardware.

Enhancing FPR: Adaptive CF [180] improves the FPR of CF by eliminating previously occurred false positives. This halts the repeating occurrence of the same false positives. Length-aware CF [156] is also able to reduce the FPR but with added storage requirements. D-ary CF [244] improves space utilization by sacrificing insertion, deletion, and query performances. By utilizing d hash functions, D-ary CF achieves a superior FPR compared to CF using the same memory allocation. However, due to decreased speed, it is not considered a preferable replacement for CF.

Dynamicity: Various approaches were proposed to enable dynamic resizing in CF, similar to SBF. One such approach is Dynamic CF (DCF) [80], which appends multiple homogeneous CFs together when it is required to extend the size of the filter, similar to DBF. Moreover, DCF can optimize space utilization by merging under-utilized CFs. While DCF can support dynamic resizing, the lookup performance is worse than CF, as it is required to access multiple linked CFs. Consistent CF (CCF) [171] is a further improved variant of dynamic CF, where each CCF is composed by attaching multiple index-independent

Cuckoo filter (I2CF). Each I2CF can have k buckets, and the value of k is a variable depending on the cardinality. The sparse I2CF can also be compressed. Nevertheless, dynamic resizing in CCF is applicable only for multi-sets and makes the implementation task more cumbersome on hardware.

Table 4.5: FPR and time complexities of data structures

Data structure	False positive rate	Time complexity (Update)		Time complexity (Query)	
		Best case	Worst case	Best case	Worst case
Hash table	NA	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Cuckoo hashing	NA	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Peacock hashing	NA	$O(1)$	$O(log\ log\ m)$	$O(1)$	$O(log\ log\ m)$
2-choice hashing	NA	$O(1)$	$O(log\ log\ n)$	$O(1)$	$O(log\ log\ n)$
Quotient filter	$1/2^{r*}$	$O(1)$	$O(log\ m)$	$O(1)$	$O(log\ m)$
Bloom filter	$(1-e^{-k.n/m})^k$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Counting Bloom filter	$(1-e^{-k.n/m})^k$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Dynamic Bloom filter	$1-(1-e^{-k(n-c\lfloor n/c \rfloor/m})^k$	$O(k)$	$O(k)$	$O(k.s)$	$O(k.s)$
Deletable Bloom filter	$(1-(1-(1/(m-r)))^{k*n})^k$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Parallel Bloom filter	$(1-e^{-k.n/m})^k$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Bloom-1 filter	Refer [212]	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Retouched Bloom filter	$(1-e^{-k.n/m})^k * (1-z/p_1.m)^k$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Double-hashing BF	$(1-e^{-k.n/m})^k$	$O(k)$	$O(k)$	$O(k)$	$O(k)$
Cuckoo filter	$2^{-(C*\alpha-2)}$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Morton filter	$1-(1-1/2^f)^{\alpha_L.B.S}$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Vacuum filter	$2b\alpha/2^f$	$O(1)$	NA	$O(1)$	$O(1)$
D-ary CF	$k/2^f$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Dynamic Cuckoo filter	$2.b.s/2^f$	$O(1)$	$O(n)$	$O(1)$	$O(2.b.s)$
Consistent CF	$s.k.b/2^f$	$O(1)$	$O(N.log\ m)$	$O(1)$	$O(k.b.s.log\ m)$

Rows in yellow - Hash variants, rows in cyan - Bloom filter variants, rows in purple - Cuckoo filter variants
m=No. of buckets; n=No. of items; f=Fingerprint size; k=No. of hash functions; b=No. of entries in a bucket; s=No. of filters
c=capacity of a single BF; r=No. of regions in the BF; p_1=$1-e^{-kn/m}$; z=No. of bits reset in BF; α_L=logical load factor
α=load factor; C=Bits per item; B=buckets accessed per negative lookup; S=logical slots per bucket; N=max probes allowed

Other filters which are more space efficient than Cuckoo filters: Xor filter [119] and Binary fuse filter [120] offer smaller memory footprint than Cuckoo filters. While the update time of Xor filters and Binary fuse filters is longer than that of Bloom and Cuckoo filters, they offer faster query times. However, these filters are immutable, which means that dynamic or in-line updates are impossible, making them ill-suitable for streaming applications. The full set of keys is required at the time of construction, and updating a key or a new set of keys to the filter would require the filter to be rebuilt.

To conclude this brief description of these data structures, a summary of the time complexities and FPR of important data structures are presented in Table 4.5. We can infer from the table that the hash table and Cuckoo filter-derived structures exhibit the best-case update/query complexity of $O(1)$, and their worst-case update/query complexity would increase as the table fills up. In contrast, Bloom filter-based data structures have constant update/query complexities irrespective of the load.

4.3.2 Hardware architectures

The design requirements of lookup and key-value stores are not the same for every application. This has to be taken into consideration while choosing the best data structures. Our primary focus is on network applications where the cardinality is predictable, and the size of the key/flow ID is constant throughout. The most suitable data structures from the existing ones are filtered based on the above-said criteria for hardware evaluation.

Architectures for lookups: For lookups, where the requirement is only the presence of an item, hash tables and hash table variants are probably a luxury in terms of memory requirement and implementation complexity. Hash tables are required to store the key, which consumes a large amount of memory when the key size and the cardinality are large. Moreover, the requirement of collision-resistant mechanisms makes it slower, where fast lookup is a necessity for high-speed processing. With inherent collision-resistant mechanisms, Bloom filters and Cuckoo filters, along with their numerous variants, offer good accuracy within the lowest memory requirement.

Architectures for key-value stores: For applications where getting the presence of an element is not enough and either the key or value has to be stored and/or returned, Bloom and Cuckoo filters have limitations. Bloom filters do not store keys or values, and recovering the key from a Bloom filter is impossible. Cuckoo filters store a fingerprint, but it is not possible to retrieve the key from the fingerprint. The case is similar to all the data structures which store only a hashed digest. This makes hash tables and their variants more suitable for probabilistic key-value stores. Nevertheless, hash tables must be associated with collision-resistant mechanisms. From a hardware perspective, the hardware suitability of mechanisms such as chaining and linear probing is low. Peacock hashing and Cuckoo hashing are more hardware-friendly as the table size is fixed and is suitable for static data sets.

Hardware implementation details

Optimized hashing: Hashing is one of the most important building blocks of AMQ data structures, as the overall throughput of the system can be affected by the speed of the hash computation. Non-cryptographic hashes with satisfactory avalanche properties are preferred as it is faster and has a low logical depth compared to cryptographic hashes. As presented in Chapter 3, we have proposed a fast non-cryptographic hash function Xoodoo-NC, derived from the Xoodoo permutation. In all our implementations, Xoodoo-NC is employed to generate the hash values. Even though the input data (key) size is fixed to 96-bit in our

evaluations, a varying key-size will only have a negligible effect on the AMQ algorithms. The only change that would require is to change the input block size of the hash functions. The false positive rates of AMQ data structures are not a function of the resulting hash values as observed from Table 4.5.

Hash table: Figure 4.16 illustrates the structure of the hash table (HT). The implementation of the hash table is straightforward and is implemented using block RAMS (BRAM) as the memory block. The depth of the table is m, and the width is equal to the size of the key and value. In a naive, straightforward implementation of the hash table, collisions are not handled, and it is implemented here to serve the purpose of a basic unit for comparisons.

Cuckoo hash table: Figure 4.19 depicts a visual representation of the Cuckoo hash table (CHT). In contrast to the illustration, the tables T_i and T_2 are implemented as distinct BRAM blocks where each memory block has one entry of each bucket. Such an implementation halves the word length of one bucket, which is equal to $2 * (size\ of\ the\ key + size\ of\ the\ value)$. Since both the memories can be accessed in parallel, the latency to access a bucket is still a single clock cycle..

Peacock hash table: To implement PHT, the main table and all the backup tables, as shown in Figure 4.20, are implemented as separate BRAM blocks. The main table has a size of m and a scaling factor of $r = 2$, which is the optimal hardware value of r. Since our goal is to analyze the performance of only the hash table, no fast lookup mechanisms are employed, as presented by Kumar et al. [154]. PHT has $(1 + log_{1/r}m)$ tables, and the last table has only one location. The last table is eliminated in the implementation as it is not possible to generate a BRAM block with only 1 location. All memories are accessed in parallel to keep the update/query latency to a bare minimum, contrasting with the original algorithm where the ith table is accessed only when results are not found in the $(i-1)^{th}$ table.

Bloom filter and Parallel Bloom filter: The implementation of the SBF is straightforward and hassle-free. The data structure as depicted in Figure 4.21, is implemented with a BRAM block of depth m and width 1-bit. The number of hash functions k is eight, and these hashes are generated using Xoodoo-NC. SBF with eight hash functions requires a total of 136 hash bits, and Xoodoo-NC generates a 192-bit output, which is split into eight hash values of sizes $log_2 m$ bits each. For PBF, all the k sub-blocks of size m/k are implemented separately using BRAM. Each sub-block is accessed in parallel, which keeps the memory access latency to a single cycle. Xoodoo-NC is used to generate k hash values for PBF in a similar way as it is for SBF.

Cuckoo filter: In contrast to Bloom filters, implementing the CF on hardware requires a bit more effort and engineering. The data structure of CF is shown in Figure 4.22. The whole table of CF is implemented as a single BRAM block. The depth of the memory is m, and the width of the memory is $b * f$, where f is the size of the fingerprint and b is the number of buckets. The memory is implemented as a true dual-port BRAM with dedicated ports for reading and writing. It requires two clock cycles to read the contents from the buckets. Our CF implementation is optimized for the number of memory accesses. During an update operation, one bucket is read in parallel with the computation of the address of the second bucket. The second bucket is only read if no empty slots are found in the first bucket. A similar process is applied during the query operation, resulting in the best-case scenario of a single cycle for memory accesses, both in update and query operations.

4.3.3 Hardware evaluation

The evaluation is performed separately for lookup and key-value store data structures. The analysis is conducted in terms of FPR or accuracy, latency, hardware resource usage, speed, insertion throughput, and implementation complexity. The evaluation is targeted on a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVC2104-1-E) using the Vivado 2020.2 design tool.

Evaluation of Lookup architectures

For the evaluation, the architectures chosen are CF, SBF, and PBF. A total of 16KB of memory is allocated to all the data structures, and the size of the tables of each structure is determined based on the allocated memory. The number of bits per item is fixed to 12, and the number of items n to be inserted equals 10922. This value of n makes the load factor α of CF to be 0.67 and m=4096. The CF consists of 2 buckets, and each bucket has b=4 entries. The fingerprint size of CF is set to 8 bits. The maximum kick value is set to 500, the optimal value employed in the original article [108]. In the evaluation, memory access cycles refer to the sum of memory read and write cycles. The number of hash functions for Bloom filters is k=8, which makes m=131,072 for SBF and PBF. The size of a single block of PBF is m/k=16384. The hardware evaluation is performed using synthetic datasets on a Virtex UltraScale+ (xcvu9p-flga2104-2L-e) platform.

False positive rate and space-occupancy: The CF demonstrates a superior FPR compared to Bloom filters, as evidenced in Table 4.6, given a fixed number of bits per item. This implies that CF has better space occupancy than Bloom

filters. CF can have 2*b duplicate entries, where b is the number of entries in a bucket. To store a similar number of duplicates, a counting Bloom filter would require a 3-bit counter in each location, resulting in 3 times more memory requirement compared to SBF. However, allowing duplicates will deplete the space and cause higher false positives for CF, which does not happen with SBF.

Latency: CF has a query time complexity of $O(1)$, while SBF and CBF have a query time complexity of $O(k)$. However, PBF outperforms CF in all other aspects, except for the number of hash bits and FPR when considering a fixed number of bits per item. Both the update and query time complexity of PBF is $O(1)$, thanks to the parallel accessing of memory blocks. Memory access cycles to insert an element are constant for Bloom filters irrespective of the load factor. In comparison, CF requires more memory access cycles for insertion due to probing/kicking when the filter starts to fill up. The analysis provided in Table 4.6 shows that the percentage of memory access cycles due to probing is only around 5.7% more than the actual requirement when the load factor is 75%. However, memory accesses due to probing increase to 14.2% when the load factor is 85%, and then sharply rise to 48.2% when the load factor is 95.5%. The total number of memory access cycles required for PBF is less than that of CF for load factors up to 85%. Moreover, when the load factor is 95.5%, the total number of memory access cycles for PBF is less than half that of CF.

Table 4.6: FPR and memory access delay vs various load factors for lookup data structures

Load factor	False positve rate		Total memory access cycles			total probe	% of cycles for
	CF	SBF, PBF	SBF	PBF	CF	cycles (CF)	probing (CF)
35%	$9.8\text{x}10^{-7}$	$5.5\text{x}10^{-5}$	91,200	11,400	11,480	6	0.05%
50%	$1.0\text{x}10^{-4}$	$6.0\text{x}10^{-4}$	131,200	16,400	16,801	72	0.43%
65%	$1.1\text{x}10^{-3}$	$2.7\text{x}10^{-3}$	170,400	21,300	22,709	500	2.20%
75%	$3.4\text{x}10^{-3}$	$6.0\text{x}10^{-3}$	196,560	24,570	27,699	1,568	5.66%
85%	$7.8\text{x}10^{-3}$	$1.1\text{x}10^{-2}$	222,800	27,850	35,361	5,036	14.24%
95.5%	$1.6\text{x}10^{-2}$	$2.0\text{x}10^{-2}$	250,320	31,290	67,658	32,584	48.16%

Average of 1000 runs; Total memory = 16KB, # of items = 16384 (at α=1.0)
CF-Cuckoo filter, SBF-Standard Bloom filter, PBF-Parallel Bloom filter

Performance on hardware: Table 4.7 illustrates the performance results achieved on FPGA. The hardware resource requirements and maximum operating frequency of SBF and PBF are better than CF while maintaining the same number of bits per item but with a lower FPR and more hash bits. CF requires more than two times the number of LUTs compared to the Bloom filter counterparts. Furthermore, CF demands two clock cycles for hashing as the second hash computation depends on the first. Nevertheless, CF can still achieve a best-case query latency of 2 cycles if the memory read of one bucket can be performed in parallel while computing the second hash index. Since the number of probes during insertion can go up to 500 as α increases, pipelining becomes difficult. PBF has a constant query latency of only two cycles (one

cycle for hashing and one for memory read), irrespective of the load. Even though SBF is the simplest to implement on hardware, it requires nine cycles for querying. In terms of throughput, PBF has an edge over CF and delivers the best insertion throughput, while SBF and CBF exhibit low throughput due to their low latency.

Table 4.7: Performance of Lookup architectures on FPGA

	Cuckoo filter (CF)	Bloom filter (SBF)	Counting Bloom filter (CBF)	Parallel Bloom filter (PBF)
Load Factor (α)	0.67	-	-	-
FPR (ϵ)	**0.0015**	0.0031	0.3124	0.0031
# of hash bits	**24**	136	136	112
Best case query Latency	**2 cycles**	9 cycles	9 cycles	**2 cycles**
LUT	930	402	412	**336**
FF	267	259	288	259
BRAM (36Kb)	4	4	4.5	4
Max. frequency	435 MHz	476 MHz	435 MHz	**488 MHz**
Insertions/second (Million)	138	28	26	**163**
Implementation complexity	++++	+	++	++

Memory = 16KB, # of items = 10922 (at α=0.67), bits per item = 12

Evaluation of key-value stores

The evaluation compares three types of hash tables, which are conventional HT, CHT, and PHT, as described in Section 4.3.2. Each data structure is allocated a fixed memory of 224KB. CHT has two tables, each with a size of m=8192 buckets. Each bucket has one entry per table. The memory accesses are optimized for CHT, similar to the optimizations in CF as discussed in Section 4.3.3. The maximum kick value is set to 500. For PHT, the main table size is set to 8192, and the scaling factor is defined as $r = 0.5$. The total number of tables t in PHT is 13 with each ith backup table having a size of $m/(\frac{1}{r})^i$, where $1 \leq i \leq t-1$. HT has a size m of 16384 buckets. The key size is 96 bits, and the value size is 16 bits. The load factors of CHT is $n/2m$, PHT is $m(1-r^t)/(1-r)$, and HT is n/m. In the evaluation, accuracy refers to the ratio of correctly queried items from the table and the total number of items inserted into the table.

Loading the table and accuracy: The accuracy and probing cycles in hash tables increase with increasing load factor α as shown in Table 4.8. In HT, the probability of failure in inserting an item is high as no collision-resistant mechanisms are employed. PHT has an accuracy close to 100% when the load factor α=0.6, but the accuracy and number of failed insertions worsen as $\alpha > 0.6$. For CHT, the accuracy is close to 100% even at α=0.88. The accuracy of CHT can be improved by increasing the number of entries for each

Table 4.8: Accuracy and memory access delay vs various load factors for key-value stores

Load factor	Accuracy			Total memory access cycles			% of cycles for probing	
	HT(%)	PHT(%)	CHT(%)	HT	PHT	CHT	PHT	CHT
35%	84.5	100.0	100.0	11,468	13,635	12,052	15.9	1.9
50%	78.6	100.0	100.0	16,384	21,246	18,751	22.9	7.5
65%	73.6	99.5	100.0	21,300	30,959	29,720	31.2	21.8
75%	70.3	97.4	100.0	24,576	39,928	45,909	38.4	40.3
85%	67.3	93.7	100.0	27,852	51,518	106,891	45.9	70.2
90%	65.9	91.3	99.3	29,492	58,719	328,079	49.8	89.6
95%	64.5	88.9	96.4	31,130	66,185	847,033	53.0	95.8

Average of 1000 runs; Total memory = 16KB, # of items = 16384 (at α=1.0)
HT-Hash table, PHT-Peacock hash table, CHT-Cuckoo hash table

bucket. Then, the table depth must be reduced accordingly to keep the memory requirement constant.

Latency: CHT has a worst-case query complexity of $O(1)$, whereas the query complexities of other hash tables vary with the chain/probe lengths. CHT has an average update time complexity of $O(1)$, but the probing length increases as α increases. When $\alpha > 0.5$, the number of probes increases drastically and can go up to the maximum probe length of 500. PHT is very much like a chained hash table, but since the table size is fixed, the maximum probe length is limited to the total number of tables in PHT. The number of memory access cycles increases for CHT due to probing as α increases. For example, the increase is ≈4417% when the load factor increases from 0.5 to 0.95. The increase in memory access cycles is considerably lower for PHT, which is an increase of ≈211% when the load factor increases from 0.5 to 0.95.

Performance on hardware: Table 4.9 presents the performance outcomes on FPGA. All the tables in PHT and CHT can be accessed in parallel, making the query latency of all three architectures equal to 2 clock cycles, including hashing and memory access cycles. It is evident from the table that there is always a trade-off between space occupancy, speed, and accuracy. The conventional hash table is the least accurate, but it is faster and consumes the least amount of resources.

The memory footprint is higher for PHT as the memory is split into many smaller blocks, which consumes a lot more memory than a large single block. This is because the targeted FPGA can only construct the memory with 18Kb/36Kb BRAM modules, even if the required memory size is less. Moreover, these blocks, spread around the main logic, cause the routing delay to increase, eventually causing the maximum operating frequency to dip. One remedy is to keep a limited number of tables, but this would cause the accuracy to decline because of the increased number of insertion fails, as the probing efficiency during collision is limited by the number of blocks. For example, keeping the total number of

Table 4.9: Performance of hash table architectures on FPGA

	HT	CHT	PHT
Accuracy at α=0.95	64.5%	96.4%	88.9%
# of hash bits	14	26	91
Best case Query Latency	2 cycle	2 cycle	2 cycle
LUT	606	1745	1399
FF	268	971	267
BRAM (36Kb)	50	50	79
Max. frequency	357 MHz	322 MHz	232 MHz
Insertions/second (Million)(α=0.75)	119	68	42
Implementation complexity	Easy	Difficult	Medium

Memory = 224KB, # of items = 15565 (at α=0.95), bits per item = 118

HT-Hash table, PHT-Peacock hash table, CHT-Cuckoo hash table

tables to 4 and each table having equal sizes of 4096 buckets would cause the accuracy to drop to 46% when the load factor is 1.0, whereas the conventional PHT achieves 86% accuracy. In a way, CHT employs the same principle as PHT, which probes for an empty slot during a collision, but with better space occupancy. However, the complexity of the architecture is high for CHT when implemented on hardware. This results in higher resource consumption in terms of LUTs. Nevertheless, CHT leverages significantly better operating frequency than PHT. In terms of throughput, CHT has better insertion throughput until α=0.75, but the throughput drops drastically for CHT and is only 6 Million items/s when α=0.95. In contrast, PHT maintains a throughput equal to 30 Million items/s at α=0.95.

Partial key Cuckoo hashing for better performance: Partial key Cuckoo hashing is one way to compromise the high resource and memory requirement of CHT and improve the operating frequency if storing/retrieving the key is not required. This approach involves storing the value together with the fingerprints in a CF. However, it is required to have an optimal fingerprint size to mitigate the adverse effect on FPR to some extent. The empirical results show that storing values in a CF, using an 8-bit fingerprint, can leverage similar accuracy as CHT while CF only uses ≈21% of the memory that is required by CHT. Moreover, CF can run at a significantly higher operating frequency and insertion throughput compared to CHT. Table 4.10 shows the results of employing CF as a partial key-value store.

Table 4.10: Results of partial key Cuckoo hashing as partial key-value store

Accuracy (α=0.95)	Total cycles	% of extra cycles for probing	LUT	BRAM (36Kb)	Max frequency	Insertions/second (Million)(α=0.75)
100%	62,740	44.47%	1442	11	417 MHz	128

Memory = 48KB, # of items = 15565 (at α=1.0), bits per item = 25, Throughput in Million item/s

Additional analysis

Memory access cycles: The total memory access cycles during the insertion of elements for various data structures are depicted in Figures 4.23 and 4.24. The load factor (α) plays a crucial role in determining the memory access cycles for hashing-based data structures like CHT, PHT, and CF. With an increased load factor, the memory access cycles increase for CHT, PHT, and CF because of the kick/probe operations during collisions. As observed from Figure 4.23, there is an exponential increase in the memory access cycles for CHT when $\alpha > 0.75$. Nevertheless, the increase is minimal for PHT because the maximum number of probes is limited by the number of tables. There is a gradual increase in memory access cycles for CF with load factor, as shown in Figure 4.24, but significantly lesser because of the higher number of entries per bucket than CHT. For Bloom filters, the increase in memory access cycles is constant because the number of cycles per insertion is constant throughout, irrespective of the load factor.

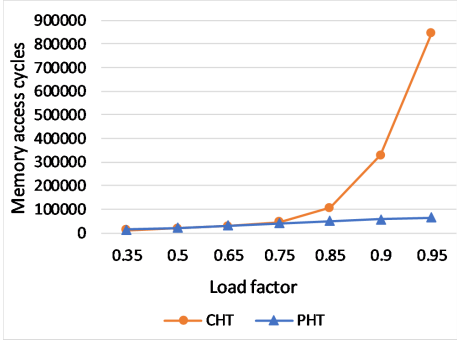


Figure 4.23: Memory access cycles for hash tables

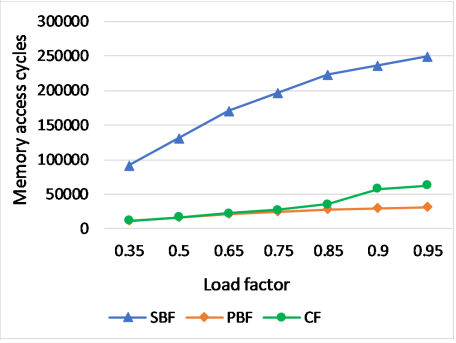
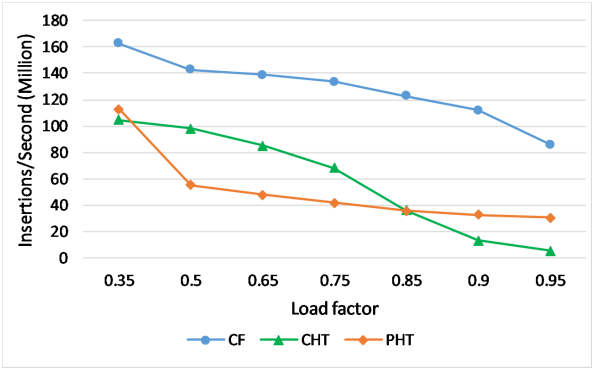


Figure 4.24: Memory access cycles of Cuckoo and Bloom filters

CHT-Cuckoo hash table, PHT-Peacock hash table, SBF-Standard Bloom filter, PBF-Parallel Bloom filter, CF-Cuckoo filter

Insertion throughput: The insertion throughput defines the maximum number of insertions possible per second. The insertion throughput is computed as $\frac{1}{(total\ latency/n))}$, where n is the number of elements inserted, and the latency is measured in nanoseconds. The throughput is constant for Bloom filters since the number of cycles for each insertion is constant. However, the throughput for hashing-based data structures varies with load factors because of the extra hash computation and memory access cycles due to kick/probe operations during a collision. The insertion throughput in terms of million insertions per second of CF, CHT, and PHT are depicted in Figure 4.25.

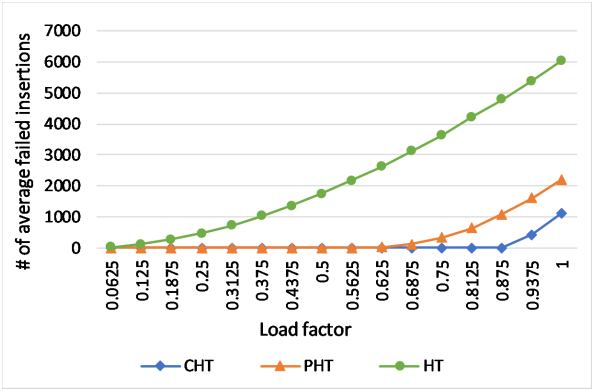


CHT-Cuckoo hash table, PHT-Peacock hash table, CF-Cuckoo filter

Figure 4.25: Insertion throughput

CF exhibits significantly higher throughput than CHT, thanks to the higher operating frequency of CF and significantly lesser number of probe cycles compared to CHT. The amount of probing increases drastically for CHT as the load factor increases. Having four entries in each bucket helps to reduce the number of probes for CF, whereas CHT has only two entries in each bucket, resulting in significantly higher probing cycles than CF. Adding four entries in a bucket can help to minimize the probing cycles for CHT, but a large memory footprint would still limit the operating frequency. The limited memory footprint of CF contributes to minimal routing delay and hence a higher operating frequency. The throughput of PHT is minimum, even for small load factors, due to the lower operating frequency. Nevertheless, the throughput of PHT is almost constant throughout, even for higher load factors, as the total amount of probing per insertion is limited by the number of tables.

Insertion failures: Hash table collisions cause insertion failures while adding elements to the table. Figure 4.26 shows the insertion failures in hashing-based data structures. It can be seen that insertion failures depend greatly on the load factor. No collision-resistant mechanisms such as chaining/probing are employed for the HT. It is evident from the figure that the collisions are maximum for HT, even for lower load factors. When the load factor is 1, the insertion failure for HT is almost 37%. For PHT, the insertion failures gradually increase when the load factor exceeds 0.62. For Cuckoo hashing, insertion failure increases only after a load factor of 0.87, and the overall failures in CHT are considerably lower than PHT. When the load factor is 1, the insertion failure is around 7% for CHT, whereas it is close to 14% for PHT.



CHT-Cuckoo hash table, PHT-Peacock hash table, HT-Hash table

Figure 4.26: Insertion failures in hash tables

Discussion

Lookup architectures: Even though CF has some clear advantages over SBF and most of the other variants of Bloom filter (BF), the achievable parallelism in BF helps to generate better results on hardware. PBF facilitates achieving an update and query memory access latency of a single clock cycle. While BF has no insertion limit, CF can deny an entry if the number of kicks exceeds the maximum kick value. Incrementing the size of the filter and rehashing all the elements (when it comes to Cuckoo hashing) at runtime is a complex task in hardware, especially for high-speed online processing. Duplicate entries have no effect on BF but drastically worsen the load factor, space occupancy, and FPR of CF. The maximum number of duplicates CF can accommodate is $2*b$, where b is the number of entries in a bucket. Trying to insert duplicates after $2*b$ times would result in an infinite kicking loop until the max kick length is reached. In network security applications, an attacker can exploit this vulnerability of CF. Deletion support is a prominent feature of CF that BF cannot provide. CBF can provide deletion support but with much higher memory usage. In conclusion, if the priority is for low-latency, lightweight lookup architecture on hardware with the support for pipelining, PBF is the better choice over CF. CF has a better memory to false positive rate, requiring fewer bits per item to achieve a better FPR. It also requires a very small number of hash bits. For a better bits-per-item with deletion support, where pipelining is not a primary requirement and given that the insertion of duplicates is minimal, CF is the preferred option.

Key-value store architectures: When choosing the best possible key-value store scheme for hardware, PHT may have a slight edge over CHT in terms of

better probing length at higher load factors. However, this advantage is valid only for a low cardinality. Even though the accuracy is comparatively higher for CHT, as observed in Table 4.8, it comes at a high cost of an extremely large amount of probing cycles when $\alpha > 0.5$. Theoretically, the load factor should be less than 0.5 for CHT, but that would cause under-utilization of memory. At the same time, PHT can offer a much smaller maximum probe length with only a small variation in accuracy compared to CHT. However, as cardinality increases, implementing peacock hashing becomes hefty as it has to manage a large number of tables, which results in an increased time complexity and much-reduced operating frequency. Regardless of the increased cardinality, the query time complexity is always constant with CHT, and the operating frequency is also much higher. The update complexity of CHT increases only for a higher load factor. Hence, it can be concluded that PHT is preferable only for a lower cardinality if the primary criterion is minimal probing cycles. On the other hand, CHT can be preferred for any other criteria, assuming that the load factor of CHT is kept small. Moreover, if storing/retrieving keys is not required, there is no better alternative than partial key Cuckoo hashing to store values.

4.4 Conclusion

This chapter discusses the design of fast lookup architectures and a detailed analysis of the state-of-the-art approximate lookup/membership query data structures. For fast lookups in network applications, we targeted the design of novel algorithms and architectures for high-speed Bloom filters on FPGA. We integrated our proposed Xoodoo-NC hash function into Bloom-1 and parallel Bloom filter architectures and evaluated on a Virtex UltraScale+ FPGA of Xilinx. The resulting resource occupation, power consumption, operating frequency, lookup delay, and false positive rate were compared to previously reported Bloom filter architectures and Content addressable Memories on FPGA. To our knowledge, our work significantly outperforms all other solutions on all these comparison criteria. We have also discussed the compliance of the presented work with another protocol stack. Nevertheless, there is always room for improvement. Exploring techniques such as hashing only the individual fields of the flow ID (for example, source address, source port, or protocol field) or masking certain fields could enhance the lookup speed. Such possibilities can be explored in future research.

A thorough analysis of the hardware compatibility of the various existing AMQ schemes and the proposed Bloom filter architectures is performed, and a comparison of these schemes is performed in terms of memory efficiency, accuracy,

latency, implementation complexity, and throughput. We have modified and implemented the hardware-optimized versions of various AMQ data structures, such as Cuckoo hash table and Cuckoo filter, based on their hardware-friendliness for network security applications on high-speed networks. The evaluation results help to identify a suitable data structure for network security applications. Moreover, this analysis also sheds light on the shortcomings of the existing membership query data structures when implemented in hardware, which was unexplored earlier.

Chapter 5

Designing network flow measurement architectures for FPGAs

This chapter discusses the design and implementation of fast membership query/lookup architectures for network security applications, especially large flow detection, on FPGAs. The focus of this chapter is mainly on the hardware-efficient implementations of probabilistic/approximate membership query data structures. This chapter is organized as follows. Section 5.1 introduces the topic and discusses the challenges associated with it. An elaborated description of Count-Min Sketch is presented in Section 5.2. Section 5.3 and Section 5.4 provide the detailed design and analysis of A-CM Sketch and SPArch, respectively. Finally, this chapter is concluded in Section 5.5.

CONTENT SOURCE

- Sateesan, A., Vliegen, J., Scherrer, S., Hsiao, H.C., Perrig, A. and Mentens, N., Speed records in network flow measurement on FPGA. In 2021 31st International Conference on Field-Programmable Logic and Applications (FPL), (pp. 219-224), 2021. (Contributions: Main author)
- Sateesan, A., Vliegen, J., Scherrer, S., Hsiao, H.C., Perrig, A. and Mentens, N., SPArch: A hardware-oriented sketch-based architecture for all-round network flow measurements. (This work may appear in novel publications in the near future) (Contributions: Main author)

5.1 Network flow measurement and its challenges

Network flow measurement, as defined in Section 2.2.3, plays a vital role in large flow detection algorithms. A simple task like counting network packets in high-bandwidth networks becomes challenging when data streams are processed at line rate. Given the real-time nature of network security applications, hardware architectures are indispensable to achieve the throughput requirements of Terabit Ethernet networks. As an example, an authenticated encryption architecture presented in [175] requires a network packet to be processed every clock cycle at a frequency of 200 MHz to achieve a bandwidth of 200 Gbps on a Netcope FPGA platform [185]. Even though existing contributions do achieve success in improving accuracy and speed on software platforms, they often fail to utilize the full speed-up potential offered by hardware platforms. Recent advances in FPGA technology offer large memory bandwidth and logic density, making FPGAs suitable implementation platforms for high-speed network processing applications. Nonetheless, implementing network security algorithms on FPGAs in a Terabit Ethernet network remains a formidable challenge. Therefore, it is important to revisit the design of measurement architectures to effectively harness the parallelization potential and achieve optimal throughput. The primary focus of these design requirements revolves around considerations such as memory requirement, speed, and accuracy.

Processing and storing high-speed data streams requires a large memory footprint. This is because high-speed data streams are characterized by a high volume of data arriving at an extremely rapid rate, requiring faster processing and storage. To fulfill the large storage requirements, most present-day computing systems employ off-chip Double Data Rate Dynamic Random Access Memories (DDR DRAM). However, using external DDR DRAMs slows down the system considerably, making online processing of network traffic impossible. Moreover, the access latency of DDR memory is influenced by various factors, including access patterns, transaction type (read or write), and the DRAM state from previous accesses. Address bus multiplexing and a bi-directional data bus also contribute to the increased access latency. The access latency of DDR3-1600 varies from 10 clock cycles (access request targeting an open row) to 72 clock cycles (access request targeting a conflicting row) for a clock with a 1.5 ns period [127]. On the other hand, the FPGA's on-chip Block RAM (BRAM) has only a single clock cycle of access latency. Constraining the memory overhead within the on-chip memory while keeping the accuracy high is the first step to tackling the challenges associated with the implementation of measurement algorithms on hardware.

Turning away from the conventional exact counting approaches, some of the existing measurement architectures, belonging to various types of detection

algorithms such as intrusion detection, anomaly detection, and heavy-hitter detection, contribute well to minimizing the memory constraints using probabilistic techniques [131, 164, 117, 253, 178, 217]. Nonetheless, such approaches to reducing memory often sacrifice accuracy, introduce higher execution delays, and involve complex arithmetic computations that cannot be efficiently implemented in hardware. Sketches have emerged as the preferred choice for flow measurement architectures, showing the best trade-off between accuracy, speed, and memory utilization. Another concept analogous to sketches is counting bloom filters (CBFs) [153]. However, CBFs primarily focus on determining whether a particular element already exists in a data structure and whether the number of occurrences of this element surpasses a specified threshold.

In this chapter, we focus mainly on sketch-based approaches. Sketches are an integral part of many network security applications, and many of the recently proposed sketch implementations [164, 134, 253, 166] have shown success in flow measurement. However, they often suffer from either high computational/resource costs or low throughput, causing a mismatch in throughput between the line speed and the measurement module. Moreover, the flexibility in terms of individual flow mapping is very much limited for sketches in applications such as large flow detection when compared to associative array counters. The high processing overhead of sketches can be attributed to multiple factors, including independent per-packet hash calculations, memory overhead, or multiple memory accesses and arithmetic operations during the counter update operation [166].

Challenges in existing sketch-based measurement architectures

Versatility in flow measurement: Probabilistic measurement data structures may not be suitable for all detection algorithms due to their lack of adherence to a standardized measurement mechanism. For instance, large flow detection algorithms such as EARDet[241] use eviction-based measurements, in which the flows are evicted when the flow sizes fall below a certain threshold. Such eviction-based algorithms require a one-to-one mapping of the flows to the counter. On the other hand, large flow detection algorithms such as LOFT [217] use sketch-based measurements. Additionally, mapping flows of different sizes to buckets of varying sizes is an ideal solution to reduce the memory footprint while improving the relative accuracy [100]. This is because the majority of the incoming flows are mouse flows that require smaller counter sizes. Despite efforts by architectures such as Elastic Sketch [253] and Count-less Sketch[147] to address this problem of mapping different-sized flows to different-sized buckets, these algorithms do

not tackle eviction-based measurements. This also demonstrates the reduced flexibility of probabilistic/sketch-based measurements.

When it comes to mapping a large number of network flows to corresponding buckets (one-to-one mapping) to assist eviction-based measurements, a counter array is the most suitable architecture. The prevalent approach in hardware architectures involves the use of associative memories, such as CAM, to implement these counter arrays. Associative memory-based counter arrays offer support for various detection algorithms, including eviction-based algorithms, thus making them highly flexible for diverse applications. Figure 5.1 illustrates a representation of a counter array based on a CAM. Not only are there downsides to using CAM, such as slower write times, lower operating frequency, larger memory footprint, and high power consumption, but creating a counter array with CAM also requires additional computational logic. As shown in the figure, an address to one-hot encoder is required to generate *enable* signals for counters. This address to one-hot encoder is translated to a large multiplexer on hardware. The drawbacks associated with associative array/CAM memories compel the development of a more hardware-efficient alternative that offers comparable versatility. An architecture that offers the memory efficiency of sketches while offering one-to-one mapping is the need of the hour.

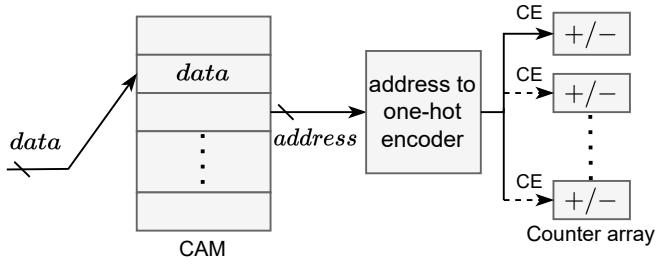


Figure 5.1: CAM-based counter array

Accuracy and memory-accuray trade-off: When using probabilistic architectures such as sketches, boosting accuracy necessitates allocating more memory. A higher amount of memory has to be allocated to sketches to achieve higher accuracy, resulting in a large percentage of the memory being unused (underutilization of memory). The reason for underutilization is that the larger memory size leads to fewer hash collisions. Given fewer hash collisions, the counter values will be lower thanks to reduced counter-sharing, leading to unused higher-order bits in many counters, thereby underutilizing memory. A Count-Min Sketch (refer Section 5.2) of size (d, w) maps each flow to d counters and can simultaneously handle w flows. This, in fact, can lead to underutilization

of the allocated resources as it requires $d \times size\ of\ the\ counter$ bits to store a single flow. Although sketches work on the principle of memory sharing, accuracy decreases as the number of elements exceeds w, as demonstrated in Section 5.4.3. Taking a real-world example for CM Sketch: for a switch having a total bandwidth of 1 Tbps and that can handle 10 million flows, limiting the overestimation within 50% of the average-flow size would require approximately 250 million counters for the CM Sketch [217]. Furthermore, the amount of memory required would be much higher than mapping a flow to a single counter, which exemplifies the limitations of sketches on high-capacity routers. On hardware, large memory requirements affect not only the resource constraints but also the operating speed. Larger memory units mean more routing delays and lower operating speeds. Moreover, if the on-chip memory does not suffice, additional external memory, with a much larger access time, needs to be used.

Numerous approaches have been proposed to improve the accuracy of sketches in network flow measurement applications, specifically in heavy-hitter detection [251, 259, 207, 250, 258, 253, 225]. While these approaches have shown improvements in accuracy, the use of additional data structures [207, 250, 258] or shared counters [251, 259, 250] increase the overall storage requirements. The increase in memory requirements is significant, considering the limited on-chip memory and the effect on operating speed and bandwidth requirements. One specific approach, Augmented Sketch [207], employs an associated filter with a CM Sketch, and the filter stores the elephant flows along with the key, and the corresponding flow is sent to the sketch if the filter is full. If any flow count in the sketch exceeds the smallest value in the filter, it is exchanged. Although Augmented Sketch provides accurate measurements for elephant flows, it does so at the expense of higher memory requirements and more memory accesses.

Elastic Sketch [253] is a well-known algorithm for heavy-hitter detection, heavy change detection, and cardinality estimation. Section 5.4.1 provides a brief description of Elastic Sketch. Elastic Sketch has a *heavy part* for recording elephant flows and a light part for recording mouse flows. The light part is a simple CM Sketch. Elastic Sketch requires different hash functions for each stage of the *heavy part* and the CM Sketch, which increases the computational complexity. Such a structure would keep the accuracy high, which is the primary goal of Elastic Sketch, but at the cost of increased latency and computational overhead. The hardware version of Elastic Sketch has multiple *heavy parts* operated sequentially, leading to increased latency. Moreover, the update and query operations require memory accesses equal to the sum of heavy and light parts in the worst-case scenario. The operational frequency of Elastic Sketch on hardware is also low [253], limiting the throughput. Elastic Sketch can be advantageous when the packet sizes are taken as '1' (flow size counting). However, it becomes less suitable for flow byte counting as the packet sizes can

be up to 64KB if we consider jumbo frames, making the 8-bit counter size in the light part of Elastic Sketch inadequate.

SF Sketch[251] also uses distinct sub-sketches for mouse and elephant flows, necessitating additional memory to enhance accuracy, which is undesirable. Diamond Sketch [250] and One Memory Access Sketch[259] use multiple levels of smaller sketches. As the lower level counters overflow for an incoming flow, that flow is moved to the next higher level. Diamond Sketch incorporates additional data structures like *deletion* and *carry* parts, and both Diamond Sketch and One Memory Access Sketch involve the storage of the flow fingerprints for improved accuracy. Consequently, the memory demand is high for both of these approaches. Furthermore, the worst-case number of memory accesses is proportional to the number of levels, adversely affecting the speed and making pipelining difficult on hardware. Bloom Sketch [258] and SA Sketch [261] introduce a combination of Bloom filter and sketch-based approach, and both employ multiple layers of sketches. Such multi-layer architectures would adversely affect the worst-case memory access time, operating speed, and memory utilization.

Existing sketch-based architectures with improved accuracy

In addition to sketches, there exist data structures that are based on sketch architectures and achieve higher accuracy than sketches. One such example is HeavyKeeper, which is briefly described in Section 5.4.1. The HeavyKeeper algorithm is used for measuring top-k elephant flows and incorporates fingerprint along with the count in each bucket to improve the accuracy. HeavyKeeper assumes minimal fingerprint collisions, and to minimize fingerprint collisions, the sizes of the fingerprints should not be too small. Storing a large fingerprint along with the count causes a significant increase in memory usage. Despite HeavyKeeper not storing all flows like the CM Sketch, the inclusion of a fingerprint makes the memory usage more or less comparable. Moreover, if the fingerprints do not match, HeavyKeeper applies exponential decay, introducing the possibility that an incoming large flow may collide with an existing large flow. Another drawback with sketch/sketch-based algorithms, such as HeavyKeeper and Elastic Sketch, is that they are not designed for flow byte counting but for flow size counting.

Recently introduced algorithms such as LOFT [217] and ALBUS [216] take memory reduction as their primary design criteria while achieving a high detection accuracy against specific attacks like large flow and pulsating attacks. Both these algorithms employ probabilistic measurement approaches while accepting hash collisions. These techniques reduce the overall memory

requirement. However, the additional logic employed to compensate for the hash collisions does not go well on hardware, even though the performance is excellent on software. The introduction of these algorithms also puts forward the interesting scenario of combining multiple algorithms that target distinct attacks to detect various types of attacks. Such a scenario may require a common measurement module that can handle associative array-based (or eviction-based) and probabilistic flow measurements.

Conclusions from the related work

An overview of different approaches along with our proposed approach, SPArch, is presented in Table 5.1.

Table 5.1: Comparison of counting architectures

Metrics	Type of data structure			
	Exact counting	Sampling-based	Sketch-based	SPArch
Measurement error	No error	Very high	Low to moderate	Very low
Memory overhead	Very high	Moderate to high	Low	Low
Latency on hardware (in ns)	Low to very high	Moderate to very high	Very low to moderate	Very low
One-to-one mapping	Yes	Possible	Not possible	Possible
Deletion support	Possible	Possible	Not possible	Possible
Computational complexity	Moderate to high	High to very high	Low to moderate	Low to moderate
Hardware friendliness	Low	Very low	Moderate to high	Very high

It is evident from related work that sketches face the challenge of striking a poor trade-off between memory usage and accuracy. Moreover, most of the sketch-based algorithms focusing on improving the detection accuracy would require a large memory overhead, which also results in the underutilization of memory due to fewer collisions and less sharing of memory. Moreover, they lack the versatility of individual flow mapping provided by architectures like associative array-based counters.

Associative array counters require a considerable amount of memory despite having high accuracy, making them more suitable for offline analysis. On the other hand, associative arrays give the flexibility to organize the counters into multiple buckets (of different counter sizes if required). In real networks, the majority of the flows are mouse flows, and elephant flows are rare [67]. Hence, keeping equally-sized counters for both elephant and mouse flows is not wise. An efficient way to handle this is by allocating separate counter arrays to store them. Some recent works exploit this technique [253, 251], but such data structures demand more memory overhead, partially negating the benefit of having separate buckets. Knowing that only a small percentage of flows would require a large counter, a one-to-one mapping architecture could be space-saving as it is easier to map the flows into appropriately sized buckets. This also gives

the freedom to choose different scaling factors for different buckets for applying approximation, and this could bring in a massive reduction in memory overhead.

In this chapter, we address the challenges associated with existing flow measurement solutions concerning their hardware-friendliness, underutilization of memory, restricting the overall memory usage to on-chip memory, and assigning counter sizes based on flow sizes. We also seek to enhance the versatility and accuracy of probabilistic data structures comparable to associative arrays. Furthermore, we aim to meet the need for a common measurement unit for both eviction-based and sketch-based detection algorithms. This work focuses on measurement architectures that are hardware-friendly and are capable of processing the incoming network traffic at line rate on Terabit Ethernet networks. These architectures are specifically designed for network security applications, such as DDoS attack detection, where the flow measurement is bound to a predetermined measurement epoch. This chapter proposes two sketch-based architectures, namely A-CM Sketch and SPArch, to counter the drawbacks of sketch-based flow measurement and improve hardware efficiency. Both A-CM Sketch and SPArch aim to reduce the memory footprint and improve the accuracy and throughput, specifically targeting high-speed networks. Moreover, SPAarch accomplishes one-to-one mapping while attaining better memory efficiency than sketches. Further elaboration on these approaches will be presented in the upcoming sections.

5.2 Count-Min Sketch

Our novel flow measurement architectures, A-CM Sketch and SPArch, draw inspiration from the Count-Min (CM) sketch, proposed by Cormode and Muthukrishnan [88]. This section provides a comprehensive overview of the CM Sketch.

CM Sketch, is a 2-dimensional array of counters having width w and depth d, as shown in Figure 5.2. CM Sketch is particularly suitable for FPGA-based flow measurement architectures. This suitability stems from its simple architecture, ease of parallel implementation, relatively high throughput, and small memory usage of $O(logn)$, where n is the number of distinct flows. Each cell in the 2-dimensional array of CM Sketch is a counter. The size of each counter is q bits. A set of independent hash functions are used to index the counters. The counters in CM Sketch are shared, and multiple flow IDs can be mapped to the same counter, which may result in an overestimation of the count. However, the fact that underestimation will never happen is advantageous for large flow

detection applications, which aim to detect whether a network flow occupies more than its allowed bandwidth.

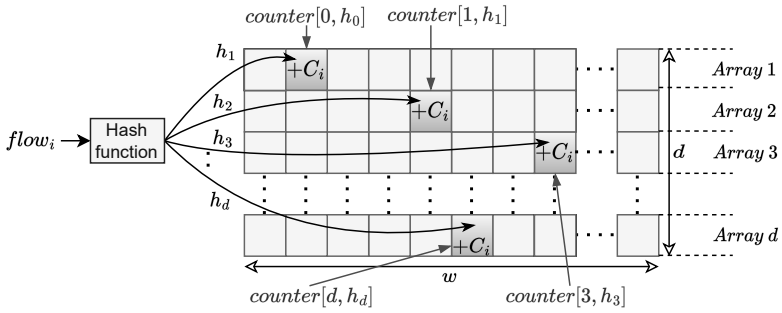


Figure 5.2: Count-Min Sketch

When a network packet arrives, characterized by a flow ID $flow_i$ and a flow size C_i, d independent hash functions are computed on $flow_i$. Each hash value $h_j,\ j = \{1, ..., d\}$ is used as an index to determine which counter should be updated/queried in the corresponding counter array row j. The update and query operations in CM Sketch are detailed below.

Update operation: The update operation for each counter is performed as follows: an incoming flow ID $flow_i$ is mapped to d counters, one counter in each array, using d pairwise independent hash functions h_j. The d counters is updated/incremented with count value C_i as $counter[j, h_j] \leftarrow counter[j, h_j] + C_i$.

Query operation: While querying the CM Sketch, the incoming flow ID $flow_i$ is hashed and mapped to d counters in the d arrays. The count values of each hash-indexed counter are read, and the minimum of all d counter values is taken as the output. The estimated count can be determined as $min(counter[j, h_j], \forall j \in 1..d)$.

As explained by Cormode and Muthukrishnan [88], the width and depth are determined by two parameters, namely the error factor (ϵ) and the error probability (δ), which are defined as follows: the additive error in answering a query is $\epsilon \times a_i$ with a bound probability $1 - \delta$, where a_i is the actual count value. Given the parameters ϵ and δ, the dimensions of w and d can be calculated as $w = \lceil \frac{e}{\epsilon} \rceil$ (with e representing Euler's number) and $d = \lceil \ln \frac{1}{\delta} \rceil$. The accuracy of the algorithm is thus determined by ϵ and δ.

5.3 Design and analysis of A-CM Skech

The accuracy of probabilistic architectures exhibits a direct correlation with the amount of allocated memory. Due to constraints in fast on-chip memory and computational speed, it is not feasible to increase the memory footprint when designing network security algorithms for hardware in high-speed networks. Simultaneously, sacrificing accuracy in favor of memory optimization would be unwise for applications such as large flow detection, as erroneous detections could result in catastrophic consequences. To tackle the challenge of memory requirement, throughput, and accuracy, we propose the Approximate Count-Min Sketch or A-CM Sketch, a novel variant of the Count-Min Sketch algorithm that uses less memory and offers higher throughput compared to other FPGA-based sketch implementations. To reach the throughput requirements of Terabit Ethernet networks and minimize the memory footprint on FPGAs, we replace the exact counters with approximate counters in the sketch.

A-CM Sketch relies on optimizations in two areas:

(1) it employs approximate counters. The newly proposed Hardware-oriented Simple Active Counter (HSAC) algorithm is employed to efficiently implement these approximate counters in hardware;

(2) it uses an optimal distribution of the embedded memory, optimized towards maximum operating frequency.

We focus on the following challenges in the design of A-CM Sketch: (1) minimize the memory and arithmetic overhead of the sketch architecture to fit all storage and computation constraints in a single FPGA, and (2) maximize the throughput to operate in Terabit Ethernet networks. Therefore, we introduce the following contributions:

- A modified and hardware-friendly version of the Simple Active Counter (SAC) algorithm [223], which we call Hardware-oriented Simple Active Counter (HSAC), and a counter array based on HSAC.
- A novel sketch architecture, termed Approximate-CM Sketch or A-CM Sketch, inspired by the Count-Min Sketch (CM Sketch) algorithm [88]. A-CM Sketch uses approximate counters based on the HSAC algorithm and is optimized for high throughput by minimizing the routing delay in the memory architecture and utilizing hardware-friendly hash computations, resulting in the first 200 Gbps flow measurement architecture on FPGA.
- A demonstrator on an FPGA in which the sketch hardware architecture has a direct Ethernet connection.

Our work is the first to achieve a throughput of 200 Gbps for 96-bit flow IDs, while the existing hardware architectures (at the time of publication) are restricted to 32-bit flow IDs and do not reach 200 Gbps.

5.3.1 Background of approximate counters

Counters are an indispensable part of any statistical measurement. Approximate or probabilistic counting [184, 110] emerges as a viable solution to minimize the counting overhead while incurring only a marginal loss in accuracy. This loss in accuracy becomes inconsequential when dealing with substantial data volumes and count values that are significantly larger. Robert Morris, in 1978 [184], proposed a probabilistic counting technique to reduce the memory requirements of counters. The basic idea behind this algorithm is that the counter only stores an "order of magnitude approximation" of the actual count. The approximate counter value could be, e.g., a power of two, with only the exponent stored in memory. Whenever an increment is supposed to occur, the approximate count value is incremented with a probability that is proportional to the number of actual count values that are present between the current and the next approximate count value. For example, suppose the current approximate count value is 8 (i.e., the stored exponent is equal to 3). In that case, each new incoming event that should update the counter will only be considered in 1/8 of the cases, as there are eight actual count values between the current approximate count value (i.e., 8, with stored exponent 3) and the next one (i.e., 16, with stored exponent 4). A pseudo-random number is used to determine when the increment should happen.

Following the approach by Robert Morris, many improved probabilistic counting techniques [223, 132, 131, 99, 162, 252, 62] have been put forward. The primary emphasis of the majority of such works has been enhancing accuracy. Algorithms such as CASE [162], DISCO [131], ICE Buckets [99], and additive error counters [62] exhibit high accuracy. Nevertheless, these algorithms are not suited for efficient hardware implementation. Yang et al. [252] present a sketch design using approximate counter arrays, which shows a significant reduction in memory compared to sketches using exact counter arrays. However, the self-adapting counter approach by Yang et al. is a slower and more complex version of the SAC [223] algorithm. Hence, it is unsuitable for hardware implementations in high-speed networks. A more detailed discussion on related works in sketch-based network flow measurement on FPGAs is presented in Section 2.6. In our work, we introduce the Hardware-oriented Simple Active Counter (HSAC) algorithm, a hardware-friendly version of the SAC algorithm. The proposed A-CM Sketch combines the CM Sketch [88] with HSAC approximate counters.

5.3.2 Algorithms

Our novel sketch architecture, which we call Approximate CM Sketch or A-CM Sketch, is based on the Count-Min (CM) sketch data structure [88] and on the Hardware-oriented Simple Active Counter (HSAC) approximate counting algorithm. The CM Sketch is presented in Section 5.2. Our newly proposed HSAC algorithm and the hash function employed in the implementation of CM Sketch are explained in the subsequent sections.

Hardware-oriented Simple Active Counter (HSAC)

We introduce a hardware-friendly approximate counting algorithm - Hardware-oriented Simple Active Counter (HSAC), which is a modified version of the Simple Active Counter (SAC) algorithm [223]. The original SAC algorithm divides the counter-memory of q bits into exponent and estimation parts, namely *exp* and *count* of sizes l bits and k bits, respectively. The counter is updated with a random probability v and counter is incremented when $v < (inc/2^{scale \times exp} - \lfloor inc/2^{scale \times exp} \rfloor)$, where inc is the flow size. Re-normalization with a global scaling parameter *scale* is applied when the counter overflows. The estimated value can be computed as $count \times 2^{scale \times exp}$. Although SAC can record large measurements with limited memory consumption, the presence of floating point divisions and multiplications and the use of a single scaling factor for all the counters in an array make it less favorable for high-throughput hardware implementations.

In HSAC, a fixed number of bits q is allocated in memory for each counter, similar to SAC. However, HSAC introduces a distinction from SAC by incorporating the scaling parameter *scale* in q and is specific for each counter. Therefore q is partitioned into an exponent part *exp*, an estimation part *count*, and a scaling part *scale* of sizes l bits, k bits, and m bits, respectively. This way, HSAC avoids the unnecessary re-normalization of all counters when only a single counter overflows. This results in better accuracy and lower computational overhead but at the cost of slightly higher memory usage. Nevertheless, in the remainder of this section, we show that the modest increase in memory usage does not significantly contribute to the overall memory requirement of the sketch. The pseudocode of HSAC for a single counter is shown in Algorithm 2.

The SAC algorithm possesses certain flaws for smaller values of l, which refers to the number of bits in the exponent part *exp*. To illustrate this, consider $l = 2$, the current exponent value $exp = 3$, and let the count be $count = 4$. At the next increment, the exponent becomes $exp = 2^l$, requiring re-normalization. The new value of *exp* is changed to $\lceil exp \times (\frac{count}{count+1}) \rceil$, which in fact remains

Algorithm 2: Pseudocode of HSAC for a single counter

```
1  Parameters: size of the pseudo-random number - S_p, size of scale (bits) - m, size of
     exp (bits) - l, size of count (bits) - k
2  InitializeVariables()
3      scale = 0, exp = 0, count = 0
4  UpdateCounter(inc)
5      F_1 = exp ≪ (1 ≪ (scale − 1))
6      count = count + inc ≫ F_1
7      p = (inc & ((1 ≪ F_1) − 1)) ≪ (S_p − F_1)
8      v = rand()
9      if v > p
10         count = count + 1
11     endif
12     if count > (1 ≪ k) − 1
13         exp = exp + 1
14         F_2 = (1 ≪ (1 ≪ scale)) − 1
15         p = count & F_2 ≪ (S_p − scale)
16         v = rand()
17         count = count ≫ (1 ≪ scale)
18         if v > p
19             count = count + 1
20         endif
21     endif
22     if exp == 1 ≪ l
23         RenormalizeCounter(scale)
24     endif
25 RenormalizeCounter(scale)
26     exp = exp ≫ 1
27     scale = scale + 1
```

4, and the algorithm collapses. HSAC eliminates these flaws. Unlike in SAC, the values of *scale* in HSAC are powers of 2. This eliminates the complexity of the re-normalization process in the algorithm. The multiplication and division operations are then replaced with shift operations which come for free in hardware. The number of bits required to store *scale* is also reduced in HSAC by modifying the equation for estimation as in Eq. (5.1).

$$V = count \times 2^{(2^{scale} \times exp)} \tag{5.1}$$

HSAC supports both flow size and flow byte counting. For flow byte counting, it is necessary to keep the size allocated for *count* greater than the maximal incoming packet size. Considering standard frame sizes of 1500 bytes, the size of *count* is kept as 11 bits, whereas no such size restrictions apply for flow size counting. Nonetheless, it is better to keep the size of *count* as large as possible to avoid loss in accuracy. In SAC, re-normalization is required for all counters each time the value of *scale* is updated. This could suspend the update process

and result in the loss of packets. In HSAC, in order to eliminate this issue and to reduce the processing overhead, *scale* is unique for each counter. The use of separate scaling parameters is similar to ICE Buckets [101], but in our HSAC architecture, each counter is considered a bucket, while ICE Buckets uses buckets with multiple counters sharing the same scaling factor for the whole bucket.

Keeping *scale* unique for each counter helps maintain the accuracy and keeps the relative error in the measurement to a minimum. In the case where *scale* is shared among all the counters in an array, as in SAC, a single flow with a large volume can cause a large relative error to all other small flows in the counters. Keeping the maximum value of 2^{scale} to 4 is optimal to limit the error to a minimum. Hence, only 2 bits are required to represent *scale*, which of course, can support a value of 2^{scale} up to 8. (The maximum value that can be represented using 2 bits is 3, leading to a maximum value of $2^{scale} = 2^3$.)

Hash function

An important bottleneck in the implementation of sketch algorithms is the speed of the hash computation. As discussed in Chapter 3, nearly all of the existing non-cryptographic hash functions are not so efficient in providing high operating frequency in hardware when the hash output size is large. Our proposed non-cryptographic hash function based on the Xoodoo permutation, Xoodoo-NC, is used for our hash computations. The design of Xoodoo-NC is detailed in Section 3.3. Most of the existing work on network flow measurement [228, 209, 257, 158, 230] focus on employing universal hash functions for hardware such as H3 hashing [201]. This would increase the latency and hence pipelining would be required to improve the throughput [228, 158, 257]. Independant hash computations in parallel would keep the latency to 1 cycle but at a moderately high computation overhead and reduction in operating frequency [257]. Our implementation keeps the hash computation latency to 1 cycle with low computation overhead leading to a total sketch update latency of 4 clock cycles. This is significantly less compared to other implementations where the use of universal hash functions affects the hash computation latency of the overall update latency as shown in the works [209]-25 cycles, and [158]-21 cycles.

5.3.3 Hardware architecture

CM Sketch is a hardware-friendly data structure, but there are multiple bottlenecks in the hardware implementation, such as multiple independent

hash computations, multiple memory accesses for updating a packet, and arithmetic computations for counter updates. The most cumbersome task in the implementation is to keep the routing delays to a minimum to improve the overall clock frequency to assist in processing the incoming packets at line rate. In this work, a combination is proposed of both the CM Sketch and HSAC with the intention of combining the best of both. From the CM Sketch, the benefits of the hardware-friendly nature for large flow measurement are gained, while HSAC helps to maintain a small memory footprint and feasibility for high-speed hardware. As explained above, the hash functions of the CM Sketch can be substituted with a single Xoodoo-NC implementation.

A high-level view of the A-CM Sketch data structure that we propose is shown in Figure 5.3. The counter array is a 2-dimensional array of size $w \times d$. Each element in the array is a counter of fixed size q. When a network packet enters the architecture, the extracted flow ID is hashed using the Xoodoo-NC algorithm [212], which calculates the hash output in one clock cycle. The output of Xoodoo-NC is split into d hash values of size $log_2 w$ to index the counters. The indexed counters are then updated. Instead of storing the exact value of each counter, the approximated value obtained from HSAC is stored.

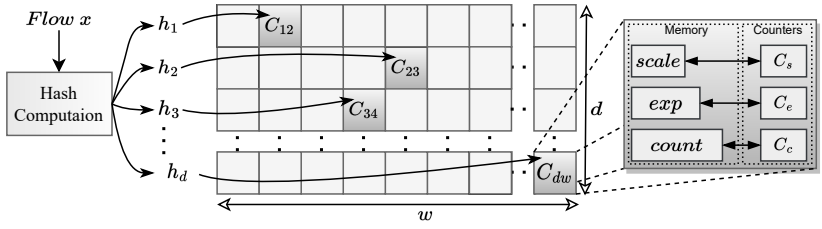


Figure 5.3: Data structure of the Approximate-CM Sketch (A-CM Sketch)

Update process

Upon the arrival of a flow x carrying a volume c, one counter, denoted as C_{dw}, in each of the d rows is updated by c, where d and w are the dimensions of the sketch. The specific counters to be updated are determined by the hash value h_j, where $1 \leq j \leq d$. The memory location of the counter to be updated is indexed by the hash value. The existing value in the counter is read, and the packet volume c is added. The updated volume is written back to the memory. All the memory read and write operations are performed in parallel, ensuring that each read and write operation takes only one clock cycle. One clock cycle each is required for hash computation, memory read, addition, and memory

write operations. To increase the operating frequency, a buffer stage is inserted between the memory read and addition stage. The latency in terms of the number of clock cycles for each update operation is five. All the stages are pipelined so that the sketch unit can receive and process a 96-bit flow ID in every clock cycle at line rate.

Query process

Each counter value consists of a fixed number of bits, which is the concatenation of *scale*, *exp*, and *count*. The most significant m bits, either 1 or 2 bits, are used for *scale*, the next l bits are used for *exp*, and the least significant k bits are used for *count*. Similar to the update process, the hash computation and memory read operations are performed, and the values of *scale*, *exp*, and *count* are read from memory. These values are concatenated to obtain the count values for each of the d counters C_{dw}. The minimum of d counter values is computed and taken as the output. The estimated value is computed using Eq. (5.1) from the minimum of all the d values. The querying process has a latency of 3 clock cycles.

To quantify our efforts, an implementation is made on a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVC2104-1-E). The memory array for the counters uses only on-chip dual-port BRAM, which is sufficient to satisfy the memory requirements of our architecture. However, a substantial number of BRAM blocks are necessary for the implementation. This reflects in an increased placement and routing complexity when the memory and logic resources are spread over multiple clock regions, which, without proper consideration, leads to a decrease in the operating frequency. We employ efficient design approaches to mitigate this issue, as described in the following paragraphs.

In order to tackle the issues caused by routing delays, the counter array is organized into smaller memory units. Since the BRAM slots in FPGAs are fixed, building up larger memories in an automated way with the design tools reduces the number of BRAM blocks used, but significantly increases the routing delay. Therefore, instead of using one single large memory block for representing a counter, we employ separate memory units for each *scale*, *exp*, and *count* value, as shown in Figure 5.4. The figure depicts only one of the $w.d$ counters. Furthermore, we direct the BRAM concatenation done by the design tools to employ fixed primitives, and we choose an optimal primitive size through trial and error. Using fixed primitives somewhat increases the BRAM utilization but helps in reducing the routing delay. During operation, these memories are read first before updating. All memory accesses are performed in parallel, which limits the total memory access delay to only two cycles for every update, one

for the memory read and one for the memory write operation. Every *scale*, *exp*, and *count* calculation has its dedicated physical counter C_s, C_e, and C_c, respectively, i.e., the total number of counters used in the whole implementation is equal to $d.C_s + d.C_e + d.C_c$. These counters are reused in every update cycle.

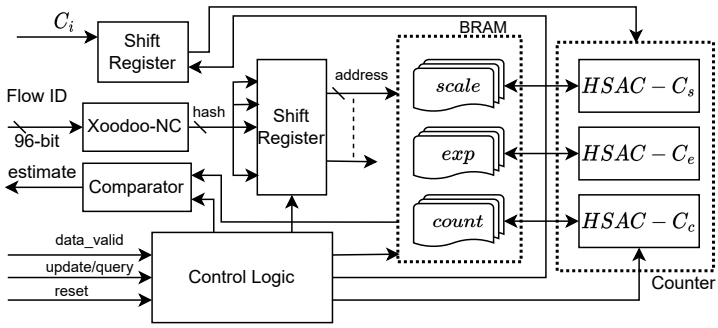


Figure 5.4: Hardware architecture of the approximate-CM Sketch

HSAC implementation

The core component of the sketch is the approximate counter HSAC. The HSAC block consists of an update, a re-normalization, and a Linear Feedback Shift Register (LFSR) module. The update and re-normalization processes are discussed in Section 5.3.2. An S_p-bit LFSR with XOR feedback is used as the pseudo-random number generator. The output size of the LFSR is equal to $2^m.(2^l - 1)$ bits ($S_p = 2^m.(2^l - 1)$) to avoid any loss in accuracy, where m and l are the number of bits for *scale* and *exp*, respectively.

5.3.4 Evaluation and Results

We present the results of three experiments. The first experiment analyzes the estimation error and the performance of HSAC in comparison to SAC. In the second experiment, the functionality of the A-CM Sketch architecture is tested on a lab setup with network traffic applied to an FPGA board that hosts the sketch architecture. In the third experiment, the performance and resource utilization of the hardware architecture is evaluated using FPGA design tools. The values of d and w for the A-CM Sketch are chosen as 4 and 2^{16}, corresponding to an error factor $\epsilon = 4.147\text{x}10^{-5}$ and error probability $\delta = 0.0183$. Each HSAC counter has a width of 16 bits, where $l = 4$ and $k = 11$, and can count up to $2.197\text{x}10^{12}$.

Analysis of HSAC

We examine the error rate of HSAC in comparison with SAC in terms of the maximum possible estimation error and mean estimation error for a single counter, as shown in Figure 5.5 (a) and (b). For a corresponding counter array, the error will be higher for SAC because the re-normalization will be applied to all the counters irrespective of the counter values, whereas the error remains constant for HSAC as only the overflown counter will be re-normalized. For a 16-bit counter (l=4, k=11, m=1) with a latency of one clock cycle, HSAC occupies 129 LUTs and 38 FFs at 372 MHz, compared to 153 LUTs and 46 FFs at 198 MHz for SAC. The maximum estimation error is reduced remarkably with increasing *count* bits (k) as shown in Figure 5.5 (c). The measurement error of HSAC is quite small and thus does not introduce the danger of false negatives for A-CM Sketch. For a count of 1×10^6, when k increases from 11 to 14 bits, the worst-case estimation error reduces from 6.4% to <1%, with the mean error being very close to zero.

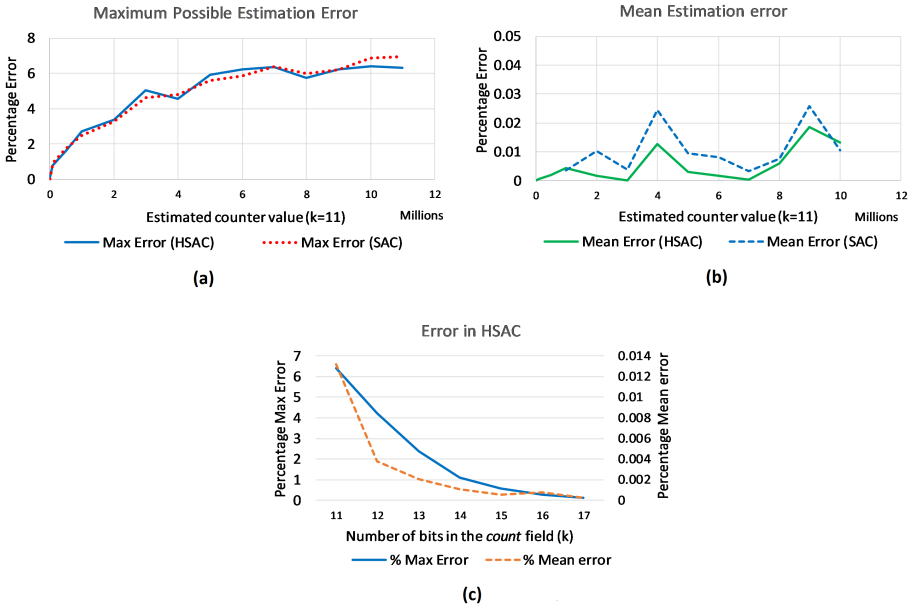


Figure 5.5: (a) Maximum possible estimation error for HSAC vs. SAC; (b) Mean estimation error for HSAC vs. SAC; (c) Maximum and mean estimation error for HSAC with a varying number of *count* bits (k)

Functional test of A-CM Sketch

The functionality of the proposed A-CM Sketch implementation is tested on a lab setup. This setup consists of a Xilinx VC707 board (hosting a Virtex-7 FPGA), which is attached to a Gigabit network switch. Traffic to and from the FPGA is generated through a Python program. In the System-On-Chip, shown in Figure 5.6, the CM Sketch component has direct connections for receiving and sending frames in the network. It is connected through AXI4S with the Ethernet core. After filtering out UDP and TCP frames with the help of a parser, the network packets are processed in the sketch at line rate. In order to test the functionality of the sketch, a MicroBlaze softcore processor queries a specific flow by sending the flow ID to the sketch, which then returns the estimated count. This communication happens over an AXI4 bus, which is also connected to a Timer and a UART. The UART is used for debugging purposes. The functionality test is not performed at maximum throughput (200 Gbps), but on a 1 Gbps lab setup. Table 5.2 shows the implementation results on the VC707, using Vivado 2017.4. The required additional glue logic, mainly for clock domain crossing, is represented under the "with wrapper" component.

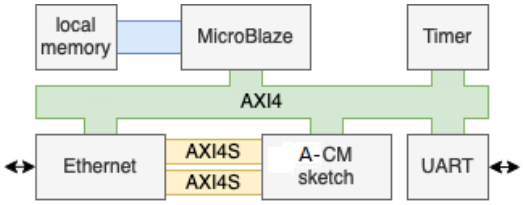


Figure 5.6: System on chip for functional test

Table 5.2: Implementation results of the SOC on the Virtex-7 FPGA

Component	Slices	BRAM
Complete SOC	3'358	153
Sketch estimator with wrapper	1'271	120
Sketch estimator without wrapper	1'090	120
Total available	75'900	1030

Implementation results of A-CM Sketch

In order to evaluate the performance, we use the Vivado 2017.4 design tool to generate implementation results for a Xilinx Virtex Ultrascale+ FPGA (XCVU7P-FLVC2104-1-E).

Resource utilization and performance: By replacing exact counters with approximate counters, the memory footprint of the CM Sketch significantly decreases. For A-CM Sketch with $d = 4$ and $w = 2^{16}$, using 16-bit approximate counters, the total memory requirement is 512 KB. In contrast, the use of exact counting requires 41 bits to represent the same counter value, resulting in a memory requirement of 1312 KB (on a similar hardware architecture, with the sole difference being the counters). Moreover, the frequency of operation is significantly improved, as demonstrated in Table 5.3. The exact-counter-based implementation utilizes $\approx 2.2\times$ more BRAM while having a latency that is $\approx 1.8\times$ higher than the approximate-counter-based implementation. The difference in accuracy is very much negligible when compared to the corresponding CM Sketch implementation. The mean and maximum accuracy variation of A-CM Sketch versus a 24-bit CM Sketch implementation almost touches zero when the number of counter field bits k is greater than or equal to 10, as shown in Figure 5.7.

Table 5.3: Comparison of approximate-CM Sketch with exact (full-precision) counter based CM Sketch

	w	LUT	FF	BRAM	Operating Frequency	Update Latency
CM Sketch	65536	1376	430	352	123.28 MHz	40.56 ns
	32768	1214	406	176	135.01 MHz	37.04 ns
A-CM Sketch	65536	1657	757	144	202.26 MHz	24.72 ns
	32768	1399	729	72	236.00 MHz	21.19 ns

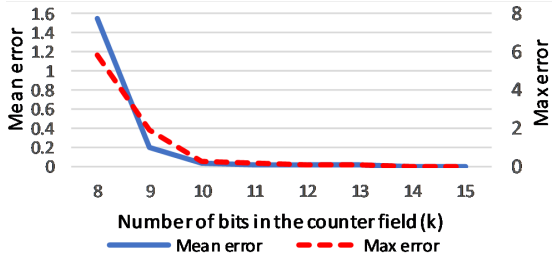


Figure 5.7: Accuracy variation of A-CM Sketch compared to CM Sketch

To analyze the efficacy of the memory optimization techniques introduced in Section 5.3.3, we compare the A-CM Sketch with two other implementations - a naive A-CM Sketch implementation (without the memory optimization) and an exact-counter-based CM Sketch. The results, as shown in Figure 5.8, visualize the improvement in the operating frequency of the optimized A-CM Sketch implementation compared to the naive A-CM Sketch. The naive A-CM Sketch

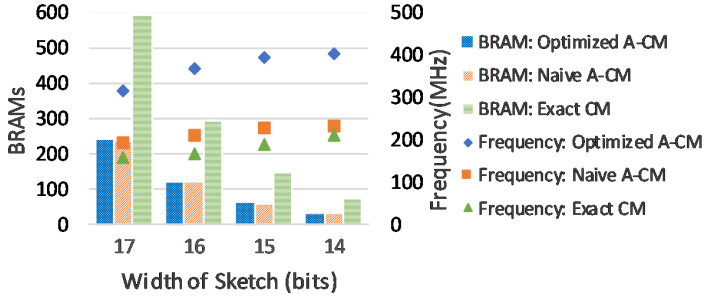


Figure 5.8: BRAM utilization and frequency as a function of the sketch width for different implementations

implementation consumes slightly fewer BRAMs compared to the optimized version. The exact-counter-based CM Sketch implementation uses more than twice the amount of BRAM compared to the other implementations. Figure 5.9 shows the floorplan of BRAMs (highlighted in cyan) after placement & routing. It is evident that the optimizations helped in compactly arranging the BRAMs closer to the main logic.

Nevertheless, the applied optimization techniques for improving the speed retaliate when the counter sizes are very small. The techniques for minimizing the routing delay in A-CM Sketch increase the resource and memory utilization percentage slightly compared to the actual logic resource requirements with the naive A-CM Sketch implementation. The difference in resources and latency for the sketches with counter parameters $m = 1$, $l = 2$, and $k = 8$ is shown in Table 5.4. The delay-optimized A-CM Sketch provides a much higher operating frequency but consumes more resources. Hence, the naive A-CM Sketch implementation is preferred if it meets the frequency requirements when the counter sizes are very small.

Table 5.4: Optimized versus Naive A-CM Sketch implementation for smaller parameter sizes

	w	LUT	FF	BRAM	Operating Frequency	Update Latency
Optimized	16384	704	585	28	329.92 MHz	15.15 ns
	8192	665	569	14	325.94 MHz	15.34 ns
Naive	16384	636	272	24	220.41 MHz	22.68 ns
	8192	632	262	12	229.41 MHz	21.79 ns

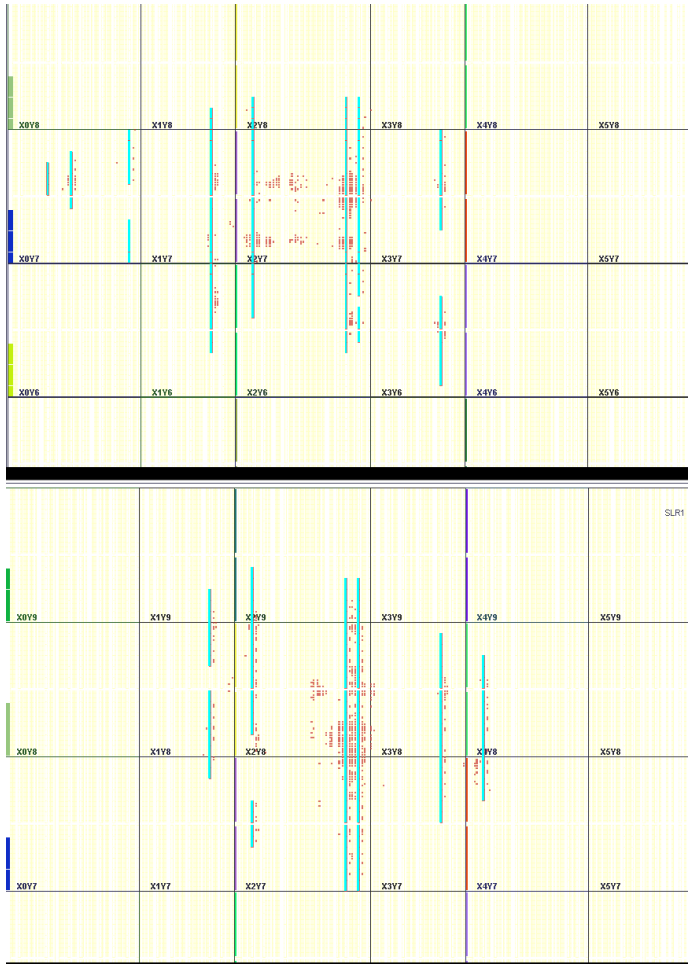


Figure 5.9: Floorplan of BRAMs (in red) in the naive (top) and optimized (bottom) A-CM Sketch

Throughput: The optimized A-CM Sketch implementation achieves a maximum theoretical throughput of ∼400 Million packets per second on the NFB-200G2QL [185] platform, as shown in Table 5.5. It can process a 96-bit flow ID per clock cycle irrespective of the packet size, with the only bottleneck limiting the throughput being the network interface. Assuming a minimal packet size of 64 bytes, A-CM Sketch can deliver a throughput of ∼204 Gbps. Table 5.6 shows that our implementation outperforms state-of-the-art sketches on FPGA.

Table 5.5: Throughput computation of the optimized A-CM Sketch

Observed period	Packet size (bits)	Processing speed (bits/CC)	Clock Frequency (ns/CC)	Packets per NIC	Overall throughput (packets/s)	Overall throughput (bits/s)
1 Second	512	512	5	2×10^8	4×10^8	2.04×10^{11}

Table 5.6: Throughput comparison of FPGA-based sketch implementations

Platform	Sketch	Flow-ID size	d, w	Memory	TP (Gbps)	ϵ	$1-\delta$
Virtex-2 [187]	CM	32-bit	$4, 2^{12}$	2KB	61	0.0006	0.98
Virtex 2000Ex3 [218]	K-ary	32-bit	$5, 2^{12}$	2.5KB	22	-	-
Virtex-5 [237]	K-ary	32-bit	$1, 2^{16}$	8KB	33	-	-
Virtex-2 Pro [158]	K-ary	32-bit	$3, 2^{15}$	12KB	4	-	-
Virtex Ultrascale [230]	K-ary	32-bit	$5, 2^{16}$	40KB	159	-	-
Virtex Ultrascale [209]	CM	32-bit	$4, 2^{14}$	8KB	128	0.0001	0.98
Virtex Ultrascale [230]	CM	32-bit	$5, 2^{16}$	40KB	155	0.00004	0.99
Virtex Ultrascale [146]	CM	32-bit	$5, 2^{17}$	80KB	16	0.00002	0.99
Ours,Virtex Ultrascale+	CM	**96-bit**	$4, 2^{16}$	32KB	**204**	0.00004	0.98

TP-Throughput; KB-Kilobyte

5.4 Design and analysis of SPArch

In this section, we propose a hardware-efficient sketch-based counter architecture called SPArch (Sketch-based Pseudo-associative array Architecture) for all-purpose flow measurement and security applications to confront the drawbacks of existing probabilistic measurement algorithms. SPArch combines the functionality of an associative array with the memory-saving benefits of sketch architectures. SPArch addresses the challenges associated with existing flow measurement solutions concerning hardware-friendliness, underutilization of memory, restricting the overall memory usage to on-chip memory, and assigning counter sizes based on flow sizes.

Furthermore, we strive to fulfill the need for a standardized measurement unit suitable for both eviction-based and sketch-based detection algorithms. As a standardized measurement unit, SPArch can serve the purpose of a standalone monitoring algorithm as well as a supporting measurement architecture. The primary focus of this work is on measurement architectures that are hardware-friendly and are capable of processing the incoming network traffic at line rate in Terabit Ethernet networks. Such architectures are particularly relevant for network security applications such as DDoS attack detection, where the flow measurement is bound to a predetermined measurement epoch.

To the best of our knowledge, none of the existing sketch-based methods employ a one-to-one mapping, akin to associative-array counters, for storing incoming flows. SPArch, on the other hand, applies one-to-one mapping to the incoming flows, allowing the flexibility to map elephant and mouse flows in separate buckets without incurring any additional memory accesses, as Section 5.4.4 explains. This functionality of mapping flows based on their sizes resembles that of some of the existing sketch architectures, such as Elastic Sketch [253] or Diamond Sketch [250], but with significantly lower memory usage. In addition to aiding the mapping of flows into appropriately sized buckets, one-to-one mapping also facilitates deletion or decrement operations. In sketches, deleting or decrementing the count value in a cell would affect the accuracy of other flows mapped to the same cell. SPArch effectively addresses this problem. Also, hash collisions increase the estimation error of sketches, whereas SPArch can detect hash collisions and identify two different flows with the same hash index as separate flows (Section 5.4.2).

The main contributions of this work are:

- The proposal of a hardware-oriented measurement algorithm termed **Sketch-based Pseudo-associative array Architecture (SPArch)**, which can achieve superior accuracy with the lowest memory utilization compared to existing sketch/sketch-based measurement algorithms. SPArch can detect hash collisions. Furthermore, SPArch mitigates the underutilization of memory, a dominant drawback of sketches.
- The proposal of a flexible counter architecture that enables one-to-one mapping, allowing the organization of counters in multiple buckets without affecting the latency. The multi-bucket architecture enables the segregation of elephant flows and mouse flows into distinct buckets, facilitating the possibility of different counter sizes for mouse and elephant flows. An approximation technique is presented to scale down the sizes of the buckets to reduce the memory footprint even further.
- The evaluation of SPArch for large-flow detection.
- The implementation of SPArch on an FPGA and the evaluation of the performance.

5.4.1 Preliminaries

SPArch follows the basic data structure of Count-Min Sketch (CM Sketch, explained below) and shows the best accuracy to memory trade-off amongst the other sketches. Hence we used CM Sketch for comparison and illustration

purposes. Another data structure that follows the architecture of CM Sketch and uses the idea of using flow-fingerprints similar to SPArch is HeavyKeeper [117]. HeavyKeeper proves to be more efficient than some other algorithms, such as Space-saving [178] and Lossy counting [173]. To analyze the efficiency of SPArch, it is essential to implement HeavyKeeper and compare the two. Additionally, Elastic Sketch [253] is currently one of the most memory-efficient algorithms for applications like heavy-hitter and heavy-change detection. SPArch is also evaluated against Elastic Sketch to assess its efficiency.

The architectural details of CM Sketch is discussed in Section 5.2. The architecture of HeavyKeeper and Elastic Sketch are explained below. The two major operations of an architecture are the **update** operation and the **query** operation. An incoming network flow is mapped as a pair of flow ID x and size s, as (x, s). The update operation updates the architecture with an (x, s) pair, while the query performs a lookup of s, given a flow ID x.

HeavyKeeper: The data structure of HeavyKeeper is shown in Figure 5.10. It is a 2-dimensional array comprising of d rows, where each row consists of w buckets. Each bucket has two fields: a fingerprint and a counter. Each incoming flow x_i is mapped to d buckets using d independent hash functions h_j, where j=1,...,d. Another hash function h_f is used to generate the fingerprint from the flow ID. Each bucket is represented as $A_j[h_j(x_i)]$, and the fingerprint and counter fields in the bucket are represented with $.FP$ and $.C$ suffixes, respectively.

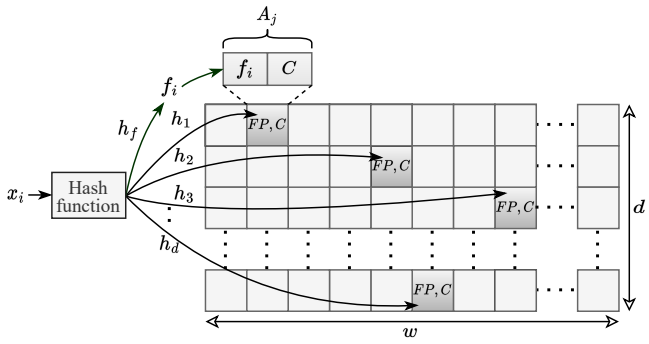


Figure 5.10: Data structure of HeavyKeeper

Update operation The update operation is based on three different strategies. If $A_j[h_j(x_i)].C = 0$, the fingerprint field is set as $A_j[h_j(x_i)].FP = f_i$ and the counter field is set as $A_j[h_j(x_i)].C = 1$, where f_i is the fingerprint of flow x_i. If $A_j[h_j(x_i)].C > 0$ and $A_j[h_j(x_i)].FP = f_i$, the counter field is updated

as $A_j[h_j(x_i)].C = A_j[h_j(x_i)].C + inc_i$, where inc_i is the incremental size of flow x_i, where $inc_i = s$. When $A_j[h_j(x_i)].C > 0$ and $A_j[h_j(x_i)].FP \neq f_i$, the proposed count-with-exponential-decay strategy is applied to this bucket and $A_j[h_j(x_i)].C$ is decayed with a probability.

Query operation While querying, the incoming flow ID x_i is hashed d times, mapping to d buckets. The maximum counter value of all hash-indexed buckets that have a matching fingerprint is taken as the output:
$estimate = max(A_j[h_j(x_i)].C, \forall j \in 1..d)$.

Elastic Sketch: The data structure of the basic version of Elastic Sketch is shown in Figure 5.11. Elastic Sketch consists of a *heavy part* for storing elephant flows and a *light part* for storing mouse flows. The *heavy part* is a simple hash table, where each bucket in the hash table stores the *key* (flow ID), positive votes ($v+$), negative votes ($v-$), and a $flag$. The number of packets belonging to the flow present in the bucket is indicated by $v+$, while $v-$ stores the number of packets belonging to any other flow mapped to the same bucket. The incoming flows are mapped to the *heavy part* using the address generated by a hash function h. The *light part* is a CM Sketch. An incoming flow is first mapped to the *heavy part*. Based on the update scenarios, the flow is then evicted to the *light part* or vice versa if required.

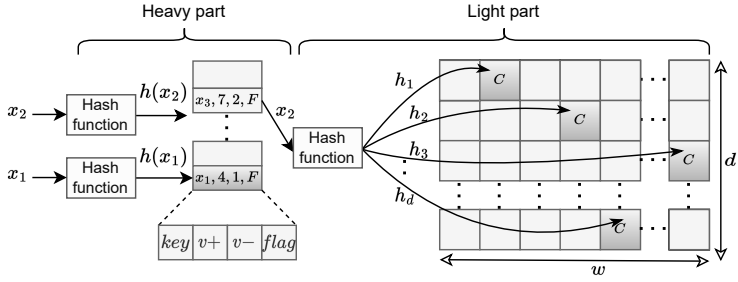


Figure 5.11: Data structure of Elastic Sketch

Update operation During the update, the incoming flow x_i is mapped to the *heavy part* first using the hash value $h(x_i)$. If the mapped bucket is empty, $(x_i, 1, 0, F)$ is inserted into that bucket. $flag = F$ indicates that no eviction has happened in the bucket. If the bucket is not empty and $x_i = key$, the value of $v+$ is incremented by 1. If the bucket is not empty and $x_i \neq key$, the value of $v-$ is incremented by 1. If $\frac{v-}{v+} \geq \lambda$, then the stored flow is evicted and $(key, v+)$ is inserted into the *light part*, and the *heavy part* is stored with $(x_i, 1, 1, T)$. Here, λ is a predefined threshold, and $flag = T$ indicates the eviction in the bucket. If $x_i \neq key$ and $\frac{v-}{v+} < \lambda$, $(x_i, 1)$ it is inserted into the CM Sketch.

Query operation During the query operation, the size is queried from the CM Sketch if the flow is not present in the *heavy part*. If the flow is present in the *heavy part* and $flag = F$, the size is the value of $v+$. If the flow is present in the *heavy part* and $flag = T$, then the values are retrieved from both heavy and *light parts*, and their sum gives the actual size.

5.4.2 Architecture of SPArch

SPArch is a 2-dimensional array of size (d, w) with an associated 1-dimensional counter memory array of size N_c, as shown in Figure 5.12. The depth of the sketch is d, also representing the number of hash functions required to map each flow to the sketch. The width of the sketch is w, representing the number of cells/buckets in each row. Each cell in the sketch, termed fingerprint-address cell or FAC, comprises two fields: a fingerprint field f and an address field A. An address generator, a simple $log_2 N_c$-bit up-counter, is used to generate the address A of the counters, and the generated address is stored in an address register (*Address Reg* as shown in the figure). The update and query operations of SPArch are discussed in detail in the following sections.

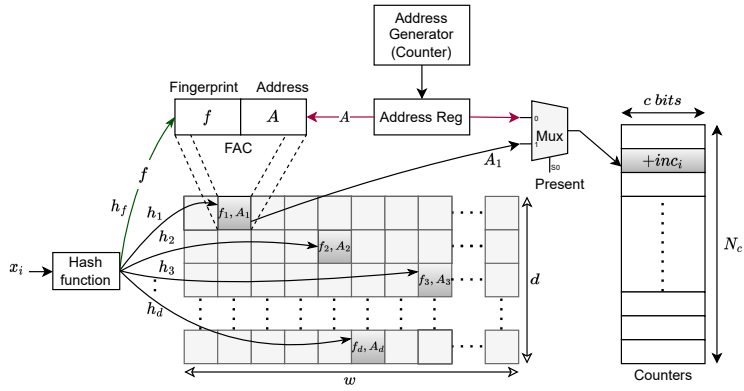


Figure 5.12: Data structure of SPArch

Update operation

An incoming flow x_i is mapped to the FAC using d independent hash functions h_j, where j=1,...,d. Each FAC is represented as $FAC_j[h_j(x_i)]$ and consists of the fingerprint-address pair (f, A). A unique fingerprint is generated by hashing the flow ID using a hash function h_f. The size of the fingerprint is determined

to achieve optimal accuracy while keeping it to a minimum possible value. The size of the fingerprint (f) in bits is denoted by q. The fingerprint must be non-zero, and if a generated fingerprint happens to be zero, it is assigned the maximum possible value of $2^q - 1$. If any of the FACs have a fingerprint value of zero, it indicates that the corresponding element is not present. Conversely, if all fingerprint values are greater than zero, the presence of the element depends on whether there is a matching fingerprint. Instead of using separate hash functions to generate sketch indices and fingerprint, a single hash function is employed. This hash function generates the required hash bits, which are then divided into d hash indices and the fingerprint. The total number of hash bits required is $d \times log_2 w + q$. Our proposed Xoodoo-NC (cf. Chapter 3, Section 3.3) is the hash function used to generate the required hash values.

The counter address A in the FAC points to the counter memory to be updated with the incremental flow size inc_i. When a new incoming flow is updated, the address of the counter A is retrieved from the address register (*Address Reg* as shown in the figure) and written to the FAC along with the fingerprint, if the FAC is not yet occupied. Once the update is complete, a new address is generated by an address generator. This newly generated address is then stored in the address register for the next incoming flow update. The counter memory array has N_c counters, each with a size of c bits. SPArch can handle n flows simultaneously, where $n{=}N_c$, and n can even be greater than w, whereas the other sketches, such as CM Sketch, can only handle w flows simultaneously. Choosing a value of w much lower than n still yields high accuracy for SPArch, and Section 5.4.3 provides a detailed analysis. The size of the address part A is a bits, where $a = log_2(N_c)$. For a comprehensive list of the notational conventions used in the SPArch architecture, please refer to Table 5.7.

There can be multiple scenarios in the update operation, and the subsequent sections discuss all of these scenarios. Algorithm 3 describes the complete algorithm of the update operation. Implementing multiple scenarios for update and query operations helps to enhance the hardware-oriented design of SPArch. Irrespective of the depth of the sketch and the number of counters, there is a memory access latency of only three clock cycles for update operation (sketch read, counter read, and sketch and counter write) and two clock cycles for query operation (sketch read, counter read). Each row in the sketch can be accessed in parallel as they are separate memory blocks. Additionally, the sketch and the counter write operations can be executed together within a single clock cycle. All the update and query computations can be performed in parallel once the memory read operation is complete. Consequently, the multiple update/query scenarios do not introduce any extra latency, resulting in a time complexity of $O(1)$.

Table 5.7: Notational conventions of SPArch

x	Incoming flow
inc_i	Incremental flow size
d	Depth of the sketch/number of hash functions
w	Width of the sketch
N_c	Number of counters
n	Number of distinct flows
A	Address of the counter
FAC	Fingerprint-Address Cell (A Bucket)
$FAC_j[h_j(x)]$	FAC in the jth row having the address $h_j(x)$
f	Fingerprint
h_j	Hash function to index the jth row of the sketch
h_f	Hash function to generate fingerprint
q	Size of the fingerprint in bits
c	Size of the counter in bits
a	Address size of counters in bits ($log_2 N_c$)

Update scenario: If the element x_i is not present

Case 1: Some (or all) of the locations indexed by the hashes are empty: This scenario is visualized by the sub-figures (a) and (b) in Figure 5.13. In this scenario, a new address A_x is obtained from the address generator. The empty FACs in the hash-indexed locations of the sketch will be updated with the fingerprint f_x and the address A_x: $FAC_j[h_j(x)] \leftarrow (f_x, A_x)$.
After sampling, the value of the address generator is incremented. The non-empty locations will not be overwritten. Finally, the counter memory location is updated at the newly added address A_x with the corresponding incremental size *inc* (lines 10-14, Algorithm 3).

Case 2: None of the hash-indexed locations are empty, none of the fingerprints match, and fingerprint-address pairs in all locations are different: This case is represented by the sub-figure (c) in Figure 5.13. In this scenario, the fingerprint-address pairs having the lowest address (which represents the oldest flow inserted) in the FAC is replaced with the incoming fingerprint and the address pair: $FAC_j[h_j(x)]_{min(A)} \leftarrow (f_x, A_x)$.
After sampling, the value of the address generator is incremented. Finally, the counter memory location at A_x is initialized with the incremental size *inc* (lines 16-20, Algorithm 3). The address of the evicted flow is not reused until the next reset period.

To ensure that the highest address values are indeed the newest, SPArch denies any new updates until the new measurement period in case of an overflow, where none of the N_c counters are empty, and the address generator has reached its

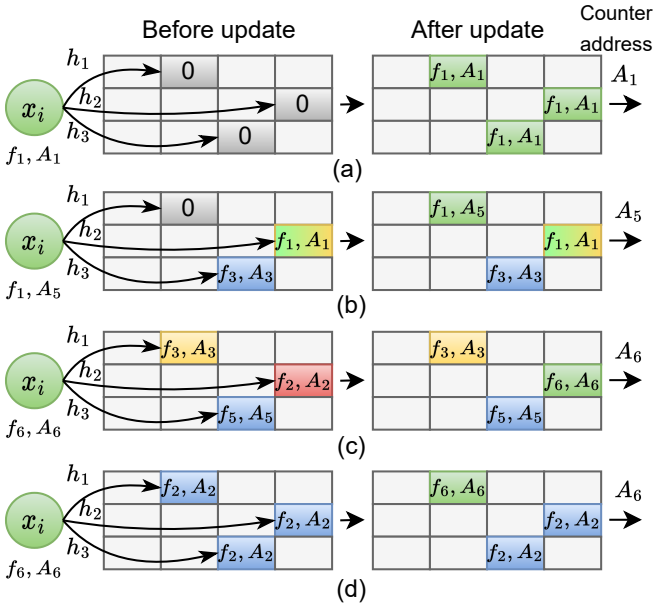


Figure 5.13: Update scenarios if the flow is not present

maximum counter value. There may theoretically arise a security vulnerability as an adversary can send arbitrary amounts of traffic without being recorded once the address generator reaches its maximum value and SPArch denies updates. However, this vulnerability cannot practically happen if we ensure that the number of counters N_c, which is the maximum value generated by the counter, chosen must be enough to accommodate all possible flows for the measurement period. Moreover, double insurance can be guaranteed by adding an extra bit to the address generator counter, which doubles the number of addresses generated, ensuring that the address generator does not reach its maximum value before the measurement period.

Case 3: None of the hash-indexed locations is empty, none of the fingerprints matches, and all locations have an identical fingerprint-address pair: This scenario is represented by the sub-figure (d) in Figure 5.13. In this scenario, the fingerprint-address pair in the first row is replaced with the incoming fingerprint-address pair: $FAC_j[h_j(x)]_{j=1} \leftarrow (f_x, A_x)$.
The value of the address generator is incremented. The counter memory location at the newly added address is updated with incremental size *inc* (lines 21-24, Algorithm 3).

Update scenario: If the element x_i is present

Case 4: One or more of the hash-indexed locations have a unique matching fingerprint, and all these matching fingerprints are paired with the same addresses: This scenario is represented by sub-figures (a) and (b) in Figure 5.14. In this case, the counter memory is updated with the address from the matching fingerprint-address pair with incremental size *inc* (lines 26-28, Algorithm 3).

Case 5: Two or more of the hash-indexed locations have a matching fingerprint, these matching fingerprints are paired with multiple different addresses: This scenario is represented by sub-figures (c) and (d) in Figure 5.14. In this case, the address which occurs the most is used to update the counter memory. If no address has a majority, the address of the most recent fingerprint-address pair is used (Lines 29-38, Algorithm 3).

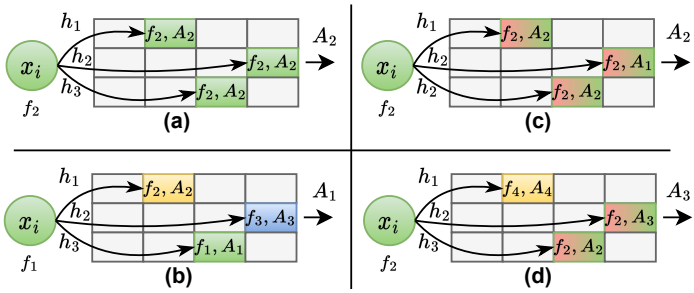


Figure 5.14: Update scenarios if the flow is present

Collision resistance

SPArch exhibits excellent collision resistance and is able to detect hash collisions during update operations. The collisions can be hash-index collisions, fingerprint collisions, or both. Cases 2 and 3 of the update operation represent the occurrence of hash-index collisions where the incoming flow is mapped to already existing locations. Cases 4 and 5 of the update operation represent the occurrence of fingerprint collisions as well as the occurrence of both fingerprint and hash-index collisions. In the case of a hash-index collision, the fingerprint is used to identify the collision and distinguish the element. In case of a fingerprint collision or both hash-index and fingerprint collision, the stored address acts as a unique identifier to distinguish the elements and detect the collision.

Nevertheless, there may have very rare scenarios where collisions cannot be detected. For example, a flow x with fingerprint f is mapped to four FACs, (f, A), (f, A), (f, A), and $(none, none)$. As per Case 1, the last FAC will be updated as (f, B). A recurrence of x would result in updating the counter

Algorithm 3: Update operation

Parameters: Width of sketch - w, Depth of sketch - d, Number of counters - N_c, Size of the counter (bits) - c, Incoming flow at time t - x_t, Fingerprint of x_t - f_{x_t}, Flow size of x_t - s_t

Initialize():
  Create a $d \times w$ sketch array and a $N_c \times 1$ counter array. All initialized to 0.
  Address generator value initialized to 0 ($AdGen[value] = 0$).
  $inc = s$
Update(x, inc):
$f_x = h_f[x], A_x = AdGen[value]$
for $j \leftarrow 1$ *to* d **do**
  $(f_j, A_j) \leftarrow FAC_j[h_j[x]]$
  if $f_j = 0, \exists\ j$ **then**
    $FAC_j[h_j[x]]_{f_j=0} \leftarrow (f_x, A_x)$
    $Counter[A_x] = Counter[A_x] + inc$
    $AdGen[value] = AdGen[value] + 1$
  end
  else if $f_j \neq 0,\ \forall\ j$ **then**
    if $f_j \neq f_x,\ \forall\ j$ **then**
      $\forall\ FAC_i[h_j[x]]_{min(A_j)} \leftarrow (f_x, A_x)$
      $Counter[A_x] = Counter[A_x] + inc$
      $AdGen[value] = AdGen[value] + 1$
    end
    else if *($f_j \neq f_x$ and $FAC_j[h_j[x]]$is same, $\forall\ j$)* **then**
      $FAC_j[h_j[x]]_{j=1)} \leftarrow (f_x, A_x)$
      $Counter[A_x] = Counter[A_x] + inc$
      $AdGen[value] = AdGen[value] + 1$
    end
    else if $f_j = f_x\ \exists\ j$ *and* A_j *is same* $\forall\ f_j = f_x$ **then**
      $Counter[A_j] = Counter[A_j] + inc$
    end
    else if $f_j = f_x\ \exists\ j$ *and* A_j *is not same* $\exists\ f_j = f_x$ **then**
      if *Any* A_j *has majority* **then**
        $A_j = majority(A_j)|_{f_j=f_x}$
        $Counter[A_j] = Counter[A_j] + inc$
      end
      else
        $A_j = max(A_j)|_{f_j=f_x}$
        $Counter[A_j] = Counter[A_j] + inc$
      end
    end
  end
end

A as per Case 5 based on the majority, as there is a fingerprint collision. However, such scenarios are extremely rare that occur due to hash collisions. The hash function Xoodoo-NC ensures that the hash output is completely random for each flow ID. Xoodoo-NC maps a 96-bit input to a 96-bit output space and has near-perfect avalanche scores, ensuring sufficient randomness to limit the simultaneous occurrence of fingerprint and hash-index collisions for the scenario described above to happen. Considering the birthday attack, an average collision occurs only after encountering around $1.17 * 2^x$ flows, where $x = min(d * lg(w) + k, 96)/2$.

Query operation

The query operation also has to consider multiple scenarios. Algorithm 4 provides the complete algorithm of the update operation.

Case 1: Some (or all) of the locations indexed by the hashes are empty, and *Case 2: None of the locations indexed by the hashes have a matching fingerprint* Sub-figure (a) in Figure 5.15 represents this scenario. In both cases 1 and 2, the counter value is zero (lines 5-7, Algorithm 4).

Case 3: None of the locations indexed by the hashes are empty; one or more fingerprints match; and all the corresponding addresses in the fingerprint-address pair are identical This scenario is represented by the sub-figure (b) in Figure 5.15. The counter value can be obtained through the unique address from a matching fingerprint-address pair (lines 8-10, Algorithm 4).

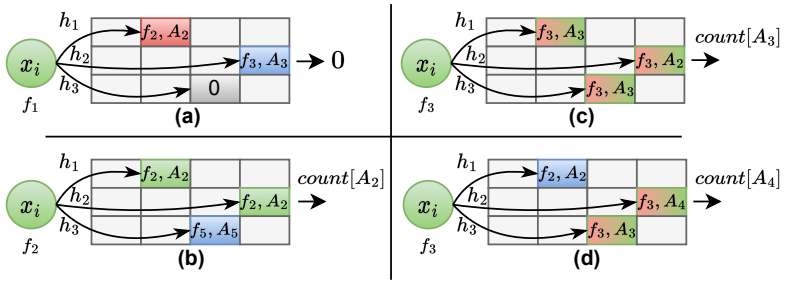


Figure 5.15: Query scenarios

Case 4: None of the locations indexed by the hashes are empty; one or more fingerprints match; but the corresponding addresses are different: This scenario is represented by sub-figures (c) and (d) in Figure 5.15. The counter value at the most frequently occurring address is retrieved. If no address has a majority,

the fingerprint-address pair having the most recent address is selected and used to retrieve the count (lines 11-20, Algorithm 4).

Algorithm 4: Query operation

```
1  Estimate(x):
2  f_x = h_f[x]
3  for j ← 1 to d do
4      (f_j, A_j) ← FAC_j[h_j[x]]
5      if (f_j = 0, ∃ j) OR (f_j ≠ 0 and f_j ≠ f_x, ∀ j) then
6          Estimate = 0
7      end
8      else if (f_j ≠ 0 ∀ j) and (f_j = f_x ∃ j and A_j is same ∀ f_j = f_x) then
9          Estimate = Counter[A_j]
10     end
11     else if (f_j ≠ 0 ∀ j) and (f_j = f_x ∃ j and A_j is not same ∃ f_j = f_x) then
12         if Any A_j has majority then
13             A_j = majority(A_j)|_{f_j=f_x}
14             Estimate = Counter[A_j]
15         end
16         else
17             A_j = max(A_j)|_{f_j=f_x}
18             Estimate = Counter[A_j]
19         end
20     end
21 end
```

Deletion operation

The deletion operation follows the same process as the update operation. If the element to be deleted is not present ins the sketch, the deletion operation is halted. If the element is present, the FAC(s) that contain the matching fingerprint-address pair is/are emptied, and the corresponding counter is reset.

5.4.3 Analysis of SPArch

This section provides an in-depth analysis of SPArch as a standalone monitoring algorithm. It should also be noted that the analysis holds true for the functionality of SPArch as a supporting associative array architecture, working in tandem with the other algorithms. An extensive analysis of SPArch is performed with varying parameter sizes. A comparison is drawn between SPArch and other related algorithms, namely CM Sketch, HeavyKeeper, and Elastic Sketch. These algorithms are also implemented with the corresponding parameter sizes to ensure a fair comparison. The analysis focuses on two key aspects: flow

measurement and large flow detection. A threshold is set for the latter, and the detection results are analyzed. The dataset used for the analysis is a subset of the CAIDA traffic trace from 2018, measured on a 10 Gbps link, modified for detecting large flows [1]. The analysis is carried out for the following metrics.

Accuracy: Accuracy defines the rate of correct estimation of the flow size compared to the actual size. Accuracy = $\frac{\hat{n}}{n}$, where $\hat{n}$ is the number of correctly estimated flows and n is the total number of flows inserted.

Average Absolute Error (AAE): AAE defines the mean absolute error rate of the estimated flow size with respect to the actual flow size. AAE = $\frac{1}{|X|}\Sigma_{x_i \in X}|\hat{n_i} - n_i|$, where X is the set of all flows x_i, and $\hat{n_i}$ and n_i are the estimated and real flow sizes of flow x_i respectively.

Average Relative Error (ARE): ARE defines the mean relative error rate of the estimated flow size with respect to the actual flow size. ARE = $\frac{1}{|X|}\Sigma_{x_i \in X}\frac{|\hat{n_i} - n_i|}{n_i}$, where X is the set of all flows x_i, and $\hat{n_i}$ and n_i are the estimated and real flow sizes of flow x_i respectively.

Precision: Precision refers to the fraction of correctly identified large flows present in the set of all large flows detected by the sketch. Precision = True Positives (TP) / (True Positives (TP) + False Positives (FP)).

Recall: Recall refers to the fraction of flows correctly detected as large flows by the sketch out of all the large flows that exist in the link. Recall = True Positives (TP) / (True Positives (TP) + False Negatives (FN)).

It is to be noted that the number of counters, N_c, in SPArch is always equal to the number of distinct flows to be handled, n. Hence, n also represents the total number of counters N_c in SPArch.

Evaluation of Accuracy

The accuracy is evaluated against the number of flows inserted and memory size. To evaluate against the number of flows n inserted, all algorithms maintain a constant width w of 1024 and depth d of 4. The purpose of evaluating against the number of flows inserted is to demonstrate SPArch's ability to handle a high influx of incoming flows. For evaluating against the memory size, the values of w and n are taken as 2048 and the value is d is taken as 4 for each algorithm. The fingerprint (FP) size of HeavyKeeper is kept as 12 bits. The fingerprint size of SPArch is set to 8 bits for evaluation against the number of flows inserted. Conversely, for evaluating against the memory size, the fingerprint size and depth d are varied accordingly to achieve the desired memory size. Since the data structure of Elastic Sketch has substantial differences from other algorithms,

the analysis is solely performed based on the memory size. The number of buckets in the *heavy part* is taken as 10% of the total number of elements n to be stored, and the *light part* has a width w equal to 90% of n. For Elastic Sketch, the counter size of *key* equals the flow ID size, and the sizes of $v+$ and $v-$ are 24 bits. The counter sizes in the *light part* are 12 bits, assuming that no jumbo frames are considered.

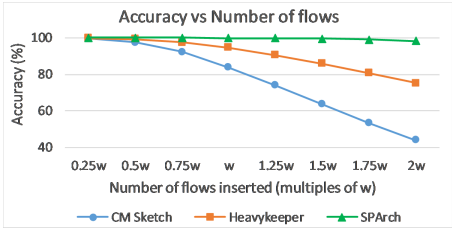


Figure 5.16: Accuracy vs number of flows

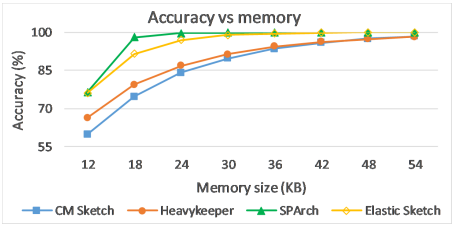


Figure 5.17: Accuracy vs memory size

Accuracy versus the number of flows n: Figure 5.16 depicts the accuracy with respect to the number of flows inserted as a multiple of w. The decrease in accuracy as the number of elements increases is minimal for SPArch, and its accuracy remains close to 100%. Even with n=2 × w, SPArch achieves an accuracy of 98.3%, whereas it is 75.3% and 44.0%, respectively, for HeavyKeeper and CM Sketch. For a counter size of 24 bits, CM Sketch and SPArch, SPArch with a fingerprint size of 8 bits, utilize a memory size of 12KB. In contrast, HeavyKeeper requires 18KB to accommodate the same number of flows.

Accuracy versus memory size: Figure 5.17 shows the plot of accuracy against the memory size. It is evident from the plot that SPArch achieves notably higher accuracy compared to other algorithms with identical memory constraints. SPArch achieves ∼99% accuracy with only 18KB of memory for n=w, whereas CM Sketch and HeavyKeeper require >300% more memory to achieve a similar level of accuracy. At a memory size of 12KB, Elastic Sketch and SPArch exhibit similar accuracy, but Elastic Sketch requires 200% more memory than SPArch to achieve 99% accuracy. The *heavy part* of Elastic Sketch is a hash table, and the hash table experiences an average collision rate of $\frac{n.(n+1)}{2w}$. Hence, for $n = w$, the average number of collisions in the *heavy part* is $\sim\frac{n}{2}$. The scenario where the size of the hash table w=0.1n results in a large number of collisions, which in turn gives rise to most of the incoming flows being mapped to the *light part*, even if the *heavy part* is not even 20% filled. Apparently, this causes the accuracy of the Elastic Sketch to be equal to the accuracy of CM Sketch. Notably, SPArch uses full-size counters, while 90% of the counters in Elastic Sketch are half-sized, making the memory reduction

achieved by SPArch highly significant. With a larger counter size, SPArch would achieve even greater memory savings compared to the other algorithms.

Evaluation of measurement error: average absolute error (AAE) and average relative error (ARE)

The AAE is evaluated against the number of flows inserted and the memory size. For evaluating against the number of flows inserted, the width w is set to 1024 for all algorithms, and the depth of the sketch d is kept as 4. For evaluating against memory size, the values of w and n are taken as 2048, and the value is d is taken as 4 for each algorithm to ensure a fair comparison. The counter size is 24 bits. The fingerprint (FP) size of HeavyKeeper is 12 bits. The authors of HeavyKeeper [117] set the fingerprint size as 16 bits for $d = 2$. As per our analysis, a fingerprint size of 12 bits is sufficient enough to provide the same accuracy as having a fingerprint size of 16 bits when $d = 4$, making HeavyKeeper more memory-efficient. The fingerprint size of SPArch is kept as 8 bits for evaluation against the number of flows inserted. The fingerprint size and depth d of SPArch are varied accordingly for the required memory size while evaluating against memory size. For Elastic Sketch, the analysis is performed only based on memory size. The parameters for Elastic Sketch are chosen as described in Section 5.4.3.

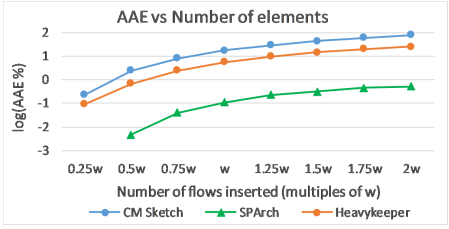


Figure 5.18: AAE vs number of flows

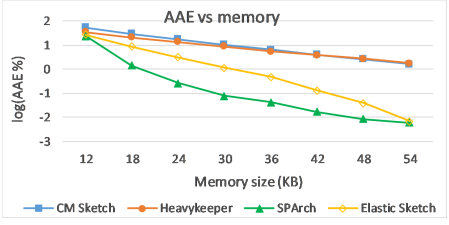


Figure 5.19: AAE vs memory size

AAE versus the number of flows inserted: Figure 5.18 shows the plot of $log_{10}(AAE)$ against the number of flows inserted. The plot shows significant differences between SPArch and the other algorithms. When $n = w$, the values of AAE are $\sim$50$\times$ and $\sim$159$\times$ higher for HeavyKeeper and CM Sketch, respectively, compared to SPArch. As n approaches $2 \times w$, the error becomes $\sim$78% for CM Sketch and $\sim$26% for HeavyKeeper, whereas the AAE is only 0.52% for SPArch, showcasing its exceptional inherent accuracy.

AAE versus memory size: Figure 5.19 illustrates the plot of $log_{10}(AAE)$ against memory size. For a memory size as small as 12KB, the AAE difference

of SPArch was only 2.2× and 1.5×, respectively, compared to CM Sketch and HeavyKeeper. As the memory size increases from 12KB to 18KB, the difference becomes a significant 21× and 15×, respectively, compared to CM Sketch and HeavyKeeper. As the memory size becomes as high as 54KB, the AAE for SPArch reduces by ∼4070× to reach 0.006%. At this point, the difference with CM Sketch and HeavyKeeper becomes ∼271× and ∼298×, respectively. It is very much evident from the analysis that SPArch possesses high accuracy, and the absolute error is considerably reduced even with a relatively smaller increment in memory.

ARE versus the number of flows inserted: Figure 5.20 illustrates the relationship between the number of inserted flows and the plot of ARE. Both HeavyKeeper and CM Sketch exhibit a gradual increase in ARE as the number of flows increases. However, HeavyKeeper achieves an ARE ∼3-4× lower than CM Sketch, for instance, 4.8% for $n = w$ and 21.8% for $n = 2w$ compared to 15.6% and 69.1% of CM Sketch. SPArch maintains a remarkably low error rate of only 0.4% even when $n = 2w$, making it 173 times lower than CM Sketch and 39 times lower than HeavyKeeper. The characteristics of Elastic Sketch fall between SPArch and the other two algorithms. Elastic Sketch offers better AAE than CM Sketch and HeavyKeeper, and reaches nearly zero when the memory size is 54KB or more. However, Elastic Sketch has a significantly higher error rate than SPArch. Furthermore, it is worth noting that 90% of the counters in Elastic Sketch use only 50% of the counter sizes than the other algorithms, and with 100% counter sizes, the error rate of Elastic Sketch will closely follow the AAE of CM Sketch.

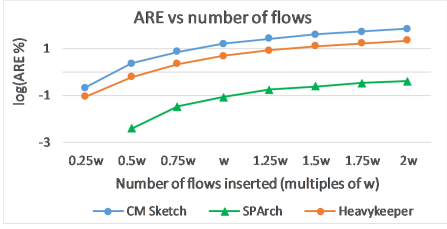


Figure 5.20: ARE vs number of flows

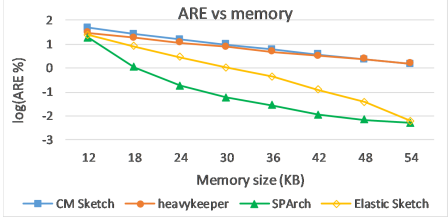


Figure 5.21: ARE vs memory size

ARE versus memory size: The plot of ARE against the number of flows inserted is shown in Figure 5.21. As seen from the figure, the curves of CM Sketch and HeavyKeeper closely align with each other, in contrast to SPArch, as observed in the plot of AAE. HeavyKeeper and CM Sketch exhibit nearly identical error rates with an equivalent memory allocation, indicating that HeavyKeeper demands a larger memory capacity in order to achieve higher accuracy. While the error rate is high for SPArch when the memory is 12KB, it

drops below 1% as the memory size surpasses 18KB, eventually approaching zero. For a memory size of 54KB, the ARE of SPArch is ∼300× better than both of the other algorithms. The ARE of Elastic Sketch demonstrates similar characteristics as observed in the AAE plot and falls between SPArch and the other two algorithms. The ARE of SPArch rapidly diminishes to zero, whereas the Elastic Sketch achieves an ARE of zero when the memory size reaches 54KB or more.

Evaluation of large flow detection: precision and recall

To analyze the detection of large flows where flow sizes exceed a predefined threshold, the values of w and d are set as 16384 and 4, respectively, for all the algorithms. The duration of the measurement cycle is set as 100ms. The fingerprint sizes of SPArch and HeavyKeeper are 8 bits and 12 bits, respectively. Similarly, the analysis of Elastic Sketch is performed solely on the basis of memory size. A threshold is set, and the flows whose aggregated sizes exceed the threshold are classified as large flows. The precision and recall are then evaluated against the number of flows inserted and the memory size.

Precision and recall versus the number of flows inserted: Figure 5.22 and Figure 5.24 display the plots of precision and recall, respectively, against the number of flows. As depicted in Figure 5.22, CM Sketch maintains a precision of 100% for any value of n, indicating that CM Sketch identifies all large flows without any misses. However, upon examining the plot of recall, the values gradually decrease as n increases. When $n = w$, the recall of CM Sketch is 98.5% and declines to 89% when $n = 2 \times w$. This is alarming as CM Sketch can falsely classify a benign flow as malicious, leading to false positives. The characteristics of HeavyKeeper are somewhat opposing to CM Sketch. The precision of HeavyKeeper experiences a significant decrease with the increase in w. For HeavyKeeper, the precision is 98.7% when $n = w$, which falls to 92% when n becomes $2 \times w$. It indicates that HeavyKeeper does not always filter the large flows out. Nevertheless, HeavyKeeper exhibits satisfactory results for recall, consistently achieving a recall rate greater than 99% even when $n = w$.

SPArch performs equally well on both precision and recall characteristics. SPArch holds a 100% precision and recall when $n = w$. The deterioration of precision is negligible as n increases. For instance, SPArch maintains the precision to 99.4% when $n = 1.75 \times w$ and 98.4% when $n = 3 \times w$. The decline of recall is also negligible and is always greater than 99%. Compared to CM Sketch, which achieves 100% precision due to overestimation, SPArch is on par when $n = w$, and there is only a slight decrease in precision as n increases. The precision of HeavyKeeper is the lowest as it underestimates the flows. On the

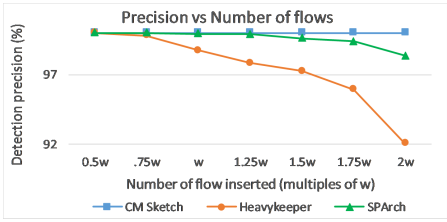


Figure 5.22: Flow detection: precision vs number of flows

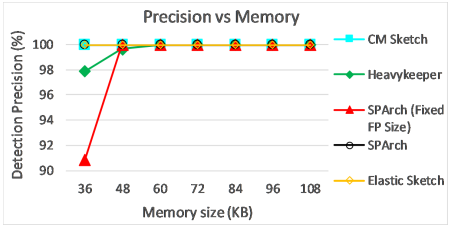


Figure 5.23: Flow detection: precision vs memory size

other hand, the overestimation of CM Sketch causes the misclassification of benign flows as malicious, resulting in blocking the legitimate flows. SPArch maintains a high recall percentage, indicating minimal false classifications. It is very much evident from the analysis that SPArch offers greater accuracy than the other algorithms. Although the precision of SPArch slightly diminishes for higher values of n when $n > w$, it does not imply underestimation like HeavyKeeper. The decrease in precision is mainly due to the replacement of the older flows with newer flows, as hash collisions become significant (Section 5.4.2, case-2 and case-3), which happens only when the value of n significantly exceeds w.

Precision and recall versus memory size: Precision and recall versus the memory size are plotted in figures 5.23 and 5.25, respectively, providing insights into the performance of the algorithms under different memory conditions. For evaluation against memory, $n = 4096$ is kept constant for each algorithm. In the case of HeavyKeeper, the fingerprint size is fixed at 12 bits, while the value of w is adjusted accordingly to meet the memory requirements. The size of the *heavy part* of Elastic Sketch is taken as $0.1n$, and the size of the *light part* is varied accordingly to fulfill the memory requirements. For SPArch, two cases are considered to meet the memory requirements. The first case involves a fixed FP size (set as 12-bits) while varying size of d (referred to as "SPArch (Fixed FP size)" in the figures). The second case involves varying both FP size ($4 \leq$FP size≤ 9) and d (referred to as "SPArch" in the figures). The latter case has a value of d one row higher than SPArch with a fixed FP size.

CM Sketch consistently maintains a precision of 100%, whereas recall becomes the lowest when the memory size is small due to significant overestimation. With increased memory size, the accuracy and recall of CM Sketch improve. HeavyKeeper exhibits a recall of 100% and reaches a precision of 100% when the memory size surpasses 60KB. Elastic Sketch is tailored for detecting elephant flows, exhibiting a 100% precision and near-hundred percent recall for a memory

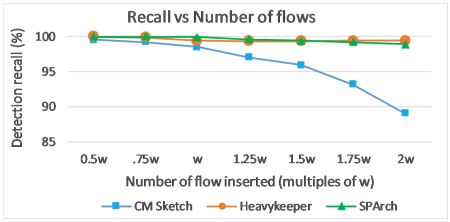


Figure 5.24: Flow detection: recall vs number of flows

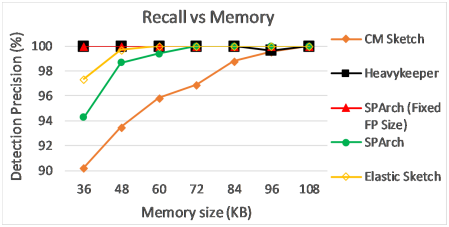


Figure 5.25: Flow detection: recall vs memory size

size of 36KB or more. As the overestimation property of Elastic Sketch closely aligns with CM Sketch, its precision remains consistently high. SPArch, on the other hand, offers the flexibility to tailor the results based on specific requirements. SPArch with varying FP size attains a precision of 100%, while SPArch with fixed FP size achieves a recall of 100%. Although the precision of SPArch with fixed FP size is low for a memory size of 36KB, it reaches 100% for memory sizes above 48 KB. Similarly, the recall for SPArch with varying FP size is close to 99% when the memory size is greater than 48KB and reaches 100% at a memory size of 72KB. Hence, compared to other algorithms, SPArch is more flexible in applying manipulations based on the memory requirements to achieve a 100% precision or recall.

Evaluating the performance of SPArch

To support the accuracy statements of SPArch, the analysis of accuracy, measurement error, precision, and recall of SPArch against varying fingerprint size q and depth d are presented here. This analysis provides insights into the high accuracy achieved by SPArch, even when employing smaller fingerprint sizes and lower depths.

Accuracy analysis of SPArch with varying fingerprint size and depth d: Figure 5.26 plots the accuracy of SPArch against the number of flows inserted, for various fingerprint sizes and a fixed depth of $d = 4$. To assess the stability of SPArch's accuracy, the number of flows inserted is increased to $4 \times w$. With a fingerprint size equal to 4, SPArch achieves an accuracy of 98.3% for n=w=1024 and 91.0% for n=$2 \times w$, surpassing both CM Sketch and HeavyKeeper while requiring less memory. With a fingerprint size greater than or equal to 6, SPArch can maintain near-hundred percent accuracy for n=$2 \times w$, and keep the accuracy higher than 75% for n=$4 \times w$.

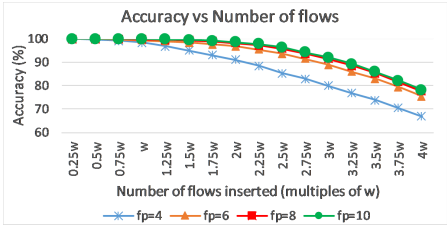


Figure 5.26: Accuracy vs number of flows (SPArch)

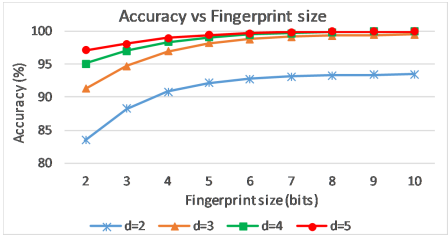


Figure 5.27: Accuracy vs Fingerprint size (SPArch)

Figure 5.27 illustrates the relationship between the accuracy of SPArch and the fingerprint sizes for $w = n = 1024$ and different values of d. Even with a fingerprint size of 2 bits and $d = 4$ (total memory requirement of 9KB), SPArch can achieve an accuracy exceeding 95%. This accuracy surpasses that of HeavyKeeper while utilizing only 50% of the memory required by HeavyKeeper. Additionally, when $d = 2$ and a FP size of 2 bits, SPArch attains an accuracy of 83.4% (total memory requirement of 6KB). This accuracy outperforms CM Sketch while utilizing only 50% of the memory compared to CM Sketch. Consequently, SPArch can achieve near-hundred percent accuracy with the lowest memory requirement.

Measurement error of SPArch with varying fingerprint size and depth: The analysis of ARE on SPArch is performed with varying depth (d) and fingerprint sizes, and the results are plotted in Figure 5.28. Remarkably, even with $d = 2$ and a fingerprint size of 5 (corresponding to a memory size of 6.75KB and $n = w$ =1024), SPArch surpasses the best attainable ARE values of CM Sketch and HeavyKeeper, where both CM Sketch and HeavyKeeper are allocated a memory size of 54KB, as indicated in Figure 5.21.

The error rates decrease gradually with increasing depth or fingerprint size, as illustrated in the figure. This characteristic is particularly advantageous in hardware implementations, allowing us to limit the usage of block RAMs by carefully choosing the number of rows of the sketch d or the fingerprint size to achieve a desired level of accuracy.

Precision and recall of SPArch versus fingerprint size and depth d: Figures 5.29 and 5.30 present the plot of precision and recall, respectively, for $n = w$ =1024 and varying fingerprint sizes of SPArch. When the FP size is at its lowest, the precision for $d = 2$ is slightly higher due to a significant overestimation error. However, it stabilizes at 93% when the FP size exceeds 3. When $d \geq 3$, the precision is ∼100% even when the FP size is as small as 2.

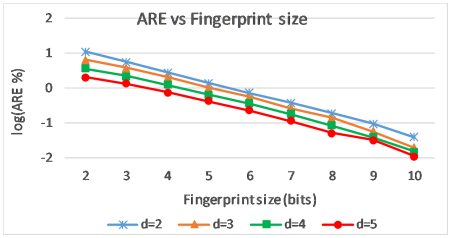


Figure 5.28: ARE vs Fingerprint size (SPArch)

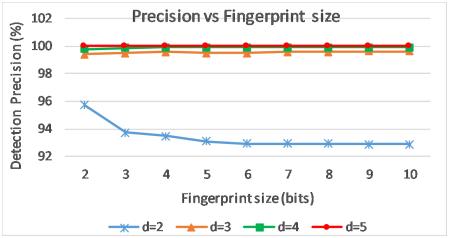


Figure 5.29: Flow detection: precision vs fingerprint size (SPArch)

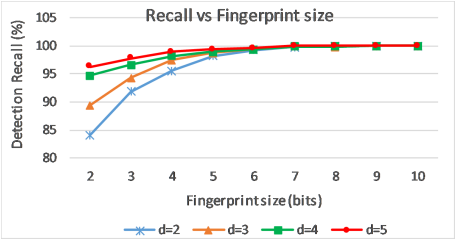


Figure 5.30: Flow detection: recall vs fingerprint size (SPArch)

As for the recall, it is lower for smaller FP sizes and gradually increases with increasing FP size. When the FP size exceeds 5, the recall attains ∼100% as the accuracy becomes maximum. This analysis underlines the earlier statement that SPArch offers greater flexibility, allowing users to select the memory size based on their preference for higher precision, higher recall, or both, considering the constraints of available memory.

5.4.4 Organizing counters in multiple buckets

Only a small proportion of the incoming flows would be classified as heavy-hitters, necessitating the use of large counters. Sketches, in general, allocate the same counter sizes for all the flows, resulting in underutilization of the allocated memory. Allocating different counter sizes for elephant and mouse flows is impossible for sketches as sketches do not follow one-to-one mapping like hash tables or associative arrays. However, SPArch offers the benefit of a flexible counter organization due to its one-to-one mapping. One way to leverage this property is by organizing flows into distinct buckets with

different counter sizes, allocating smaller counter sizes to mouse flows and larger counter sizes to elephant flows. Keeping separate buckets for elephant flows eliminates the possibility of removal of an elephant flow during collisions where the oldest address is removed. As the addresses allocated to elephant flows will be the largest, they cannot be removed during collisions when a mouse flow is already present in the counters. Another advantage is that approximation can be applied to different buckets with different scaling factors. Since such an implementation is advantageous on hardware, a hardware-oriented design is described here. The counter buckets can be organized in two ways: each bucket as an independent memory block or multiple buckets within the same memory block. The implementation details are discussed in the subsequent sections.

Counter buckets: as separate memory blocks

Figure 5.31 illustrates one approach to organizing counters into separate buckets. The counters are arranged in two buckets, b_m and b_e, with b_m dedicated to storing mouse flows and b_e designated for elephant flows. The total number of counters is N_c, with 80% of them allocated to b_m and the remaining 20% allocated to b_e. The number of bits allocated to each counter in b_m and b_e is c_m and c_e, respectively. Unlike SPArch, where a counter serves as the address generator, a FIFO, initialized with the addresses, is used as the address generator for bucket b_m, while b_e employs a counter as its address generator. It is possible to have a total of N buckets where the size of each subsequent bucket is larger than the preceding buckets. For all the buckets except the last one, FIFOs are employed as the address generators, whereas a counter is used as the address generator for the final bucket.

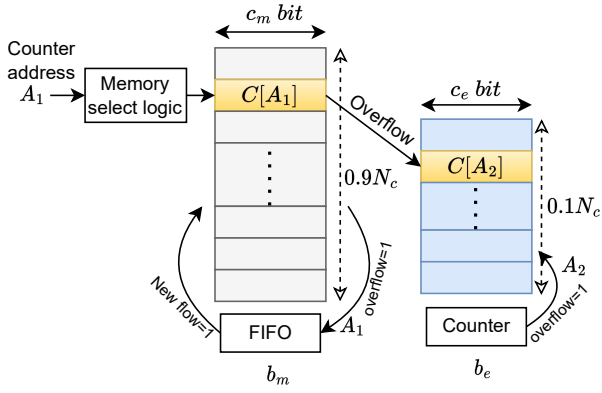


Figure 5.31: Organization of buckets: separate memory blocks

Update and Query operations: During the update process, the sketch sends the address (A_1) of the counter to be updated to bucket b_1. The address is read from the FIFO if the incoming flow is new. The counter corresponding to A_1 is updated, and if the counter overflows, it is moved to the address A_2 in the bucket b_2. The address of the counter A_2 is generated using the counter. The address A_1 of the overflown counter is written back to the FIFO. The new address A_2 is written to the FAC, replacing the existing address. Since we split the total number of counters into two buckets, one for mouse and one for elephant flows, an issue may arise. What if there are no elephant flows, and the counters for mouse flows are entirely occupied, while there are new incoming flows to be processed? In this case, if all the mouse counters are occupied, and the FIFO becomes empty, the next incoming flow is automatically stored in the elephant counters. Hence, as long as the number of flows to be stored is less than or equal to the total number of counters N_c, no flows will be missed. The query operation remains the same as discussed in Section 5.4.2.

Since each bucket is a separate memory block, the addresses of these blocks must be unique. Hence, a memory-select logic is employed to differentiate and map the addresses to the buckets. To keep the memory-select logic simple, an additional bit is introduced at the most significant bit (MSB) of the addresses. This creates a simple multiplexer serving as the memory-select logic. The value of the extra bit is '0' for the first bucket, and the value is '1' for the second bucket. While writing the address to the sketch, the corresponding bit at the MSB is appended. Typically, the existing MSB is used as the appending bit unless an overflow occurs. In case of an overflow, the MSB is incremented and appended with the new address. This keeps the computation requirements to a bare minimum. For the number of buckets (N) larger than 2, the number of extra bits to be added is $log_2 N$.

Latency: In contrast to alternative methods that employ multiple memory blocks to segregate flows, our approach maintains consistent latency. Transitions between buckets occur only when there is an overflow. During an overflow, the second bucket is not read but only written. By writing both the sketch and counter memory simultaneously, it ensures that latency remains unaffected, regardless of whether there is an overflow.

Approximate counting: Applying approximation to counters can reduce memory usage when precise measurement values are not required. For finding heavy-hitters that exceed a certain threshold, accurate measurements are not necessary. However, considering that mouse flows are relatively small, it would be unwise to apply the same scaling factors to buckets containing both mouse and elephant flows, as this could result in significant accuracy discrepancies for mouse flows. Therefore, having dedicated buckets for elephant flows simplifies the process of approximating the counts. Our proposed approximate counter

called HSAC, as discussed in Section 5.3.2, requires only 16 bits to represent a 41-bit counter.

Counter buckets: in the same memory block

Another way of organizing counters involves keeping multiple buckets in the same memory block, which helps in eliminating the memory-select logic and additional MSB bit to differentiate the counter buckets. Figure 5.32 illustrates the organization of these buckets. The memory block has a depth of N_c, which is divided into two (or more) imaginary sections. The first $0.9N_c$ counters are designated for storing mouse flows, while the remaining $0.1N_c$ counters are used to store elephant flows. In contrast to the counter buckets using separate memory blocks, FIFOs are used as the address generators for all buckets. When employing a single memory block for all the buckets, the size of all the counters must be the same, adhering to the width of the memory block. Hence, approximation or scaling is necessary to increase a bucket's storage capacity. Using HSAC approximation, only 16 bits are required to count up to 2^{41}. As a result, both b_1 and b_2 can be set to 16 bits, with approximation or scaling enabled in b_2. If more than two buckets are present, the scaling factor can be adjusted for each bucket, allowing for varying levels of approximation according to the flow sizes.

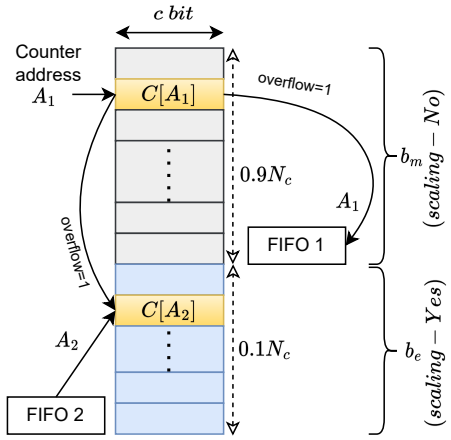


Figure 5.32: Organization of buckets: single memory block

Update and Query operations: When the counter having an address A_1 placed in bucket b_1 overflows during the update operation of a preexisting flow,

it is moved to bucket b_2, simultaneously reading the new address A_2 from FIFO2. The new address of the counter is written to the FAC, replacing the existing address. The original counter address A_1 is written back to FIFO1. Whenever a new flow arrives, it is typically written to b_1 using the address obtained from FIFO1 unless all counters in b_1 are already occupied. If all the counters in b_1 are occupied, the next incoming flow is mapped to b_2. The query operation remains unchanged, as discussed in Section 5.4.2, except that the estimated value is computed based on the approximate count for the values stored in b_2.

Approximating the count while moving from b_1 ***to*** b_2***:*** Approximating or scaling the measurements while moving the values from b_1 to b_2 is accomplished using HSAC. Initially, the count is rescaled to fit the format of HSAC. Also, a minor modification is applied to HSAC, such that we divided the counter into only two sections consisting of the exponent part (exp) and the counter part ($count$). The counter size of c-bits of bucket b_2 is divided into an l-bit exponent part and a k-bit counter part. The algorithm 5 presents the pseudo-code of the approximation process. The function $approximate(V_1)$ is applied explicitly during the transition from b_1 to b_2, while the $update(count, exp, inc)$ function is executed for all counter update operations in b_2.

Algorithm 5: Approximation operation

Parameters:V_1=Actual count, V_2=Approximated count, exp=l-bits, $count$=k-bits
`approximate(`V_1`):`
  Initialize:
    $count$=$\frac{2^k}{2}$
    exp= $c-k-1$
    inc=V_1-2^k
  end
  update($count, exp, inc$)
end
`update(`$count, exp, inc$`):`
  $count = count + \lfloor \frac{inc}{2^{exp}} \rfloor$
  With probability $\{\frac{inc}{2^{exp}}\}$: $count = count + 1$
  if $count > 2^k$ **then**
    $exp = exp + 1$
    $t_1 = count$
    $count = \frac{count}{2}$
    With probability $\{\frac{t_1}{2}\}$: $count = count + 1$
  end
end
Return$\{V_2 = [exp, count]\}$

Latency: The latency is similar to the counter buckets with separate memory blocks. The counter memory is read only once and written only once, making no increment in latency.

Reduction in complexity of multi-bucket counters

Multi-bucket counter significantly reduces the memory overhead in SPArch and has minimal additional computational requirements. However, incorporating FIFOs can add some complexity to the datapath and requires initialization time during startup. Furthermore, the overflown counter address written back to the FIFOs can cause a reduction in accuracy as the old addresses will be reused at the end. Nevertheless, the accuracy variation will be insignificant because only the last $0.1N_c$ elements will be using the reused addresses. Also, the removal of the re-used overflown addresses in the worst case of a collision (Case 2 of update operation) will not cause the eviction of any elephant flows as the elephant flows are in a separate bucket. These issues can be resolved by allocating a total of $1.1N_c$ counters: N_c counters for mouse flow bucket b_m and an additional $0.1N_c$ counters for the elephant flow bucket b_e. With this approach, reusing the addresses of b_m is no longer necessary, and a simple counter that serves as an address generator can replace the FIFO. The *empty* signal of the FIFO can be replaced by the *full* signal of b_m when the count value of the address generator reaches N_c. Similarly, the FIFO for b_e can be replaced by another counter having an initial value of N_c. This way, the complexity can be kept to a minimum by increasing the overall memory requirement of the multi-bucket counters by only $0.1N_c \times c_m$, where c_m represents the counter size of mouse counters.

Memory reduction

In addition to the benefit of having separate buckets to classify flows, there is a further reduction in memory requirements for the counters. Assuming 90% of the counters ($0.9N_c$) are allocated to the mouse flows, and the counter size is reduced by half for mouse flows, the total counter memory requirement is reduced by 45% without approximation. While applying approximation to elephant flows with the size of the counters reduced by half, the total counter memory requirement is reduced by 50%. However, there is no reduction to the sketch memory size as the fingerprint and address sizes remain unaltered. Assuming a fixed fingerprint size of 8 bits and $n = N_c = w$, the overall reduction in memory for $w = 65536$, $d = 4$ is 11.25% without approximation and 12.5% with approximation.

5.4.5 Hardware architecture of SPArch

Figure 5.12 shows a high-level view of the SPArch data structure, which comprises the sketch and the counter array. The sketch component is represented by a two-dimensional array with dimensions of $w \times d$, and the counter array consists of N_c counters. Figure 5.33 depicts the hardware architecture diagram of the SPArch data structure.

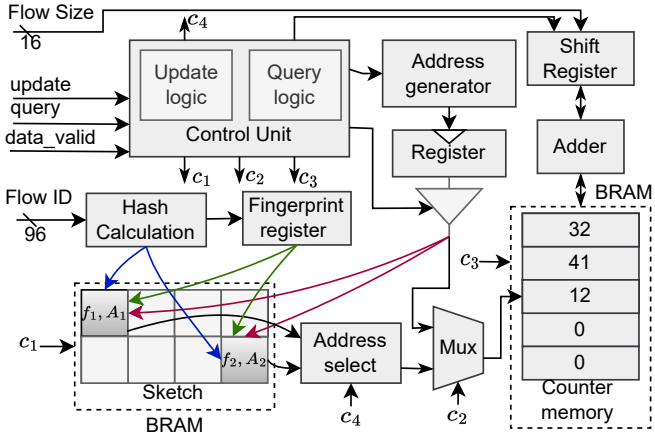


Figure 5.33: Hardware architecture of SPArch

The implementation of the sketch and counter array utilizes true-dual-port BRAM. The control unit is the brain of the architecture, which controls the sequence of operations to be performed. From the network packet header, a 96-bit flow ID is extracted and the hash values are computed. The hash-computation module is a crucial component of the hardware architecture and significantly impacts the latency of probabilistic data structures. Xoodoo-NC is employed to hash the flow ID. The resulting hash output is split into d hash values of size $log_2 w$ bits and a fingerprint of size q-bits. The hash values map the flow ID to the FACs, while the fingerprint helps determine the presence or absence of the incoming flow.

SPArch stands out from other sketch architectures primarily due to its unique counter array of N_c counters, and notably, N_c can be greater than the width of the sketch w. Unlike other sketches, where exceeding the number of counters by more than w can result in decreased accuracy (Section 5.4.3), SPArch offers greater flexibility in terms of counter quantity. For assigning flows to counters, an address generator generates addresses that are used to map each flow to a specific counter. The address-select logic determines the address of the counter

to be updated/queried if an incoming flow is present in the sketch. The shift register acts as a delay element to compensate for the lookup time of the sketch before the flow size computations can be performed.

The latency of the hash function, Xoodoo-NC, is a single clock cycle. The hash-indexed sketch memory is read in the subsequent clock cycle. The sketch is implemented with separate memory blocks for each row, allowing parallel execution of memory read and write operations within a single clock cycle. The update logic, in conjunction with the address-select logic, determines the address of the counter to be updated, and the counter memory is updated accordingly. The read/write operations of the counter memory also require only one clock cycle. The sketch and counter memory write operations are performed simultaneously in a single cycle, keeping the number of extra cycles required for SPArch compared to CM Sketch to only one. Even in the worst-case scenario, where a new packet needs to be processed in every clock cycle, SPArch can process it efficiently through pipelining. The parallel read/write capabilities of all the memory blocks facilitate pipelining easier. The true-dual-port BRAM ensures seamless execution of simultaneous read and write operations within a single cycle on a memory block when needed. Moreover, even during pipelining, there is no need to perform two simultaneous read or write operations within a single cycle for the sketch or counter memory blocks, even in the worst-case scenario.

5.4.6 Hardware evaluation

Experimental setup: The algorithm is evaluated on an FPGA using the Vivado 2020.2 design tool, specifically targeting the Alveo U250 Data Center Accelerator Card (xcu250-figd2104-2L-e). The parameters are chosen to allow for a fair comparison with other algorithms. The memory organization of CM Sketch, HeavyKeeper, and SPArch is kept similar for a fair comparison. The original hardware version of Elastic Sketch consists of several sub-tables of the *heavy part* connected sequentially, leading to a significant increase in memory accesses and latency. Hence, the simplified and hardware-friendly version of Elastic Sketch with a single *heavy part* is implemented, where the number of buckets in the *heavy part* of Elastic Sketch is 12.5% of the total number of counters. In order to minimize the hardware complexity and improve the throughput of Elastic Sketch, the division operations are converted to right-shift operations by approximating the denominators to the power of two. This approximation is applied only in the hardware implementation to achieve a better hardware performance, and the accuracy is computed as given in the analysis provided in Section 5.4.3. Only the sketch part of HeavyKeeper is implemented, and the fingerprint size is fixed to 12 bits, as mentioned in the

original literature. The pseudo-random number generation for HeavyKeeper is implemented using a linear feedback shift register (LFSR). As the number of counters of SPArch is equal to the number of distinct flows, the parameter n is used to represent both.

Accuracy versus resource utilization: To ensure a fair comparison, the performance of CM Sketch, HeavyKeeper, Elastic Sketch, and SPArch was assessed primarily based on accuracy. Since the accuracy with respect to memory usage varies significantly for these algorithms, evaluating these algorithms based on any other parameter would not be equitable. Additionally, comparing them solely on the basis of the depth of the sketch d would also be unfair, as the relative increase in the memory requirement of SPArch with an increase in the value of d is much lower than other algorithms. It is also evident from Section 5.4.3 that the error, precision, and recall of flow measurements are proportional to the accuracy. Although SPArch has a significant advantage in terms of relative memory consumption as d increases (which is also the depth of the *light part* for Elastic Sketch), the evaluation results are presented only for a constant value of d=4. The evaluation is performed for n=16384. The fingerprint size of SPArch is kept at 8 bits. The counters, including the *heavy part* of Elastic Sketch, are sized as 32 bits. The counter size of the *light part* of Elastic Sketch is taken as 16 bits, as the incoming flow size can be up to 16 bits.

Table 5.8 provides the details regarding the size of the data structures, the total theoretical memory requirement, and the hardware evaluation results for various configurations (n=16384 for all configurations) of all the implemented algorithms, aimed at achieving a specific level of accuracy. As previously mentioned in Section 5.4.3, the memory requirement to achieve ∼100% accuracy is the lowest for SPArch. To achieve an accuracy of at least 99.8%, SPArch requires only 54 BRAMs, whereas CM Sketch, HeavyKeeper, and Elastic Sketch require 4.3×, 5.9×, and 2.4× more BRAMs, respectively. The memory usage of SPArch can be further reduced by reducing the fingerprint size without any drastic changes in accuracy, as demonstrated in Section 5.4.3, and by organizing counters in multiple buckets and applying approximation, as discussed in Section 5.4.4.

Although HeavyKeeper demonstrates superior software efficiency compared to CM Sketch, its architecture is not optimized for hardware, resulting in poor performance in terms of latency and memory utilization. Additionally, the relative increase in memory (BRAM) utilization with respect to accuracy is higher for HeavyKeeper than for the other algorithms. Elastic Sketch, while also experiencing a relatively high increase in memory utilization concerning accuracy, still performs better than HeavyKeeper in this regard. However, the number of LUTs required for Elastic Sketch is higher than that of other algorithms because of the hardware unfriendliness causing additional hardware logic requirements. It should be noted that the hardware resource requirement

Table 5.8: Comparison of hardware resources and latency of SPArch against CM Sketch and Heavykeeper

	Accuracy	Sketch size	Memory requirement	LUT	BRAM	Operating Frequency	Update Latency	Throughput
CM Sketch	84.04	w=16384,d=4	2048 Kbit	**690**	58	270 MHz	**14.8 ns**	138 Gbps
	99.02	w=43008,d=4	5376 Kbit	**1059**	154	240 MHz	**16.6 ns**	123 Gbps
	99.77	w=65536,d=4	8192 Kbit	**1041**	232	216 MHz	**18.5 ns**	111 Gbps
HeavyKeeper	84.12	w=11392,d=4, FP size=12bit	1958 Kbit	1090	58	247 MHz	16.2 ns	126 Gbps
	99.02	w=32768,d=4, FP size=12bit	5632 Kbit	1132	160	220 MHz	18.2 ns	112 Gbps
	99.90	w=65536,d=4, FP size=12bit	11264 Kbit	1375	320	188 MHz	21.2 ns	97 Gbps
Elastic Sketch	84.07	LP:w=14464,d=4; HP:w=2048	1226 Kbit	1471	39	196 MHz	35.7 ns	100 Gbps
	99.01	LP:w=38400,d=4; HP:w=2048	2722 Kbit	1651	83	186 MHz	37.7 ns	95 Gbps
	99.84	LP:w=65536,d=4; HP:w=2048	4418 Kbit	1654	129	185 MHz	37.7 ns	95 Gbps
SPArch	84.03	w=4096,d=4, FP size=8bit	**864 Kbit**	1424	**24**	**294 MHz**	17.0 ns	**151 Gbps**
	99.00	w=16384,d=4, FP size=5bit	**1728 Kbit**	1504	**48**	**260 MHz**	19.25 ns	**133 Gbps**
	99.93	w=16384,d=4, FP size=8bit	**1920 Kbit**	1522	**54**	**258 MHz**	19.4 ns	**132 Gbps**

FP-fingerprint, HP-heavy part, LP-light part

of Elastic Sketch would be considerably higher if the approximated division optimization had not been performed in the hardware implementation. SPArch, on the other hand, proves to be more efficient on hardware, and the relative increase in memory usage with respect to accuracy is the lowest among the others. As the accuracy varies from ∼99.0% to at least 99.8%, the increase in memory usage of HeavyKeeper is ∼100%, while CM Sketch and Elastic Sketch experience a range of 50% to 60% increases. In contrast, SPArch only sees a modest increase of around 12%. From this analysis, it is evident that SPArch exhibits a remarkable combination of high accuracy and significantly low memory requirements, making it an optimal architectural choice.

Latency and throughput: Both HeavyKeeper and CM Sketch have the same latency since the operations involved are similar, such as hashing, memory read/write, and counter update. For SPArch, one additional counter memory read/write operation is required. Nevertheless, only one extra cycle, which is counter memory read, is required for SPArch because the sketch memory write operation can be performed simultaneously with the counter memory read/write operation. This leads to a latency that is only one clock cycle more than CM Sketch and HeavyKeeper. Despite the extra cycle latency, SPArch has a higher operating frequency than Elastic Sketch and HeavyKeeper for the same memory overhead. Additionally, SPArch operates at a higher frequency compared to the other algorithms while achieving similar levels of accuracy. SPArch employs separate memory blocks for each row of the sketch and the counter array. By using multiple smaller BRAMs instead of a single large block of memory, SPArch achieves better placement and routing, reducing the routing delay [215]. Since all memory blocks can be accessed in parallel, pipelining becomes more effortless, enabling SPArch to process a packet in every clock cycle. The basic version of Elastic Sketch with a single heavy and light part, however, has a worst-case latency of seven clock cycles.

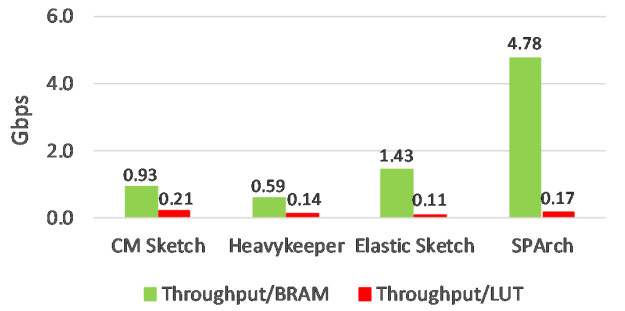


Figure 5.34: Throughput versus memory and area

Assuming a minimum packet size of 64 bytes and a worst-case scenario where a packet is received in every cycle, SPArch can process 258 Million packets per second (Mpps) with ∼100% accuracy for n=16384. In other words, SPArch can achieve a throughput of 132 Gbps with ∼100% accuracy for n=16384. Elastic Sketch has the lowest throughput among all the algorithms, while HeavyKeeper consumes the most memory. SPArch consumes less than half the memory of Elastic Sketch and ≈ 6× less memory than HeavyKeeper. The pipelining becomes laborious for Elastic Sketch as the latency in terms of the number of clock cycles varies with update scenarios, and pipelining may further increase the update latency. Figure 5.34 depicts the throughput density in Mpps for ≈ 100% accuracy plotted against the area and memory utilization. SPArch achieves significantly higher throughput per BRAM, 3.34× than Elastic Sketch, thanks to its low memory requirement for achieving 100% accuracy. SPArch also has a higher throughput per area, 1.2× that of HeavyKeeper and 1.5× that of Elastic Sketch. However, CM Sketch exhibits a better throughput per area despite having a low throughput by virtue of its lowest area requirement.

5.5 Conclusion

Probabilistic data structures are the most profitable components when it comes to memory consumption and speed, rendering them the ideal architectures for high-speed network flow measurements. The drawbacks of existing probabilistic flow measurement architectures made us rethink the way we design such probabilistic architectures for hardware implementations. In this chapter, hardware-efficient network flow measurement architectures are presented. Two distinct sketch-based flow measurement architectures are proposed, namely A-CM Sketch and SPArch.

A-CM Sketch (Approximate Count-Min Sketch) is an ultra-high-speed FPGA implementation of the CM Sketch algorithm using approximate counters. Our architectural optimization efforts in the CM Sketch data structure and the algorithmic optimization measures also resulted in the HSAC (Hardware-oriented Simple Active Counter) algorithm. With these optimizations, we managed to obtain an FPGA implementation that outperforms, to our knowledge, all existing sketch-based data structures on FPGA. Moreover, our implementation is compliant to a 200 Gbps network interface, making it suitable for network flow measurement at line rate in Terabit Ethernet networks.

SPArch, a hardware-oriented sketch-based counter architecture for per-flow measurements, exploits the parallelism of FPGA and delivers high accuracy while consuming a minimal amount of memory. We evaluated SPArch in all aspects and compared it against the best available sketch-based measurement architectures, and the analysis validated the superior performance of SPArch compared to other architectures. Unlike any other probabilistic data structure, SPArch can identify hash collisions and still correctly map the flow to distinct counters. SPArch is also capable of operating at line rate in Terabit Ethernet networks.

Even though this research only discusses and demonstrates the functionalities of SPArch as a counter array, the architecture of SPArch makes it ideal for various other applications. One such application is membership query, which is integral to network security applications. Employing associative arrays/exact membership query data structures is not preferred for real-time flow monitoring applications because of the reasons already mentioned in this research, which holds true for membership queries as well. Our analysis in Chapter 4 has demonstrated that existing approximate membership query data structures fail to achieve the best balance between memory usage and accuracy. Moreover, to the best of our knowledge, none of the existing probabilistic data structures can offer the same functionality as a CAM.

In contrast, SPArch can serve as a perfect alternative to CAM by removing the counter array and retaining only the sketch part. This architecture, during a query, can return the presence and the address of the element where it is located. Unlike CAM, this structure does not store the elements but provides a virtual representation of the storage. Additionally, SPArch can fulfill the functionalities of a key-value store without imposing any additional computational overhead or latency. SPArch also provides deletion support, just like a CAM. On the other hand, implementing CAM for the same purpose would require extra computational logic as well as latency, similar to what is needed for counter arrays. Our recent results provided in Chapter 4 validate that a CAM with a depth of only 1024, designed to store a 96-bit element, consumes 6731 LUTs and 320 BRAMs, which is immense. On the other hand, a SPArch key-value store

that can store 1024 elements with near-hundred percent accuracy requires only 810 LUTs and 2 BRAMs (cf. Chapter 6, Section 6.5.4). The above-mentioned results validate the advantage of SPArch as a replacement for CAM. Therefore, SPArch as a key-value store is employed in implementing our large flow detection framework, FLARE, which is presented in Chapter 6.

Chapter 6

Towards a universal large flow detection framework on FPGA for Terabit Ethernet

This chapter presents the design and implementation of a universal large flow detection framework for high-speed flow monitoring. The primary focus of this chapter is on detecting a range of large flow attack patterns by integrating standalone detection units dedicated to distinct attack patterns. This chapter is organized as follows. Section 6.1 provides a brief introduction to the topic. Section 6.2 discusses the challenges in the design of a comprehensive detection system, followed by Section 6.3, which provides details about the existing work on universal flow monitoring systems. Section 6.4 explains the system architecture and components of FLARE. A proof of concept and hardware evaluation of FLARE is performed in Section 6.5, and Section 6.6 concludes the chapter.

CONTENT SOURCE

- Sateesan, A., Vliegen, J., Scherrer, S., Hsiao, H.C., Perrig, A. and Mentens, N., FLARE: FPGA-based universal large flow detection engine for high-speed flow monitoring. (This work may appear in novel publications in the near future.) (Contributions: Main author)

6.1 Introduction

With the increasingly diverse attack patterns in recent years, mitigating attacks such as *large flows* with existing approaches becomes increasingly difficult. The rapid influx of data often cripples the large flow detection frameworks in various aspects, including processing speeds, memory requirements, and, most importantly, the versatility to handle a range of attacks. Previous approaches in large flow detection lack the ability to detect a wide range of attack patterns, as a single dedicated detection algorithm cannot provide complete detection capabilities against such patterns. The detection threshold and measurement epoch designated to detect an attack pattern may not be suitable for detecting another type of attack. This calls for deploying multiple detection algorithms to precisely detect a wide range of large flow attacks. However, the approach of deploying multiple algorithms requires significant effort to coordinate the standalone algorithms, in addition to the increased computational complexity and resource requirements. The complexity in coordination and computational overhead adversely affects the speed as well, which is of prime concern when real-time processing of data is required.

FPGAs can help achieve considerable acceleration in computational load and thereby improve the operating speed to allow real-time data processing, thanks to their substantial parallelism. However, deploying multiple independent detection units on FPGA is still a cumbersome task and requires effective management of resources. Despite having ample resources in high-end FPGAs, the coordination of multiple algorithms has an adverse effect on the complexity and operating speed of FPGA platforms as well.

None of the existing research targets a universal detection framework that can coordinate multiple detection algorithms to effectively handle a wide range of attacks. Existing universal sketches such as UnivMon [164] and Lightweight Universal Sketch (LUS) [243] take advantage of dedicated measurement modules to collect flow metrics that can be utilized for various other applications. Nevertheless, this approach is orthogonal to the detection of a range of attack patterns of a specific DDoS attack type. As an effort to iron out the challenges with large flows in high-speed networking environments, this work focuses on developing a universal detection framework targeting 100GbE networks.

This chapter presents FLARE, an FPGA-based large flow detection engine for online monitoring of network flows on high-speed networks, capable of handling data rates up to 200Gbps. Unlike conventional universal sketches that only integrate flow measurement units, FLARE seamlessly integrates complete detection units to function as a comprehensive detection system. Moreover, the application of FLARE is not just confined to large flow detection but to all kinds of network attack detection beyond the scope of large flow detection.

6.2 Challenges in the design of a comprehensive detection system

Deploying a large flow detection unit to precisely detect all attack patterns is not straightforward, and it requires multiple detection algorithms to cover the whole range of attack patterns. Resolving the challenges associated with the hardware deployment and simultaneously designing an accurate and comprehensive large flow detection system involves a multitude of related problems, which will be discussed in the subsequent sections.

6.2.1 The need for dedicated algorithms for comprehensive flow monitoring

Detecting large flows presents a multi-dimensional challenge involving the detection accuracy and the detection range of attack patterns. It often requires dedicated detection algorithms to handle different types of attacks. Large flow detection algorithms usually set a detection threshold, and any flows that exceed this volume threshold are identified as large flows. The measurement period and the threshold set play a role in determining the accuracy of detection algorithms, as discussed in Section 2.5.2.

Apart from the measurement period and volume threshold, the packet sending rate and packet sending volume also play a major role in determining detection accuracy. The existing state-of-the-art algorithms are designed for detecting large flows that are 10 to 1000 times larger than the allocated bandwidth and have a steady sending rate. While detecting flows with a high volume threshold, a certain measurement error is tolerable, making simple measurement architectures such as sketches the preferred data structure for designing the algorithms as the memory requirements are low. Although sketch-based detection algorithms produce significant overestimation, such overestimation is tolerable for detecting flows that exceed a high volume threshold. Simple sketches as measurement modules also keep the complexity of the algorithm to a minimum. However, a high detection threshold can create a vulnerability that attackers can exploit. The attacker can continuously send malicious traffic at a rate slightly lower than the detection threshold, effectively operating in stealth mode and evading detection. These attacks go undetected, allowing the attacker to successfully bring down the target without being detected. Setting a low threshold in sketch-based algorithms might cause false positives due to the overestimation. In short, it requires multiple dedicated algorithms even for tackling steady-rate large flow attacks.

Another drawback with the algorithms that can only detect high-volume flows is that they are designed for detecting flows that send data at a steady rate. Altering the measurement period to detect flows that send packets at a non-steady rate is a challenging problem for these algorithms. Certain types of attacks, such as burst-flood attacks, involve the periodic transmission of large data bursts within a short timeframe [216]. These attacks are difficult to detect due to the unpredictable timing and duration of the bursts. The algorithms designed for detecting high-bandwidth and steady-rate flows can cause the average measured volume of packets over a longer measurement period may fall well below the detection threshold while detecting burst-flood attacks. Reducing the measurement period to catch these bursts may result in false positives for flows that send data at a steady rate.

Therefore, detection algorithms for such non-steady-rate attacks or burst attacks require meticulous design and precision. For detecting flows with a volume slightly higher than the allocated bandwidth or flows that send packets at a non-steady rate, the estimation error of the measurement module must be negligible. High estimation errors in sketches due to overestimation can amplify the number of false positives in cases where the flow rate is slightly higher than the allocated bandwidth. Hence, simple sketch-based algorithms are not preferable. Accurately detecting such flows may require additional computational logic and high memory requirements to minimize estimation errors.

Recently, researchers have directed their efforts toward identifying flows that slightly exceed the allocated bandwidth. Scherrer et al. [217] proposed LOFT, a method capable of detecting significant flows that are only $1.5\times$ the magnitude of the allocated bandwidth. These flows are termed as *low-rate overuse* flows, and their detection holds great importance in ensuring the integrity of bandwidth-reservation systems. Moreover, algorithms like LOFT eliminate the requirement of distinct detection algorithms to detect low-rate and high-rate overuse flows. Even though LOFT is more sophisticated than simple sketch-based approaches, it would still be more profitable than deploying multiple detection algorithms to detect flows with varying magnitudes. ALBUS [216] is another precise detection algorithm designed for detecting burst-flood attacks. Such accurate detection algorithms are less tolerant of estimation errors in contrast to the approaches that use probabilistic measurement architectures such as sketches as the sole measurement architecture.

While an approach like LOFT is capable of detecting flows that send packets at a steady rate, whether low-rate or high-rate overuse flows, the need for dedicated algorithms such as ALBUS cannot be overwritten as LOFT cannot detect burst-flood attacks. Relying solely on a single detection algorithm may not be sufficient to detect a wide range of attack patterns. However, deploying

different algorithms as independent detection modules on hardware is not ideal due to limited hardware resources and strict latency requirements for packet processing at line rate.

6.2.2 Challenges in deploying individual detection algorithms on hardware

Deploying multiple detection units for large flows is suboptimal due to the increased resource requirements and sophisticated coordination, especially when it comes to processing high-speed data streams where speed is of utmost importance. When considering complete detection units involving blacklisting as the mitigation policy (cf. Chapter 2, Section 2.5), each unit requires individual blacklists. Having multiple blacklists calls for more memory. A significant factor that affects detection accuracy is the effective management of resources on FPGA platforms because of the limited availability of resources, especially the on-chip memory. The accuracy of existing large flow detection algorithms solely depends on the accuracy of the measurement module. When constrained by limited memory, sketches tend to produce significant overestimations, thereby limiting the overall detection accuracy.

Additionally, different algorithms may identify the same flow as malicious, resulting in multiple instances of the same flow ID in the respective blacklists, affecting memory efficiency. Moreover, each individual blacklist must be queried for every incoming flow. In high-speed accelerator platforms like Alveo [246], 64 bytes of data are received in every clock cycle, equivalent to one packet assuming the minimum packet size. Querying each individual blacklist increases the latency, which is crucial when it is required to process a packet in every clock cycle. Moreover, coordinating different algorithms becomes cumbersome due to their varying operating frequencies and query latency. The overall system speed is contingent on the slowest detection unit, and leveraging the parallel computing and pipelining capabilities is hindered by such disparities in the individual units.

6.3 Background on universal flow monitoring

Real-time detection of large flows in high-speed networking environments is a challenging research problem considering the detection speed and storage constraints. Although sampling-based algorithms [219, 85, 132] are commonly employed for large flow detection, they are not ideal for high-speed networks because of their high estimation error and loss of information [131, 164]. On the

other hand, sketch-based algorithms [253, 225, 251] are proven to have superior performance in terms of resource efficiency and speed. However, as mentioned earlier, most sketch-based algorithms are designed to detect flows that are 10-1000 times larger than the allowed bandwidth, calling for a sophisticated and precise design to handle various attack patterns [217, 216]. A universal framework to address the detection of all kinds of large flow attack patterns is the need of the hour.

There exist certain approaches that focus on a unified framework to handle various measurement tasks, such as OpenSketch [254]. The introduction of universal sketch-based flow monitoring units such as UnivMon [164] and LUS [243], which are more versatile than OpenSketch, shed light on a new direction of this challenging research problem. However, the universality of these approaches is based on collecting flow metrics/statistics by making use of the measurement modules dedicated to other specific applications, such as heavy-hitter detection. While these flow metrics or traffic statistics can be used in various other applications, such an architecture may not effectively serve the purpose of a complete detection module for identifying all attack patterns associated with a specific attack type. The reason behind this limitation is that UnivMon and LUS employ sampling and generic sketch algorithms, such as Count Sketch [79], which are simpler but more error-prone than the dedicated algorithms for the precise detection of a range of attack patterns, as discussed earlier. Univmon and LUS are also targeted for passive monitoring on programmable switches.

Moreover, these universal approaches incur high latency and computational complexity due to their hierarchical structure and parallel copies of estimation algorithms. For instance, the average processing overhead of UnivMon is 33 hash operations and 32 memory accesses per packet due to its hierarchical sampling, which is highly undesirable [243]. Consequently, this hinders the application of these approaches in monitoring high-speed data streams at line rate. Augmented Light-weight Universal Sketch (ALUS) [243], a version of LUS, proposes a more accurate detection of heavy-hitters. However, its coverage remains restricted to top-k heavy-hitters and does not include other large flow attack patterns. Other approaches like MV Sketch [225] propose a single sketch-based system for detecting multiple abnormal patterns, such as heavy-hitter and heavy-change detection. Nevertheless, MV Sketch is designed for identifying a specific pattern of large flows and is not a scalable system that can incorporate multiple attack patterns.

Presently, no such hardware frameworks exist for universal network flow monitoring that can instantiate multiple detection units and still minimize the overall hardware resource and memory requirements than all of them operating individually. When it comes to the use of high-speed hardware platforms for

large flow detection, only a limited amount of work is available on exploring the possibilities of data center accelerator cards, such as Alveo [246], even for network security applications. Recently there have been promising trends toward FPGA-based data center accelerators [235, 107, 72]. EasyNet [129] and VNx [245] propose 100Gbps network stacks targeting Alveo. Wand et al. [234] propose an FPGA-based smart network interface card (NIC), and AMD introduces an OpenNIC [2] on Alveo. However, the application of these NICs and network stacks are mostly directed toward data analytics, machine learning, and cryptography [83, 142, 141, 68]. The application of such platforms in large flow detection is very much unexplored. Nevertheless, the availability of the existing works, such as the NICs and Ethernet stacks, are very much helpful for any networking applications, including large flow detection, as reusing these architectures helps save a tremendous amount of design time.

6.4 Proposed detection framework - FLARE

This work proposes FLARE (FPGA-based large flow detection engine), that can integrate multiple individual detection algorithms to serve the purpose of a full-fledged large flow detection framework for Terabit Ethernet. FLARE targets high-performance data center accelerators for its deployment, and explores a different research direction than the universal sketches mentioned earlier. The term *universal* in FLARE indicates the versatility of the framework. This versatility allows the complete detection of any specific network attacks and their diverse attack patterns by integrating dedicated detection units. In contrast, the existing universal sketches, such as UnivMon and LUS, focus on eliminating the need for custom sketches for custom applications. The main contributions of FLARE are:

- A detection framework that can effectively integrate and coordinate distinct and standalone detection units to detect a wide range of attack patterns, such as flood attacks, elephant flows, pulse wave/burst attacks, and high-rate/low-rate overuse flows.
- A high-speed architecture targeting data center accelerator cards that can effectively handle online processing of incoming data streams on Terabit Ethernet, achieving data rates up to 200Gbps.
- A shared architecture that facilitates the sharing of blacklists among detection units, thereby minimizing the memory requirements and eliminating the possibility of redundant entries in the blacklist. Having a shared blacklist makes it easier to coordinate the functionality, as the query and configuration of the blacklist do not disrupt the normal operation of detection units.

- A control and management interface with which the user can test and verify the functionality, examine the blacklist, and add entries to the blacklist manually.

System architecture of FLARE: Figure 6.1 depicts the system architecture of FLARE. Two primary elements of FLARE are a detection framework and the blacklist module. A parser, traffic filtering unit, and other functional and control logic responsible for passing/dropping the frames, along with the detection framework and blacklist module, form the complete detection engine. The detection framework includes the detection algorithms and the control and management interface, where each independent detection algorithm is termed as a *brain*. FLARE is designed to function as a bump-in-the-wire, placed between the outside world and the network to be protected. The incoming traffic is directed to FLARE, and FLARE forwards the legitimate traffic to the network. Further elaboration on the individual components of FLARE is provided in the subsequent sections.

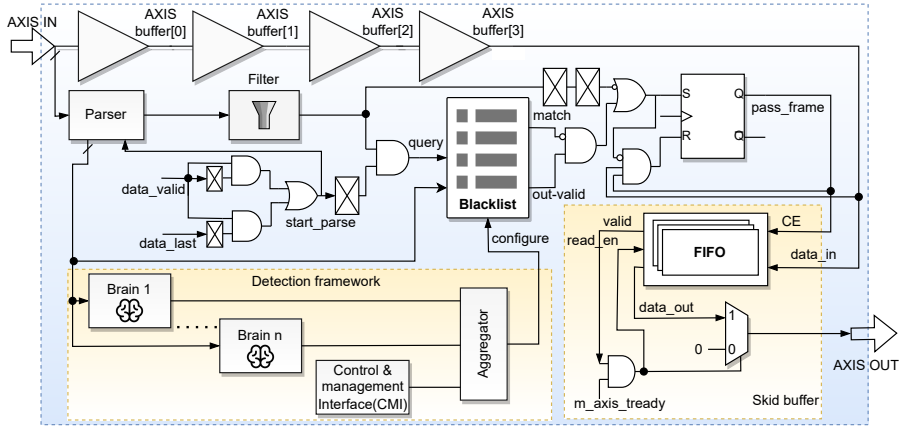


Figure 6.1: System architecture of FLARE

6.4.1 Parser and Network filter

The flow ID and various attributes, including flow size, EtherType, and protocol, are extracted by the network parser from the incoming flow. Using these flow attributes, the network filter identifies and directs the flows that satisfy the filtering criteria to the blacklist and detection framework. In FLARE, the filtering criteria are based on the EtherType and IP protocol. The flows with

$EtherType = IPv4$ and $Protocol = TCP/UDP$ are inspected. If the flows are not blacklisted, they are allowed to pass to the network. All other flows that do not adhere to the filtering criteria are allowed to pass without any inspections. The filtering policy is modifiable and can be changed during run-time using the control and management interface in the detection framework.

6.4.2 Detection framework

The detection framework is the core of the detection engine, consisting of the detection algorithms and the control and management interface (CMI). Each detection algorithm is referred to as *brain*, and multiple brains can be included. The CMI is an AXI-Lite module that enables the control and manipulation of specific operations, such as adding a flow ID to the blacklist and managing the filtering policy. The aggregator coordinates the interactions of the brains and the AXI-Lite module with the blacklist. A detailed description of these modules is provided in the following sections.

Brain

The brain, or the detection algorithm, decides whether an incoming flow is malicious or not. Multiple brains can operate simultaneously, each performing a specific detection task. Different detection algorithms employ distinct measurement strategies and operate independently; for instance, LOFT [217] uses sketch-based measurement, whereas EARDet [241] uses associative array-based measurement. Although individual brains function as independent detection modules that share a common blacklist, utilizing a shared counter array could minimize memory utilization. FLARE encourages the design of detection algorithms that share a common counter architecture, thereby further minimizing hardware resources. We incorporate the LOFT algorithm as the brain in the proof of concept implementation of FLARE (cf. Section 6.5). Since CMI also enables the manual addition of flow IDs to the blacklist, it effectively functions as a user-controlled brain.

Control and management interface (CMI)

The control and management interface (CMI) is an AXI-Lite module responsible for overseeing the control and validation of the detection framework and network filter. Additionally, the control unit facilitates testing of the detection framework and serves as a human interface for accessing or verifying the content of the blacklist. CMI uses a 32-bit control register and can have a number of data

and status AXI slave registers. These registers facilitate reading and writing to the blacklist, as well as retrieving data from other modules to verify their functionality. By examining the blacklist using CMI, one can determine if a specific flow ID is present and ascertain which brain was responsible for adding the flow ID to the blacklist. The bits in the control register are used to modify the filtering policy. CMI is also considered a brain in our implementation, as it is capable of adding entries to the blacklist.

Aggregator

The aggregator is the interface connecting the blacklist and the brains. Multiple brains operate simultaneously and can send the detected malicious flows to the blacklist to be written. The aggregator only requires a flow ID, and its operation is independent of the detection mechanisms and attack patterns. Whenever a brain needs to add an entry to the blacklist, it sends a *write enable* signal along with the corresponding flow ID, which remains high for a single clock cycle. However, if the blacklist receives multiple write requests from different brains while a previous write operation is still ongoing, it fails to add the malicious flow IDs to the blacklist, which is an undesirable outcome. The aggregator is a smart multiplexer connecting each brain to the blacklist, eliminating data loss. The hardware architecture of the aggregator is shown in Figure 6.2.

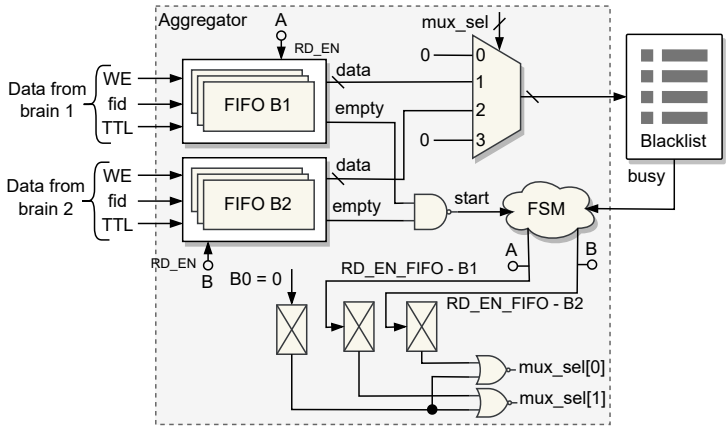


Figure 6.2: Aggregator

The aggregator architecture presented in this section is capable of handling three brains and can be easily modified to accommodate more. The data received from each brain is directed to their respective data FIFO queues. The *empty* signals from the FIFOs indicate the presence of pending write operations, triggering

the *start* signal for the finite state machine (FSM). After the blacklist completes the ongoing write operation, the data from the FIFOs is sent to the blacklist based on a predefined priority. The FSM manages this process by controlling the *read enable* signals of the FIFOs and a multiplexer responsible for directing data from the FIFOs to the blacklist. In the figure, brain B1 corresponds to the LOFT algorithm and brain B2 corresponds to the CMI, connected to input pins 1 and 2 of the multiplexer, respectively. Brain B0 is connected to pin 0 of the multiplexer, but since B0 is absent, pin 0 is tied to 0. The multiplexer selects input pin 3 when the FIFOs are empty, which is also tied to 0.

6.4.3 Blacklist module

The blacklist module consists of a shared blacklist and the associated *time-to-live* (TTL) logic. The components and functionalities of the blacklist module are explained in the subsequent sections.

TTL logic

The TTL logic helps to distinguish the flow IDs written to the blacklist by different brains. Each algorithm may have a distinct measurement period or epoch, resulting in varying reset periods. While sharing a common blacklist, it is important to differentiate the entries in the blacklist based on the brains that wrote them in order to determine their expiration. The expiration of each entry depends on the reset period of the corresponding brain. To address this issue, a TTL value is associated with each flow ID. This TTL value specifies the expiry time of a blacklist entry with respect to a timestamp.

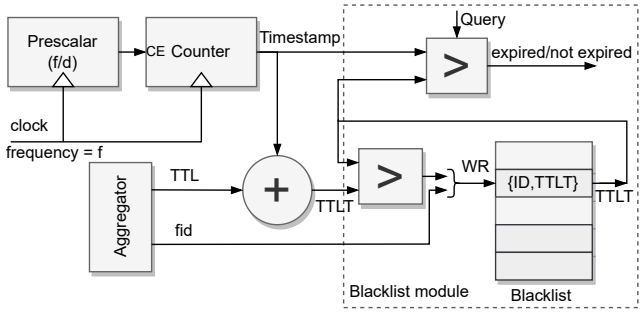


Figure 6.3: TTL logic

Figure 6.3 illustrates the logic circuitry employed in TTL logic. A combination of a prescaler, counter, and an adder generates a value referred to as TTLT (Time-

To-Live Timestamp), which is the sum of the current timestamp and the TTL value. This value is then stored alongside the flow ID in the blacklist. It is to be noted that the blacklist does not store the entire flow ID but only an 8-bit fingerprint. The TTL value is specific for each brain. The TTL logic receives the TTL value and the flow ID from the aggregator. The prescaler is a clock divider, and it drives a 16-bit counter that produces the timestamp. In this work, the size of the TTLT value is taken as 16 bits. The divisor to the prescaler is calculated based on the clock frequency and the size of the TTLT value. The calculation is carried out in a manner that ensures the sum of the total runtime of the brain and the TTL value fit within 16 bits.

Blacklist

The blacklist is implemented using SPArch-based content addressable memory (CAM). The architectural features of SPArch are detailed in Chapter 5. SPArch-based CAM can be used as a probabilistic key-value store, a perfect alternative to content addressable memories. This functionality of SPArch is used in this work to construct the blacklist. The architecture of the blacklist is shown in Figure 6.4. It consists of two main components: a sketch and an associated memory unit to store the TTL values. The sketch component stores the fingerprint of the flow ID and the corresponding address of the memory location. To inspect which brain added a specific flow ID to the blacklist (with the assistance of CMI), an additional identifier tag field of 1 or 2 bits (depending on the number of brains) can be added to the TTL memory.

The flow ID is mapped to the sketch using hash values h_1 to h_d, where d is the number of hash values. Xoodoo-NC is the hash function used in SPArch. The hash output is split to generate the hash values and the fingerprint. SPArch only stores an 8-bit fingerprint of the flow ID. SPArch is extremely memory efficient and faster than any other associative array-based key-value store architecture like CAMs. Other probabilistic data structures, such as the Bloom filter and Cuckoo filter, cannot provide the functionality of a CAM or a key-value store, making them unsuitable for implementing the TTL logic.

The blacklist module receives the TTLT value from the adder and is written to the blacklist. If the flow ID already exists in the blacklist (previously written by another brain), the existing TTLT value is read and compared with the incoming TTLT value. If the incoming value is greater, it replaces the existing value in the blacklist. The *start_parse* signal (refer Figure 6.1) indicates the beginning of the parsing operation of a legitimate flow. A blacklist query is performed for every incoming flow based on the *start_parse* signal if the filtering criteria are met. During a blacklist query, the TTLT value is read if the incoming flow is

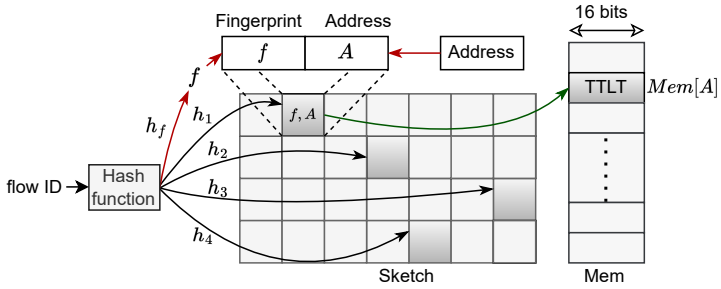


Figure 6.4: Architecture of blacklist

present in the blacklist. The TTLT value is then compared against the current timestamp. If the current timestamp value is greater, the blacklist entry is expired, and the flow is allowed to pass. If the entry is expired, it is removed from the blacklist, and the address of the entry is added to a FIFO. During the next write operation, the address from the FIFO is used to store the blacklisted entry. If required, the blacklist can be reset periodically depending on the reset periods of the brains. The user can also perform a query operation using the AXI-Lite CMI to inspect the contents of the blacklist. This user query takes precedence over the other queries.

Handling fixed/static blacklist entries: Typically, the blacklist undergoes periodic resets, erasing all its entries, including the manually added fixed blacklist entries. Users need to rewrite these fixed entries after each reset. There are two potential options to handle such fixed entries: either allocating an initialization period after every reset or maintaining a separate blacklist specifically for fixed entries. The former is viable if the number of fixed entries is limited, while the latter is more advantageous if there is a substantial number of fixed entries that could induce comparable delays in writing. This separate fixed-entry blacklist can be positioned outside FLARE, allowing incoming network traffic to be screened through it before passing to FLARE.

6.4.4 Skid buffer

A skid buffer is implemented using data FIFOs. It is employed to buffer the outward packets until the *ready* signal of the master AXIS interface is high and to address the clock domain crossing. The $pass_frame$ signal, generated by the SR flip flop, acts as the *chip enable* (CE) signal for the FIFOs, and once it is high, the incoming data from the AXI Stream (AXIS) buffer stages are

written to the FIFOs. If the FIFO contains valid data and there is an active *ready* signal from the master AXIS interface, the *read enable* signal is held high, and the packet is read and transmitted. The *reset* signal of the SR flip flop ensures that the $pass_frame$ signal is held high for only a single packet, guaranteeing that the packets are transmitted out only based on the output from the blacklist.

6.5 Proof of concept

A proof of concept of FLARE is implemented to test the feasibility. The implementation consider the LOFT [217] algorithm and CMI as the brains, where LOFT plays the role of the detection algorithm and CMI functions as a user-controlled brain. The detection algorithm is chosen based on the mitigation policy: the blacklisting model. LOFT adds large flows to a blacklist, and incoming flows are processed only if they are not present in the blacklist. Even though EARDet [241] also follows the blacklisting model, it employs associative array-based counter arrays, which are inefficient on hardware. Furthermore, LOFT exhibits better accuracy compared to EARDet. To the best of our knowledge, LOFT is the only algorithm that can precisely detect large flows having sending rates as low as 1.5× larger than the allowed bandwidth. The sophisticated design and the architecture of LOFT as a complete detection unit prompt us the use LOFT to showcase the efficiency of FLARE. While other sketch-based algorithms are generally less complex than LOFT, they are not as precise as LOFT, and incorporating them into FLARE may not sufficiently challenge the efficiency of FLARE. The other components of FLARE are as discussed in Section 6.4. The implementation and hardware evaluation setup and analysis of FLARE are discussed in detail in this section.

6.5.1 Experimental setup

The experimental setup consists of a testbed as depicted in Figure 6.5. Alveo U250 (cf. Section 2.1.5) is the hardware platform used for the implementation. FLARE is embedded into the dynamic region of Alveo U250 as an AXI4-Stream IP and, along with the CMAC kernel, using Xilinx Vivado 2022.1. We used the implementation infrastructure of XUP Vitis network example (VNx) [245] as the reference model for our implementation. The CMAC kernel is reused from VNx, which uses the Ultrascale+ integrated Ethernet subsystem. The CMAC kernel offers two 512-bit AXI4-Stream interfaces to connect with FLARE. The operating frequency of the Ultrascale+ integrated Ethernet subsystem is 322 MHz. The VNx CMAC kernel incorporates clock-domain-crossing (CDC)

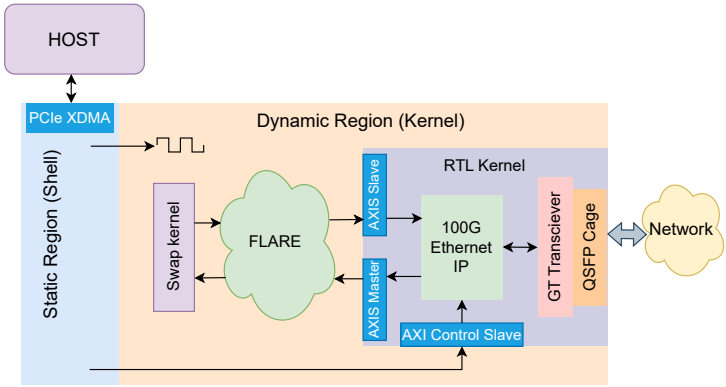


Figure 6.5: Evaluation testbed

logic to facilitate the conversion from the kernel clock to the 100G Ethernet Subsystem clock. An AXI4-Lite interface is also available in the CMAC kernel to inspect the CMAC statistics. The device is connected to the network using a QSFP cage, and the platform communicates with the host computer through PCI Express. FLARE is interfaced with another AXI4-Stream IP named *Swap kernel*, which echoes the traffic back after swapping the source and destination addresses and ports. This allows us to test and verify the functionality of FLARE.

6.5.2 Implementation details

The hardware architecture of the overall system of FLARE is as depicted in Figure 6.1. The 512-bit AXI Stream data received is buffered for four cycles, which is the processing delay of the query data path of the blacklist. The query data path involves parsing, filtering, and blacklist query time. The parsed data is also forwarded through the detection data path, which involves the detection framework. The detection framework contains the LOFT algorithm and the CMI as the brains, along with an aggregator that facilitates communication between the blacklist and the detection framework.

The counter arrays have 16384 counters, with each counter having a width of 24 bits. Out of 24 bits, 16 bits are dedicated to the flow volume and 8 bits to cardinality, respectively, assuming that there are no *jumbo* frames. The active flow list (AFL) and flow table (FT) are implemented using the Cuckoo hash table, and both have a depth of 16384. Each incoming element to the Cuckoo hash table is mapped to two buckets, with each bucket having two

entries. The watchlist, which has a size of 64, is a sorted list that is dynamically sorted during each entry. FIFOs of size 1024 are used between each module to prevent data loss, as each module has different processing times. The CMI, which acts as the second brain, is an AXI-Lite module with eight slave registers of size 32 bits. One slave register serves the role of a control register, which is used to manually read/write the blacklist and change the filtering policy. The hardware architecture of the aggregator is as described in Section 6.4.2. The depth of FIFOs in the aggregator is chosen as 16, which is sufficient enough as the blacklist write occurs rarely.

The hardware architecture of the blacklist aligns with the description provided in Section 6.4.3. The blacklist is implemented using dual-port block RAM (BRAM), controlled by two FSMs that allow simultaneous query and configure operations in the same clock cycle. The SPArch-based CAM is capable of handling a query in every clock cycle, which is required to maintain pace with the processing at line rate. The update operation of the blacklist is not pipelined to conserve resources as well as to improve the operating frequency. As the update operation occurs less frequently, the aggregator can store and queue multiple update requests, causing the pipelining of the update operation unnecessary. The depth of the TTL memory and the sketch module of the blacklist is taken as 1024. The number of sketch arrays (d) is taken as 2, adequate to provide near-hundred percent accuracy. The fingerprint size is 8 bits, and the address size is 10 bits. Xoodoo-NC is the hash function employed, and the output of the hash function is split to obtain the required hash values. Coming to the skid buffer, the FIFOs of the skid buffer have a depth of 512, which is large enough to meet the requirements. Even a depth of 64 would be sufficient for the skid buffer under normal operating conditions.

Since LOFT is the core component of FLARE that facilitates large flow detection, detailed hardware implementation of LOFT is provided in the following section.

6.5.3 Hardware implementation of LOFT

LOFT is a lightweight detection algorithm capable of detecting low-rate overuse flows faster than any other existing approaches. LOFT adheres to stringent constraints related to time and memory. It can effectively reduce counter noise with the help of a multi-stage strategy, taking both traffic volume and flow count in any counter and aggregating these metrics over time. LOFT also enables high-speed packet processing, thanks to the fewer operations required per packet compared to conventional approaches.

Architecture of LOFT

Figure 6.6 illustrates the architecture of the LOFT. The complete LOFT detection unit comprises a *Classifier*, *Blacklist*, *Probabilistic Overuse Flow Detector (OFD)*, and a *Precise monitoring* component. The Classifier extracts the flow ID from the packet (pkt) header and ascertains its permitted bandwidth. The size of the flow ID, consisting of the source address and port, is 48 bits. If the incoming packet is not present in the blacklist, it is passed to LOFT OFD for further analysis to identify the suspicious flows. The *Precise Monitoring* component scrutinizes the list of suspicious flows provided by the LOFT OFD and decides which flows must be added to the blacklist. In our implementation, the Classifier and the blacklist are shared among all the detection algorithms that are in use.

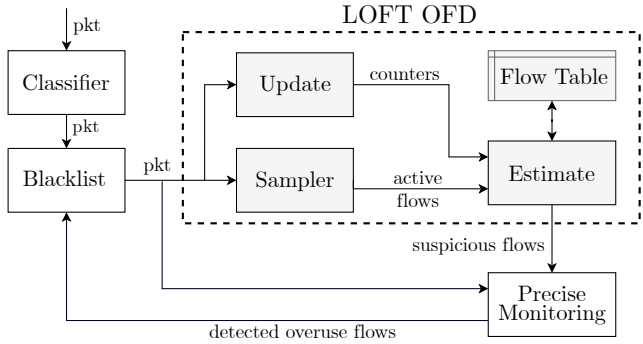


Figure 6.6: Architecture of the system model of LOFT. Source: [217]

The operation of LOFT is bounded by specific time intervals: a shorter time period called the minor cycle and a larger time frame called the major cycle. The measurement epoch of LOFT is divided into several major cycles, and each major cycle is further divided into several minor cycles. In our implementation, there are four major cycles, each lasting 250 ms. Each major cycle is divided into sixteen minor cycles, each lasting 15.625 ms. The *Update* algorithm of LOFT OFD operates within a minor cycle and aggregates the individual flows based on the packet size. When the minor cycle changes, an alternative set of counters is used. The sampler creates a list of active flows using a predefined sampling rate and stores it in the *active flow list* (AFL). The *Estimate* algorithm of LOFT OFD operates within a major cycle. With the help of the active flow list, the estimate algorithm analyzes the counter arrays and aggregates the flow volumes received from the *Update* algorithm after each major cycle. The aggregated bandwidth utilization of each flow is updated in the flow table. The *Estimate* algorithm then utilizes the data in the flow table to determine

a watchlist of suspicious flows. The watchlist is passed on to the *Precise Monitoring* component at the end of each major cycle to identify large flows and update the blacklist. For a more comprehensive explanation of LOFT, we refer to the original paper [217].

Implementation details

The LOFT algorithm is implemented with specific optimizations to enhance hardware efficiency. The core of the LOFT algorithm is the overuse flow detector (OFD). The main components of OFD are the *update* and *estimate* algorithms. Figure 6.7 depicts the hardware block diagram of LOFT, with a detailed look into the architecture of *active flow list* (AFL).

The update logic: The update logic controls the operations of the AFL and the update counters. There are two instances of the AFL table employed in the update logic in our implementation. During a major cycle, one instance is updated while the other is queried. After each major cycle, their roles are swapped, meaning the one that was updated becomes the one queried, and vice versa. The signal *active_table* indicates which AFL table is currently being updated in the ongoing major cycle. The flow ID (*fid*) is passed on to the sampler and update counter arrays. Based on the sampler output and *active_table* signal, one of the AFL tables is updated, and the other is queried.

The update counter arrays receive the incoming flow ID and flow volume and update the corresponding counters. The counter array used in LOFT is a CM sketch that maps each flow to a single counter. The update counter arrays also include counters for storing cardinality. The flow volume counters are 16 bits in size, while the flow cardinality counters are 8 bits in size. The architecture of the update counter arrays and the estimate logic is illustrated in Figure 6.8. The update counter arrays employ two hash functions, one for update and one for query operations. Xoodoo-NC is the hash function used. The number of minor cycles used in this implementation is denoted as z, and the number of major cycles used is denoted as j. The hash functions are randomized for each minor cycle, achieved by generating a hash output H of size $z \times m$ and then splitting it into z hash values. Each minor cycle uses a predefined set of m hash bits; for instance, the first minor cycle uses $H[0 : m-1]$, and the second minor cycle uses $H[m : 2m-1]$, and so on.

The hardware implementation utilizes a modified approach for updating counters compared to the original LOFT algorithm. The original LOFT algorithm uses on-chip memory during the *Update* operation, and the counter array is copied to the off-chip memory at the completion of the minor cycle. Such a design is much more profitable in terms of on-chip memory consumption. However, accessing

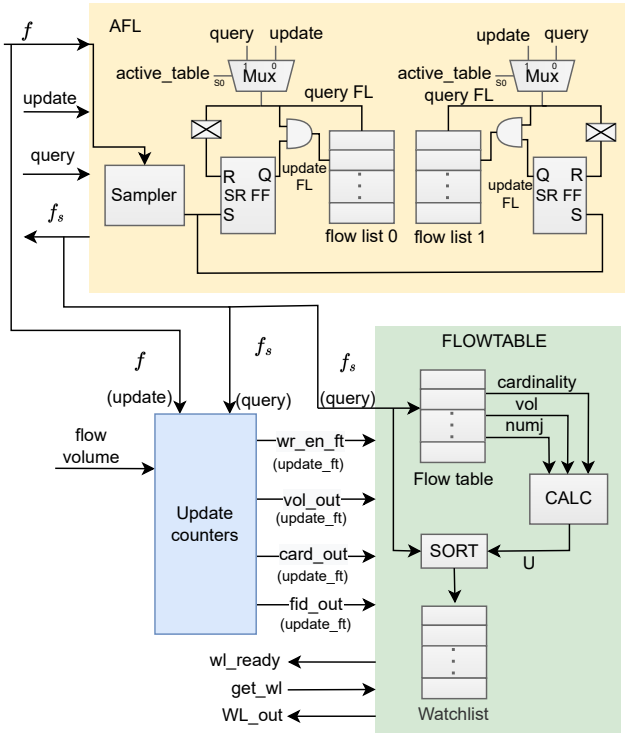


Figure 6.7: LOFT architecture with a detailed look into Active Flow List

the external memory on hardware results in additional routing delays and a significantly higher number of clock cycles [127]. It also results in higher power consumption. Moreover, the *Estimate* algorithm has to process the data from all the minor cycles altogether at the end of the major cycle while the subsequent major cycle is already in progress. This would require some bookkeeping to keep track of the data from different major and minor cycles, which is inefficient in hardware. The delays associated with external memory access could also cause the operational time of the estimate algorithm to exceed the major cycle when the duration of the major cycle is short. Such delays and low operating frequencies are unacceptable, especially when operating with Terabit Ethernet. To address these concerns, we implement counter arrays entirely in the on-chip memory. Moreover, this also helps to analyze the worst-case scenario in FLARE regarding the on-chip memory requirement.

However, storing all the counter arrays in the on-chip memory requires a different strategy to efficiently allocate resources. Since the counter arrays need

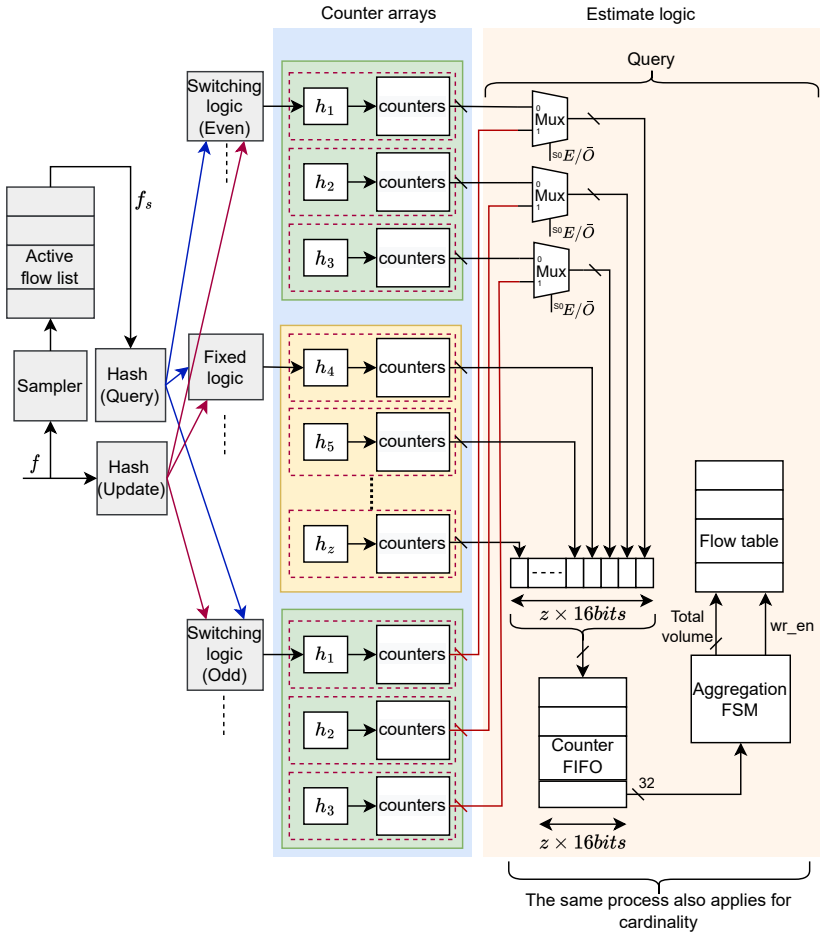


Figure 6.8: Update counter array and estimate logic of LOFT

to be read before being overwritten in a major cycle, keeping two instances of counter arrays is necessary - one is read while the other is updated in a major cycle. However, keeping two separate counter arrays would introduce considerable memory overhead. To overcome this challenge, we have optimized the architecture, as depicted in Figure 6.8.

The counter array architecture in this approach is designed to replace the usage of two instances of minor cycle counter arrays with a more efficient setup. Instead of keeping two instances of minor cycle counter arrays, it involves three groups of arrays: one *fixed* array group and two *switching* array groups. During

a major cycle, one switching array group is updated with the incoming flow data while the other is read by the estimate logic. The switching array group consists of three counter arrays, and the update time is set to three minor cycles, providing ample time to complete the operations of the estimate algorithm. The estimate algorithm reads and aggregates all z minor cycle counter arrays during the update time of the switching counter arrays and leaves the fixed counter arrays ready for the update in the same major cycle. In the subsequent major cycle, the switching counter arrays that are read in the previous cycle perform the update operation while the other set of switching arrays is read. In this work, the number of minor cycles is 16, which means that this approach significantly reduces the total number of required counter arrays from 32 to only 19. The switching arrays are labeled as *Even* and *Odd* switching arrays. The *Even* switching arrays are updated when the major cycle count is even, and the *Odd* switching arrays are updated when the major cycle count is odd. This arrangement allows for an efficient and streamlined counter array management system.

The switching and fixed array logic implementation is shown in Figure 6.9. The array select logic responsible for the fixed counter arrays sets the update signal for the counter array when both *update* and c_z signals are high. The c_z signal represents the one-hot encoding of the minor cycle count, and $c_z[i]$ indicates the one-hot bit corresponding to the ith counter array. The query signal is directly connected to the fixed counter arrays. The switching logic also includes an $E/\bar{O}$ signal, which indicates whether the minor cycle count is even or odd. The odd or even switching counter arrays are chosen based on the $E/\bar{O}$ signal.

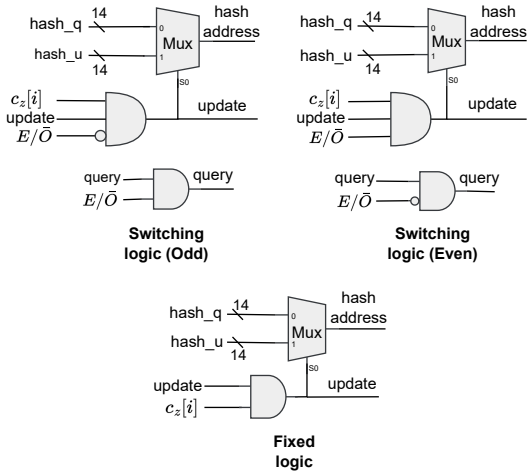


Figure 6.9: Switching and fixed counter array logic

The estimate logic: The estimate logic queries the counter arrays, and the flow volumes are aggregated and updated to the flow table. During the estimate operation, the flow ID f_s is read from the AFL, and the counter arrays corresponding to f_s are searched to retrieve the volume and cardinality. Depending on whether the major cycle count is even or odd (indicated by the $E/\bar{O}$ signal), one of the switching logics is selected for the query. The update algoritm provides the flow volume (vol_out), cardinality ($card_out$), flow ID (fid_out), and write enable signal (wr_en_ft) to the estimate logic. A timeline of LOFT summarizing the update and estimate operations is depicted in Figure 6.10. The timeline displays major cycles divided into z minor cycles. During the update operation, counter arrays in the minor cycles are updated. The switching and fixed counters arrays are alternatively updated and queried during each major cycle. During the estimate operation, the flow table is updated, and the flow volumes are analyzed to find the group of overuse flows.

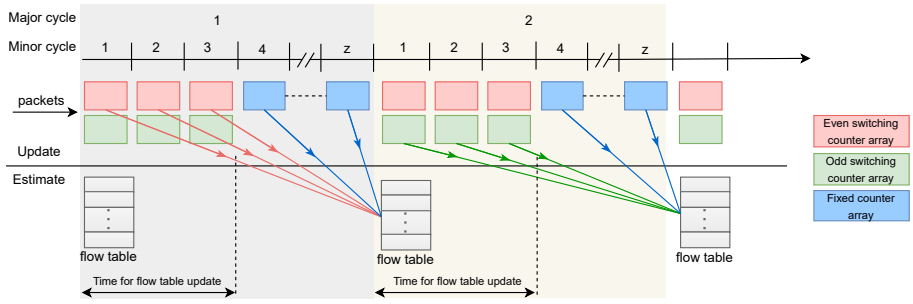


Figure 6.10: A timeline of LOFT

The flow volumes from all z counter arrays are stored in a FIFO of input width $z \times 16$ bits. Similarly, the cardinalities are stored in another FIFO of input width $z \times 8$ bits. The output size of the volume and cardinality FIFOs are only twice the counter and cardinality sizes, 32 bits and 16 bits, respectively. The *aggregation* finite state machine (FSM) enables the FIFO read signal to read the content of the FIFO. The aggregation FSM receives 32-bit output from the volume FIFO, consisting of two 16-bit counter values. The FSM performs the aggregation operation, and the result is stored in a temporary register. The total flow volume is updated in the flow table once the aggregation is performed for all z values. The flow table is a Cuckoo hash table that stores the total flow volume, cardinality count, flow ID, and $numJ$ value. The $numJ$ value represents the cumulative count of major cycles during which the corresponding flow ID remains active. Figure 6.8 only illustrates the flow volume aggregation; the same process also repeats for aggregating cardinality.

Once all the flow IDs in the AFL are updated to the flow table, the estimate algorithm calculates a roster of suspicious flows for precise monitoring with the help of the calculation module represented as $CALC$ in Figure 6.7. This roster of suspicious flows is stored in a watchlist as a sorted list. The watchlist remains sorted, and after every major cycle, the watchlist is updated. The sorted list simplifies the process of determining whether to include new watchlisted flows in the watchlist after each major cycle. This decision can be made based on the flow with the lowest aggregate volume. The sorting logic involves a parallelized combinational logic unit that updates the sorted list in a single clock cycle for each incoming input denoted as U, where U is the flow volume corresponding to the flow ID f_s. The watchlist size is 64, and it takes 64 clock cycles to update the watchlist after the completion of the estimate algorithm. A watchlist ready signal (wl_ready) indicates that a sorted watchlist is available. By sending a get_wl signal, the watchlist can be retrieved for precise monitoring.

6.5.4 Implementation results

The resource consumption of the testbed implementation is shown in Table 6.1. It details the resource usage of different modules, including the CMAC core, blacklist, swap kernel, and LOFT. The overall resource consumption of the testbed includes the resources of the static shell region. The LUT and register usage of FLARE is only 1.2% and 0.3% of the available resources, respectively. Among the utilized FPGA logic, LOFT consumes the lion's share, which is still about only 1% of the available resources. Although LOFT consumes ≈20% of the on-chip BRAMs, it is to be noted that most of the memory usage of LOFT, excluding the counter array for the minor cycle and active flow list, can be shifted to off-chip memory if needed. Even with the full-fledged LOFT implementation entirely using BRAMs, it consumes only 20% of the available on-chip memory, leaving plenty of room to add other algorithms. The SPArch-based blacklist consumes negligible logic resources and memory compared to CAMs. A CAM of size 1024 requires ≈8× more LUTs and 160× more BRAMs than the SPArch-based blacklist of the same size [212]. Moreover, the operating frequency of CAM is ≈3× less than that of SPArch-based blacklist.

FLARE can achieve bidirectional data rates up to 200Gbps, which is the throughput the network interface can support. The operating frequency of the CMAC core is 322 MHz, and it can transmit/receive 64 bytes per clock cycle. Assuming a minimum packet size of 64 bytes, FLARE can receive a packet in every clock cycle. The overall operating speed of FLARE depends on the speed of the individual modules, specifically the detection algorithms, which could be the slowest modules. The blacklist module is amongst the fastest module in FLARE with an operating frequency greater than 350 MHz. The CMAC core

is equipped with clock-domain crossing, ensuring proper communication even in the presence of a frequency mismatch with the user logic.

Table 6.1: Resource utilization

Design	LUTs	Flip flops	BRAM	DSP
Testbed	**166646 (9.6%)**	**270704 (7.8%)**	**860 (32.0%)**	**9 (0.0%)**
FLARE	21006 (1.2%)	11761 (0.3%)	547 (20.3%)	0
Blacklist	810 (0.0%)	107 (0.0%)	2 (0.1%)	0
LOFT	17728 (1.0%)	9683 (0.3%)	532 (19.8%)	0
CMAC	9126 (0.5%)	32051 (0.9%)	17 (0.6%)	0
Swap kernel	686 (0.0%)	802 (0.0%)	17 (0.6%)	0

6.6 Conclusion

This chapter presents FLARE, a universal large flow detection engine on high-speed FPGAs. FLARE can integrate multiple large flow detection algorithms while sharing a common blacklist to serve the functionality of a complete detection unit. The functionality of FLARE is tested on an Alveo U250 data center accelerator card using high-speed network interfaces, supporting data rates up to 200 Gbps.

As a future work, the design of FLARE can be further enhanced by leveraging the platform-specific components of Alveo U250, particularly UltraRAM (URAM). URAM provides higher memory density than BRAM, requiring a smaller area and thereby reducing chip size and routing delays. Speed-critical applications that require a higher memory footprint, such as network flow monitoring, can benefit from the use of URAMs. With this consideration, exploring the replacement of BRAMs with URAMs in FLARE emerges as a promising option for future research.

Chapter 7

Conclusion

This chapter presents the conclusions and possible directions of future research of this thesis. This thesis focuses on designing hardware-efficient architectures for implementing large flow detection algorithms on FPGA targeting Terabit Ethernet. The primary objectives of this thesis can be summarized as follows: (1) Design hardware-efficient components and algorithms for large flow detection, (2) Implement large flow detection algorithms using the designed components, and (3) Design and implement a detection framework to integrate distinct detection algorithms. Section 7.1 summarizes the contributions made by this thesis towards achieving these objectives, while Section 7.2 explores potential avenues for future research.

7.1 Summary of contributions

This section provides a summary of all the contributions presented in this thesis. The first two chapters of this thesis are intended to provide the general audience with a basic understanding of the topic of interest of this thesis prior to diving deep into the objectives of the thesis. Chapter 1 provides a general introduction, highlighting the relevance of this work in the real world and emphasizing its importance. Chapter 2 introduces the basic concepts of FPGAs, network security, and large flow detection. The introduction to FPGAs provides the architecture, connectivity, and design flow of FPGAs and includes a concise comparison of FPGAs with other competing hardware platforms. Additionally, a detailed overview of computer networks, network security, and distributed denial of service attacks is provided. Finally, Chapter 2 concludes with an

exploration of existing research, challenges in large flow detection, and the implementation of large flow detection algorithms on FPGAs.

Hash functions are accountable for considerable computational overhead in large flow detection algorithms. Chapter 3 explores the design and implementation of hardware-efficient non-cryptographic hash functions tailored explicitly for networking applications. A technique to enhance hardware efficiency is proposed in this chapter wherein lightweight cryptographic algorithms are modified to perform the functionality of a non-cryptographic hash function. By limiting the state size and minimizing the number of rounds, the Xoodoo permutation is transformed into a non-cryptographic variant called Xoodoo-NC. Following this approach, eight other non-cryptographic hash functions are designed from lightweight cryptographic ciphers. These hash functions are proven to be efficient in hardware and outperform all other existing non-cryptographic hash functions.

The speed and memory requirement of lookup architectures plays a vital role in determining the ability of a detection algorithm to perform data processing at line rate. Chapter 4 investigates the design of hardware-efficient lookup architectures and presents the design of efficient Bloom filter architectures for fulfilling this purpose. Moreover, we have proposed the use of a single collision-resistant hash function instead of multiple independent hash functions to reduce the hardware complexity. Our proposed parallel Bloom filter design turned out to be faster and more resource-efficient than existing architectures. Furthermore, a comprehensive analysis of approximate membership query data structures is conducted based on their hardware compatibility. For this purpose, we have modified and implemented the Cuckoo hash table, Peacock hash table, and Cuckoo filter to serve the purpose of lookup and key-value stores. These implementations demonstrate significantly improved hardware efficiency compared to conventional architectures like CAMs.

Network flow measurement is the core of all network security applications and requires careful design to ensure accuracy as well as resource efficiency. Chapter 5 focuses on exploring the design of flow measurement architectures. The primary design parameters discussed in this chapter include accuracy, speed, memory usage, and versatility of architectures. This chapter describes the design of two sketch-based measurement architectures, namely A-CM Sketch and SPArch. A-CM sketch leverages the advantages of approximate computing to enhance memory efficiency and thereby reduce routing delay to improve the operating frequency. On the other hand, SPArch exploits the parallelism of FPGAs, resulting in excellent accuracy while minimizing memory usage. In addition, SPArch supports the individual mapping of flows analogous to associative arrays, providing versatility to sketch-based flow measurement. Both A-CM sketch and SPArch exploit the advantages of FPGAs and exhibit excellent memory-accuracy trade-off. The chapter provides a detailed hardware analysis of these

architectures, highlighting their superiority over existing flow measurement architectures.

The inability to detect a wide range of attacks using a single detection algorithm poses a limitation in DDoS attack detection. To address the challenges in deploying multiple detection algorithms on hardware, Chapter 6 proposes the design of a detection framework called FLARE. FLARE integrates multiple detection algorithms by sharing components, such as a blacklist, to deploy a full-fledged detection engine. In addition, this chapter efficiently implements LOFT as a detection algorithm. FLARE has eliminated the challenges associated with coordinating multiple algorithms and redundant use of hardware resources while allowing processing at line rate.

7.2 Future work

Extending to other network security applications

FLARE, the detection engine, is capable of handling a wide range of detection algorithms, including large flow detection, intrusion detection, and machine learning-based algorithms. What sets it apart is its adaptability to diverse disciplines, encompassing intrusion and anomaly detection, flow-measurement-based detection, and machine learning-based detection. This versatility makes FLARE an appealing solution. Moreover, upon examining the components proposed in this thesis, it becomes evident that the hardware-efficient designs presented are not limited to large flow detection alone. The fundamental building blocks employed in various network security applications, such as anomaly detection, intrusion detection, and traffic engineering, exhibit significant similarities to those of large flow detection. Components like hash functions, membership query data structures, and flow measurement architectures that are designed for large flow detection can be utilized to implement other applications on hardware with minimal additional design effort. Hence, both the components and the detection engine can be seamlessly extended to cater to other network security applications.

Expanding the possibilities of approximate computing

Our proposed A-CM Sketch demonstrates the effectiveness of approximate computing in constraining memory requirements. The use of approximate counters, as showcased by the A-CM Sketch using the approximate counter HSAC, has little to no effect on the accuracy of large flow detection. However,

accuracy becomes a concern when precise results are necessary within a short measurement period, particularly when we limit the bits allocated to the counter. For instance, the update counters of LOFT operate in the minor cycle and have a low error tolerance. Employing approximate counters with sizes below 10 bits may introduce fluctuations in measured volume. Although the flow table of LOFT that stores the aggregated values over a major cycle can make use of approximate counters, the accuracy needs to be improved when we aim to replace all exact counters with approximate ones. Existing accurate approximate counters employ complex hardware primitives such as logarithmic computation, which render it inefficient on hardware. This points to a future research direction of designing hardware-efficient and accurate approximate computing architectures, which is very much relevant because of the large amount of data to be aggregated and stored in DDoS detection algorithms. This research direction is applicable not only to network security but also to other computationally intensive applications such as deep learning and post-quantum cryptography.

Simplifying further: employing shared measurement modules

The network flow measurement modules employed in various detection algorithms can be either sketch-based or eviction-based. SPArch, our proposed architecture, can efficiently serve the purpose of both. This is particularly beneficial to a framework such as FLARE, where specific functionalities can be shared to reduce complexity. It also opens up avenues for future research. In this thesis, as discussed in Chapter 6, FLARE uses a common blacklist among the detection algorithms. However, the detection algorithms require a large memory overhead, the majority of which is attributed to the measurement module. In the FLARE implementation, the LOFT detection algorithm alone accounts for 97% of the total memory consumed by the testbed. When employing N different algorithms with comparable complexity, the overall memory consumption would be approximately N times the memory requirement of a single algorithm. Sharing a common measurement unit would be able to keep the memory requirement equivalent to that of a single algorithm. Furthermore, sharing measurement units drastically reduces the computational and latency burdens placed on individual detection units, as they are now only responsible for data processing and decision-making.

There are several factors to consider when using a common measurement unit. One challenge is the difference in the measurement mechanism among algorithms, such as sketch-based, eviction-based, and machine learning-based algorithms. Furthermore, the measurement epoch could be different for various algorithms, rendering it challenging to set the size and reset period of the

measurement module. Future research in this direction can focus on handling these challenges. One research approach to address the challenge associated with distinct measurement mechanisms is by categorizing the algorithms based on the measurement mechanism and assigning a shared module for each category. Another approach could be to limit the deployed algorithms to those that employ the same measurement mechanism. This also calls for another research direction focused on designing or modifying algorithms specifically tailored for FLARE.

Another research direction focuses on addressing the issue of different measurement periods while sharing a measurement module. One approach could be that the data can be tapped out from the measurement module at different intervals, driving individual detection algorithms. These algorithms may associate simple flow tables or counters to analyze the data. The reset period can be the measurement period of the algorithm having the longest period, considering that the measurement period or multiples of the measurement period of other algorithms align with the longest period. Again, this calls for designing or modifying algorithms specific to FLARE.

Transferring FLARE to other hardware platforms

Programmable ASIC switches are another hardware platform popularly employed for deploying network security algorithms. The FLARE framework requires limited resources and memory, allowing it to be easily transferred to programmable switches. The existing detection algorithms targeting these switches can be employed as brains. Moreover, the existence of universal sketches like UnivMon [164] and LUS [243] targeting programmable switches put forward an interesting research prospect of sharing the universal sketches as measurement units in conjunction with various detection algorithms as brains. Therefore the research prospect of deploying and enhancing FLARE to programmable switches is wide open as there are more deployment options for components, such as shared measurement units.

Commercializing to data centers

Another research direction of this thesis is commercializing it to data centers. Although the 10GbE and 40GbE are currently the prevalent Ethernet standards in data centers, the shift towards Terabit Ethernet is on the move. Data centers are already equipped with Terabit Ethernet, giving importance to the line of research this thesis follows. The 1.6TbE standard is expected to be established by 2025, demanding the need for high-speed volumetric-attack

detection algorithms and architectures. FLARE is targeted for data center accelerator cards, making it a cost-effective product that can be easily deployed in data centers. The potential market for FLARE is expected to grow as data centers switch completely towards Terabit Ethernet. This makes the commercialization of FLARE and the associated hardware-efficient components one of our primary future research directions.

Chapter 8

Valorization

Research valorization refers to the process of transforming research findings and knowledge into tangible and valuable outcomes that positively impact society and the economy. A valorization plan serves as a roadmap for disseminating and exploiting the results of a research project to ensure that the research has a real-world impact. Through valorization, the aim is to identify viable commercial applications for the research findings, leading to the development of new products, services, or processes. This chapter explores the potential valorization plan for this thesis outlining the strategies to maximize real-world implications.

8.1 Introduction

Present state of DDoS attacks

Over the past two decades, we have witnessed a steep surge in DDoS attacks, of which volumetric DDoS attacks accounted for the lion's share. The frequency, volume, and complexity of these attacks have increased many-fold, especially in recent years post-COVID-19 pandemic. Analyzing the attack patterns in 2022 reveals a dramatic surge in comparison to 2021. Notably, the emergence of uncommon attack patterns, such as Carpet Bomb attacks, and their sudden spike is alarming as they differ from other DDoS attacks. These attacks target multiple destination IP addresses, utilizing numerous low-volume packets, making them difficult to detect and defend against. The attackers can easily hide under the radar as the volume over a single IP address does not exceed the threshold

volume, making it difficult for traditional defense mechanisms to identify and counter them. Additionally, the number of notorious existing attacks like Mirai botnets is also increasing at a rapid rate, further contributing to the escalating threat landscape. Figures 8.1 and 8.2 depict the sharp increase in Carpet Bomb and Mirai botnet attacks over the past two to three years.

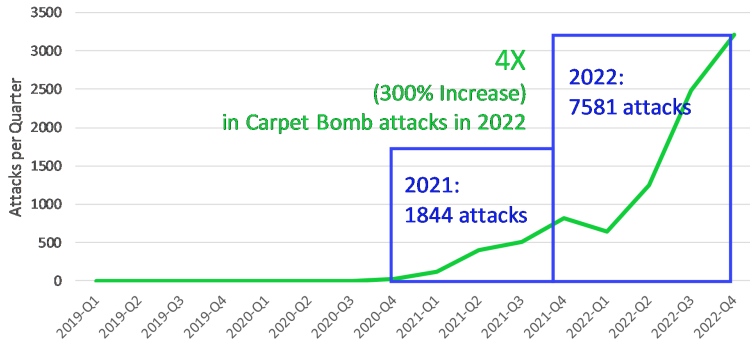


Figure 8.1: Growth in Carpet Bomb attack (Data: [32])

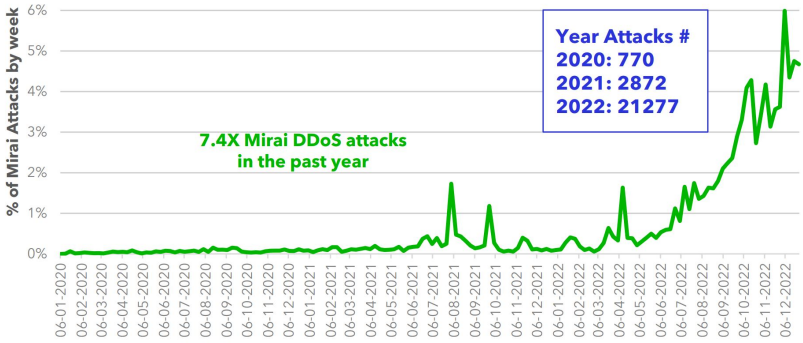


Figure 8.2: Growth in Mirai botnet attack (Source: [32])

The rising occurrence of sophisticated attacks has also fueled the increase in mitigation costs. For instance, DDoS attacks can cost organizations an average of USD 100,000 per every hour the attack persists [23]. The data from Kaspersky reveals that businesses spend an average of USD 20,000 to USD 2 million to neutralize a DDoS attack, which covers only the direct expenses [44]. The emerging trend of new attacks and the subsequent higher financial repercussions mark the importance of employing robust DDoS defense mechanisms. Most organizations opt for professional third-party protection services. Although the

organizations have to pay for these services, they typically pale in comparison to the potential financial consequences of a DDoS attack. The demand for expert defense solutions opens a vast market of opportunities for DDoS protection as a service. This is particularly encouraging for the valorization of this research.

Existing DDoS defense solutions

The DDoS defense/protection solutions can be categorized, on the basis of deployment mode, as **Cloud-based** solutions, **On-premise** solutions, and **Hybrid** solutions. In cloud-based solutions, the network traffic is redirected to a scrubbing station in the cloud. The scrubbing station analyzes and cleanses the network traffic before directing it to the client. On-premise solutions are deployed at the client's location and work alongside other devices, such as firewalls. Hybrid approaches utilize a combination of both cloud-based and on-premise solutions. A representation of DDoS defense solutions is shown in Figure 8.3.

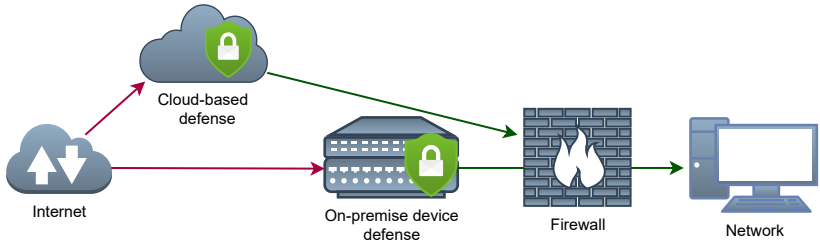


Figure 8.3: DDoS defense solutions

Both cloud-based and on-premise solutions come with their respective limitations. Cloud-based solutions typically rely on NetFlow, which reduces visibility to payloads and thereby reduces detection accuracy [37]. In cloud-based solutions, the routing of network traffic to the scrubbing centers in the cloud results in delays, leading to notable mitigation timeframes, usually measured in minutes. These delays in detecting and mitigating threats can result in considerable damage, particularly because attackers can inflict harm within seconds, especially with attacks like Carpet Bomb, which require swift mitigation within seconds. Furthermore, due to the limited bandwidth of scrubbing centers compared to an internet service provider's (ISP) edge capacity, they often get overwhelmed during multiple simultaneous attacks. Consequently, ISPs are left with no choice but to blackhole all traffic to the affected customers, rendering them entirely offline.

On the other hand, on-premise solutions also have their share of limitations. The primary concern here is the cost, as dedicated devices can be quite expensive, ranging from tens of thousands to hundreds of thousands of dollars. Additionally, there are associated overheads such as manpower and management costs, maintenance expenses, and utility costs. With the continuous increase in bandwidth and the emergence of new and sophisticated attacks, the on-premise device must undergo frequent updates to remain effective, leading to more expenses.

However, compared to cloud-based solutions, on-premise solutions hold certain advantages. On-premise solutions offer near-zero latency as the device is deployed locally in the data centers, resulting in faster mitigation. The most important feature of on-premise solutions is the level of control they offer. The organization has maximum control over the defense mechanism that cloud-based solutions cannot offer. Another factor that is advantageous to on-premise solutions is data privacy. Sectors such as healthcare, military, and finance have strict regulations on transferring their network traffic, which includes sensitive data, to the cloud. Recent data indicates that 75% of DDoS attacks last less than ten minutes [32], making cloud-based solutions less useful for combating such attacks. Moreover, 98% of the DDoS attacks are less than 10Gbps in size [32, 39], which could easily be defeated by an on-premise device. A comprehensive and all-encompassing protection strategy can be achieved by combining an on-premise approach to efficiently counter small and short-duration attacks with a backup cloud solution specifically designed to tackle large volumetric attacks.

Choosing a defense solution: Considering all these advantages and disadvantages associated with different schemes, the choice of a defense solution relies on a multitude of factors. Among these factors are the organization's size and longevity, mitigation latency requirements, the level of control needed, regulations governing the transfer of sensitive data, and the organization's threat profile [36]. On-premise solutions are more advantageous to service providers and organizations that own data centers and expect greater longevity, organizations that operate with sensitive data, and organizations that handle latency-critical applications. On the other hand, cloud-based solutions are more appropriate for organizations hosted on the cloud and those with budgetary constraints.

8.2 Market Dynamics

The DDoS protection and mitigation market is worth 3.9 billion in 2022 and is projected to hit 7.3 billion by 2027, at a compound annual growth rate (CAGR) of 13.2% [40]. The rise in DDoS attacks and high demand for DDoS security solutions make North America the largest market, followed by Europe and Asia Pacific [26]. Currently, Asia Pacific is the fastest-growing market and is expected to surpass Europe in the near future. Given that the Internet is an indispensable service, the DDoS protection and mitigation market is widespread in all segments, including government and defense, banking and financial sector, healthcare, research and education, IT and telecommunication, manufacturing, retail, and energy sector. The banking and finance sector is the largest segment in the market, while the healthcare sector exhibits the highest CAGR as per the recent reports [40, 26].

The DDoS security market is highly competitive due to the rising demand and increasing revenue. Key players in DDoS security market include Akamai Technologies, Radware, F5 Networks, Cloudflare, Netscout, Fortinet, Huawei, Imperva, Verisign, and Corero. The security solutions provided by these companies can be categorized into three components: hardware solutions, software solutions, and services. Currently, the services sector holds the major share of the market revenue. As outlined in Section 8.1, the deployment modes of the solution can be cloud-based, on-premise, and hybrid. While on-premise solutions dominate the Northern American market, cloud-based solutions are gaining popularity worldwide [40] as more and more businesses are migrating to the cloud. In fact, the cloud-based mode is the most widely chosen across the globe, accounting for just over 50% of the total market share [41].

8.3 Valorization potential and commercialization of the research

8.3.1 Summary of research results

After gaining insights into the market dynamics and existing defense solutions, as illustrated in the previous sections, the knowledge gaps in the existing solutions can be clearly noticed. These gaps align perfectly with the objectives of our research. This research intends to develop a hardware-oriented solution for DDoS detection, specifically targeting Terabit Ethernet. Focusing on valorization potential, the major contribution of this research is FLARE, an all-purpose large-flow detection framework designed for high-speed hardware. While this

thesis focuses solely on large flow detection, FLARE can be effortlessly extended to other DDoS detection applications without necessitating any design changes, as discussed in Section 7.2.

It is to be noted that FLARE is built upon the other novel contributions in this research, which include hardware-efficient hashing, lookup, and flow measurement architectures, namely Xoodoo-NC, Parallel Bloom filter, and SPArch. These novel contributions help to achieve state-of-the-art performance on high-speed hardware, enhancing the value of the final product. Among these innovations, SPArch stands out as the sole available probabilistic alternative to CAM-based counter arrays that can significantly boost speed and reduce resource consumption compared to existing solutions. This aspect is particularly important for deploying DDoS detection in high-speed hardware environments. As data rates and storage demands continue to increase, the proposed components in this research add value to improve the sustainability of our detection solution in the market.

8.3.2 What does this research offer to the market?

In this research, we put forward FLARE, a hardware-oriented DDoS detection engine targeting Terabit Ethernet, with the aim of transforming it into a commercially viable product. The detection framework, FLARE, is particularly well-suited for deployment as an on-premise device due to its compatibility with various high-speed hardware setups. Both cloud-based and on-premise solutions have their own merits and demerits, which are applicable across different vendors. Selecting the appropriate deployment mode should be primarily based on factors such as cost and the product's suitability for the mode while pushing this research to market. Considering these factors, FLARE is positioned as an ideal candidate for marketing as an on-premise detection solution. This decision is supported by the fact that cloud-based solutions necessitate substantial infrastructure investments when it comes to DDoS mitigation, leading to high costs and making it a less favorable option for FLARE, especially at the initial stages of market entry.

Based on the existing market studies, it can be inferred that there is significant room for improvement when it comes to on-premise DDoS detection solutions. High cost is one of the primary drawbacks of on-premise solutions. The one-time deployment cost itself is substantial, adding to the recurring costs like maintenance, staff management, and utility costs. Additionally, it would require regular device updates based on the varying data rate and threat profile, further incrementing the maintenance cost. This high cost may not favor small and medium-sized enterprises (SME). Most cost-effective on-premise solutions are

bandwidth-limited to Gbps, restricting their applicability to small and short-duration attacks.

In contrast, FLARE offers a cost-effective deployment due to its utilization of FPGA as the deployment platform. This enables data centers or businesses to seamlessly integrate it into their existing compatible hardware setups, lowering installation expenses. This low installation cost makes it highly suited for SMEs. Another important feature of FLARE is that it can support Terabit Ethernet, placing it well above the existing solutions due to its low cost. As an FPGA-based solution, updating the hardware is much easier and cost-efficient than other hardware-based on-premise solutions.

Market Competitors

Present on-premise devices mostly offer protection against application layer attacks and very limited protection against volumetric attacks. Complete protection against volumetric attacks is provided only with cloud-based solutions. Notable on-premise DDoS protection solutions include FortiNet's FortiDDoS [8], Radware's DefensePro [42], Checkpoint's DDoS Protector [43], NetScout's AED [33], and Corero's Smartwall [38]. Vendors like Akamai and Imperva also provide on-premise options, but only on a subscription basis, similar to cloud-based solutions. Among these, FortiDDoS stands out as a hardware-based solution that employs 100% ASIC-based technology. High power consumption is one of the drawbacks of on-premise protection devices, and a complete hardware solution significantly reduces energy requirements.

The high-end versions of available on-premise protection devices can support high bandwidths; for instance, DefensePro 400 can support 400Gbps and Checkpoint DP 800 can support 380Gbps. However, the cost is also proportional to the throughput. DefensePro 400 is priced at approximately USD 720,000, while Checkpoint DP 800 costs around USD 740,000. An even higher-priced option is Netscout's HD1000, which can handle 200Gbps but comes at a hefty price of about USD 1.5 million. In addition to the initial cost, the yearly support premiums are also substantial, often exceeding USD 100,000. Price falls when opting for devices with reduced bandwidth support but is still considerably high. The high-end version of FortiDDoS, 2000E-DC, can support only up to 90Gbps throughput and costs around USD 750,000, with an annual premium support cost of nearly USD 150,000. Moving to cloud-based subscriptions does not necessarily bring cost relief either. For a minimum of 40Gbps throughput, the monthly prices start from approximately USD 20,000, the cheapest option from Cloudflare. These exorbitant prices pose significant challenges, especially for small and medium-sized enterprises (SMEs), who may find them unaffordable.

The high energy requirement is another drawback of the existing software-based solutions, which not only have a financial impact but also a huge environmental impact. FortiDDoS, being a hardware-based solution, has reduced energy requirements. However, an ASIC-based device like FortiDDoS may suffer from scalability issues as the new threat profiles cannot be integrated into the existing hardware. This scalability issue is not unique to FortiDDoS but also affects other software-based solutions when their hardware platforms struggle to meet the demands of upgraded software requirements.

Compared to the competitors, FLARE holds certain advantages.

- **Reconfigurability and scalability:** A complete FPGA-based solution like FLARE is non-existent in the current market. Reconfigurability adds much value to FLARE as its threat profile and the corresponding hardware designs can be easily modified. In addition, FLARE's ability to seamlessly integrate multiple algorithms within a unified framework enhances its scalability.

- **Security:** Unlike software-based alternatives, FLARE offers a hardware-based solution that is more resilient against threats. Software-based defenses can be susceptible to attacks such as resource saturation, where an attacker can saturate the resources, thereby disabling the defense.

- **Cost-effectiveness:** An FPGA-based solution is less costly when compared to other solutions. The ASIC-based FortiDDoS device that can support only up to 8Gbps comes at a price exceeding USD 80,000 and additional yearly support costs. On the other hand, our target FGPA, the Alveo U250 hardware accelerator card, costs less than USD 10,000. Moreover, the overall logic resources needed for the FLARE testbed, including the LOFT algorithm, account for less than 10% of the resources available in the Alveo U250. Even though FLARE necessitates the incorporation of several algorithms for comprehensive DDoS detection and mitigation, obtaining eight Alveo cards at the cost of a single 8Gbps FortiDDoS device is viable and more than sufficient to accommodate all the algorithms. Additionally, the FPGA solution boasts low energy requirements and update costs. In addition to the hardware costs, the costs associated with developing a market-ready product contribute to the overall cost of the product. However, with FLARE, the customers have the flexibility to choose the mode of the deployment (cf. 8.3.4:Product marketing options), making it much more affordable than existing products. If the customers opt not to purchase the hardware, the incurred cost is primarily associated with research and development (R&D) expenditures only, making it manageable for the customer.

- **Superior control:** On-premise devices provide customers a higher degree of control, and an FPGA-based device can be exploited to further extend

this control. The FPGA can be effortlessly reconfigured to provide users with expanded command and control capabilities.

8.3.3 Target audience

The primary focus is directed toward small and medium-sized enterprises and organizations subject to data transfer regulations. Larger businesses like internet service providers are not within the scope of our target audience due to the challenges posed by handling high data loads. Although large enterprises currently account for 70% of the overall market revenue, it is strategically advantageous for emerging or entry-level vendors to invest in the SME market, as it is less competitive. Most of the established vendors are focused on large enterprises as they hold the lion's share of the market. Moreover, the market share of SMEs is projected to experience a substantial 4.6% growth in the next ten years [41]. This anticipated growth is likely to drive an increase in market demand for our products and services.

The ideal target market segments for on-premise protection are government and defense, healthcare, finance, and research institutes, as they operate in a regulated environment. These organizations generate sensitive data that necessitates robust security measures. Since SMEs operate in diverse environments, a one-size-fits-all approach is unsuitable. Instead, customized control and management solutions are essential. FLARE addresses this requirement by offering individually tailored control solutions, thanks to the reconfigurability of FPGAs. The current version of FLARE already features a control and management interface that can be personalized according to the user's specific needs. Furthermore, the expertise and recent accomplishments of our research group, particularly in enhancing secure communication in medical devices [240], add credibility and market value to our product in sectors such as healthcare.

The cost would be a primary concern for research institutions and smaller enterprises. Even the low-end on-premise devices with limited throughput would come with a significant price tag. Cloud-based options are not viable due to data privacy concerns. Consequently, FLARE benefits greatly from this situation, as its expenses, including updates and maintenance, would be considerably lower than the other solutions.

8.3.4 Exploitation strategies

There are three possible routes to exploit the research and drive our research outcomes into the market.

1. Collaborating with other companies
2. Licensing the product
3. Creating a spin-off

Collaborating with other companies: Collaborating with other companies is the most advantageous approach to introducing a new product to the market. This strategy not only eliminates the need to invest in building new infrastructure but also reduces the initial costs associated with marketing, sales, and distribution, as these resources can be shared with the partner companies. Moreover, leveraging the established market presence and credibility of the collaborating company provides the new product with a wider reach and increased visibility. Furthermore, by associating the new product with complementary products and services already offered by the parent company, customer loyalty can be enhanced, leading to a more sustainable customer base. The shared expertise between the collaborating companies fosters innovation and facilitates the continuous improvement of the product.

Identifying whether it is possible to establish a collaboration for the product is the first step. Various factors, including the company's objectives and product offerings, must be taken into account while choosing a potential collaborator. If the company is a direct market competitor, establishing a collaborative relationship would be difficult. Since an FPGA-based DDoS security solution is currently non-existent in the market, finding a collaborator for FLARE who is not a direct competitor should be easier. Once the decision to collaborate with a company is made, discussions regarding collaboration agreements and strategies should ensue.

Licensing the product: Licensing the product offers a way to generate revenue without bearing financial risks and expenses associated with manufacturing and marketing. For instance, a licensing fee for each unit sold can be applied, allowing for a steady income stream. Similar to collaboration with other companies, licensing provides access to a diverse customer base and enables the provision of complementary services to the product, leveraging the resources available to the companies.

When licensing our product, several factors must be taken into account. Identifying whether a company can exploit our solution is the first question to be answered. Licensing the product to a direct competitor is not advisable. A drawback with licensing our product is that we lose some of the control that we have over the product. Therefore, the drafting of the licensing agreement should be done meticulously to preempt any potential legal complications in the future. Moreover, it is essential to ensure robust protection against any violations of intellectual property rights.

Creating a spin-off: The benefit of initiating a spin-off lies in the increased control and freedom we gain over the product. However, this advantage comes with certain risks. The primary concern that must be addressed when establishing a spin-off is the product's capability to generate adequate and steady income in the market over the long term. It would also require finding potential external investors to raise the capital. Additionally, an external entrepreneurial team would need to be recruited since it falls outside the expertise of the research team. It is also essential to effectively manage the competition from the market. Creating a spin-off is riskier than the other two options, but its advantages outweigh the challenges.

Product marketing options

The product can be offered to the market in different ways.

- Provide a complete hardware stack and GUI software: This option offers a complete, ready-to-install solution. The payment options can be defined in a way that includes an initial cost of the product and a recurring monthly/annual support fee. New threat profiles can be revised at no additional cost, as they are covered by the support fee. Establishing a collaboration with a hardware vendor, such as AMD Xilinx [34], could help reduce the hardware cost and thereby minimize the initial cost of the device.
- Provide only the bitstream to be installed on the compatible customer hardware, along with the GUI software: This option offers the bitstream alone, requiring the customer to have a compatible hardware platform. The initial cost includes only the cost of the bitstream. This is cost-effective for the customer as they do not have to pay for the expensive hardware. The support fee can be kept the same as the complete solution.

The existing ASIC-based on-premise solutions would require a complete replacement of the hardware to update the threat profile, thereby increasing the installation charges for the customer. In recent times, frequent updates of the threat profile become necessary due to the frequent emergence of new threats. For instance, the Carpet Bomb attack came into the picture in 2021, and it is near-impossible for the existing algorithms to detect this attack.

By potentially partnering with a cloud-based company, the product can be deployed as a cloud-based solution. This is particularly beneficial for the parent company as there are reduced cost, energy, and maintenance requirements, as well as reduced vulnerability compared to software-based solutions. Nevertheless, from a customer's perspective, this approach negates the benefits of an in-line

hardware solution, potentially leading to less customer loyalty compared to competing products.

Route to market

This section briefly overviews the timelines and prerequisites involved in marketing FLARE to the public by establishing a spin-off or collaboration. The product, FLARE, has a proof-of-concept prototype ready but is not yet market-ready. Presently, the prototype of FLARE is designed exclusively for large flow detection, while the other detection algorithms and mitigation strategies need to be updated to FLARE. The software program for the graphical user interface must be designed and implemented. Once the complete prototype is ready, proper validation should be carried out, and the required certifications should be acquired to launch the product into the market. With the completion of all the requirements to market, the preparation to launch the product into the market can commence. Initially, the product can be launched in Belgium, with the intention to expand into international markets following a successful launch in Belgium. The expected timeline of the route to the market is illustrated in Figure 8.4.

IP rights

The IP rights ownership is shared by the university and the external funding agencies, FWO and SNCF. The implementation is entirely free from the use of libraries or third-party IPs; instead, it solely relies on publicly available research materials, granting us complete autonomy over our IP.

Protection of IP: Given that the fundamental concept behind the architecture is openly available through publications, pursuing a patent may not be advisable. However, we can maintain confidentiality by keeping the hardware implementations and specific parameters used for these implementations as trade secrets. When delivering the product, we provide only the bitstream, ensuring that the code remains inaccessible to customers and competitors. Although there is a possibility that other parties may attempt to reverse engineer the bitstream over time, the frequent update of the threat profile will result in frequent hardware changes. An attacker would require in-depth knowledge of the hardware architectures we use and expertise in low-level hardware design to efficiently replicate the hardware threat profile updates. Without these updates, the device would quickly become outdated.

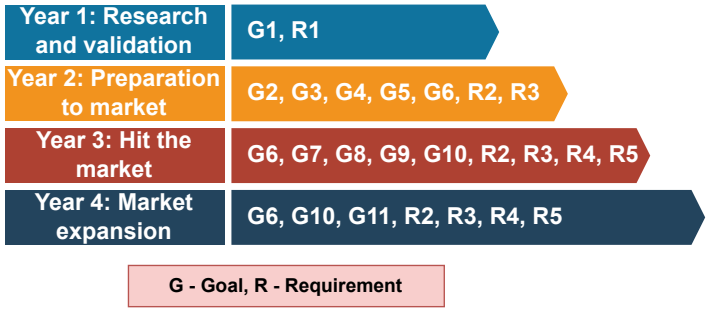


Figure 8.4: Timeline to market

8.4 Economic and social value

A hardware-based DDoS protection solution is significantly more energy-efficient than its software-based counterparts. From an economic perspective, energy savings achieved could contribute to significant cost reduction for SMEs. In the aftermath of the COVID-19 pandemic, the surge in energy prices directly or indirectly affected numerous SMEs, driving some of them to the brink of bankruptcy. It is crucial to note that SMEs represent 99% of the businesses worldwide and generate approximately 60% of employment opportunities [46]. In Europe alone, SMEs play a pivotal role, offering employment to more than 80 million individuals and contributing to the production of over 50% of domestic products [47]. Such is the economic impact of SMEs worldwide, and an economic crisis in SMEs would dent the global economy. Hence, even a modest reduction of 10-20% energy costs could prove lifesaving for SMEs and have a positive impact on the global economy.

According to the 2015 report from the International Energy Agency, SMEs account for 13% of the global energy demand [46]. The world's green energy comprises only 30% of the total energy generated [48]. Increased energy

consumption would negatively impact the current climate crisis, as 70% of the energy comes from non-renewable sources such as fossil fuels. Non-renewable energy sources increase greenhouse gas emissions, thereby worsening the impact on the global environmental crisis. Reducing energy consumption would lead to lower greenhouse gas emissions, reduced water consumption, improved air quality, and better public health.

In addition to the reduced energy expenses, our proposed solution offers a significantly lower cost price. Moreover, the customer can choose to reduce the costs further if they already possess compatible hardware. The high cost of existing DDoS security solutions, whether it is cloud-based or on-premise-based, is a financial burden to small and medium enterprises. This high cost is pulling these companies back from employing effective security measures. As per the National Cybersecurity Alliance report, only 14% of small businesses currently have effective cybersecurity measures in place [51]. Low-cost solutions mean less financial burden and more companies and research institutions will be encouraged to install robust security solutions.

Furthermore, the high cost of existing solutions will also compel enterprises to use free or pirated versions of DDoS solutions. However, the so-called "free" versions are never free, and it could come at the price of the private data they leak. The data leak could have catastrophic consequences in society, especially when it involves the sensitive information of private individuals. For instance, the 2018 AAdhar data breach exposed personal and confidential data, including biometric records, of 1.1 billion Indian citizens due to inadequate security measures. Moreover, employing robust DDoS solutions not only safeguards against data leaks and breaches but also ensures that confidential information remains out of the wrong hands. Protecting customer data and combating piracy becomes a significant and commendable social endeavor.

Bibliography

[1] https://www.caida.org/data/passive/trace_stats/nyc-B/2018/?monitor=20181018-130000.UTC.

[2] AMD OpenNIC project. https://github.com/Xilinx/open-nic/blob/main/OpenNIC_manual.pdf.

[3] Cost of a Data Breach Report 2022. https://www.ibm.com/downloads/cas/3R8N1DZJ.

[4] Denial-of-service attack. https://en.wikipedia.org/wiki/Denial-of-service_attack.

[5] Differences between CPU, GPU, FPGA, and ASIC . https://forum.huawei.com/enterprise/en/differences-between-cpu-gpu-fpga-and-asic/thread/966389-895.

[6] DIGITAL AROUND THE WORLD 2022. https://datareportal.com/global-digital-overview.

[7] Ethernet speeds. https://tadviser.com/index.php/Article:Ethernet.

[8] FortiDDoS. https://www.fortinet.com/products/ddos/fortiddos.

[9] FPGAs to Replace GPUs in AI Accelerators. https://www.eetasia.com/fpgas-to-replace-gpus-in-ai-accelerators/.

[10] Global average Internet speed. https://futuretimeline.net/data-trends/2050-future-internet-speed-predictions.htm#.

[11] Harvest Now, Decrypt Later. https://www.insidequantumtechnology.com/news-archive/harvest-now-decrypt-later/.

[12] How much data is created every day in 2022? https://earthweb.com/how-much-data-is-created-every-day/. Published on: 2022-07-25.

[13] IEEE 802.3. https://en.wikipedia.org/wiki/IEEE_802.3.

[14] The impact of covid-19 on cybercrime. https://fully-verified.com/the-impact-of-covid-19-on-cybercrime/.

[15] Minimizing Risk with FPGAs and Hardware-Based Security . https://www.armadainternational.com/2021/08/minimizing-risk-with-fpgas-and-hardware-based-security/.

[16] The State of Phishing 2022. https://www.slashnext.com/wp-content/uploads/2022/10/SlashNext-The-State-of-Phishing-2022.pdf.

[17] UltraScale Architecture Configurable Logic Block User Guide. https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb.

[18] Gbps vs. pps vs. rps DDoS: On Volumetric, Protocol and Application Layer Attacks. https://www.imperva.com/blog/gbps-pps-rps-ddos-attacks/, 2017.

[19] 34 Cybersecurity Statistics to Lose Sleep Over in 2020. https://www.techtarget.com/whatis/34-Cybersecurity-Statistics-to-Lose-Sleep-Over-in-2020, 2020.

[20] Cisco Annual Internet Report (2018–2023) White Paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, 2020.

[21] Exponential growth in DDoS attack volumes. https://cloud.google.com/blog/products/identity-security/identifying-and-protecting-against-the-largest-ddos-attacks, 2020.

[22] FPGAs Create the Optimal Data Accelerator. https://www.achronix.com/blog/fpgas-create-optimal-data-accelerator, 2021.

[23] Global DDOS protection market size and share analysis - growth trends and forecasts. https://www.mordorintelligence.com/industry-reports/ddos-protection-market, 2021.

[24] Product brief Alveo. https://www.xilinx.com/publications/product-briefs/alveo-product-brief.pdf, 2021.

[25] Vivado design suite user guide. https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_2/ug893-vivado-ide.pdf, 2021.

[26] Ddos Protection and Mitigation Security Market. https://www.polarismarketresearch.com/industry-analysis/ddos-protection-mitigation-security-market, 2022.

[27] Hiscox Cyber Readiness Report 2022. https://www.hiscox.com/documents/Hiscox-Cyber-Readiness-Report-2022.pdf, 2022.

[28] History of The Transistor (the "Crystal Triode"). https://www.bellsystemmemorial.com/belllabs_transistor.html, 2022.

[29] Ieee standard for ethernet. *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)* (2022), 1–7025.

[30] Intel tofino 3 intelligent fabric processor brief. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-3-brief.html, 2022.

[31] What is intel tofino? an easy guide to its benefits and use cases. https://www.whiteboxsolution.com/blog/what-is-intel-tofino-an-easy-guide-to-its-benefits-and-use-cases/, 2022.

[32] 2023 DDoS Threat Intelligence Report. https://go.corero.com/threat, 2023.

[33] Adaptive DDoS Protection Solution. https://www.netscout.com/solutions/ddos-protection, 2023.

[34] AMD Xilinx. https://www.amd.com/en/products/xilinx, 2023.

[35] Carpet Bomb DDoS Attacks: On the Rise and Evading Detection. https://www.corero.com/threat-report-carpet-bomb-intro/, 2023.

[36] Choosing the Right DDoS Solution – Part I: On-Prem Appliance. https://www.radware.com/blog/security/2018/03/choosing-ddos-solution-on-prem-appliance/, 2023.

[37] Cloud, Hybrid, or On-Prem: Which DDoS Protection Is Right for Your Organization? https://www.corero.com/cloud-hybrid-or-on-prem-ddos-protection-options/, 2023.

[38] Corero Smartwall. https://www.corero.com/solutions/on-premise/, 2023.

[39] DDoS Attack Trends for 2022 Q1. https://blog.cloudflare.com/ddos-attack-trends-for-2022-q1/, 2023.

[40] Ddos Protection and Mitigation Security Market. https://www.marketsandmarkets.com/Market-Reports/ddos-protection-mitigation-market-111952874.html, 2023.

[41] DDoS Protection Market. https://www.futuremarketinsights.com/reports/ddos-protection-market, 2023.

[42] DefensePro X: The Next Level of DDOS Protection. https://www.radware.com/products/defensepro/, 2023.

[43] Quantum DDoS Protector. https://www.checkpoint.com/quantum/ddos-protector/, 2023.

[44] Understanding the Financial Impact of DDoS Attacks: Why Investing in Mitigation Strategies is Crucial. https://ts2.space/en/the-cost-of-ddos-attacks-why-mitigation-is-a-smart-investment/, 2023.

[45] National Institute of Standards and Technology (NIST). FIPS180-2: Secure Hash Standard. http://www.itl.nist.gov/fipspubs/, August 2002.

[46] (2015), I. Policy Pathway - Accelerating Energy Efficiency in Small and Medium-sized Enterprises 2015, IEA, Paris. https://www.iea.org/reports/policy-pathway-accelerating-energy-efficiency-in-small-and-medium-sized-enterprises-2015, 2015.

[47] (2015), I. Coping with the crisis: Increasing resilience in SMEs through energy efficiency. https://energy.ec.europa.eu/topics/markets-and-consumers/actions-and-measures-energy-prices/coping-crisis_en, 2022.

[48] (2021), I. Global Energy Review 2021, IEA, Paris. https://www.iea.org/reports/global-energy-review-2021, 2021.

[49] AbuHmed, T., Mohaisen, A., and Nyang, D. A survey on deep packet inspection for intrusion detection systems. *arXiv preprint arXiv:0803.0037* (2008).

[50] Al-Shatari, M. O. A., Hussin, F. A., Abd Aziz, A., Witjaksono, G., and Tran, X.-T. FPGA-Based Lightweight Hardware Architecture of the PHOTON Hash Function for IoT Edge Devices. *IEEE Access 8* (2020), 207610–207618.

[51] Alliance, N. C. Oh Behave! The Annual Cybersecurity Attitudes and Behaviors Report 2022. https://staysafeonline.org/online-safety-privacy-basics/oh-behave/, 2022.

[52] Almeida, P. S., Baquero, C., Preguiça, N., and Hutchison, D. Scalable bloom filters. *Information Processing Letters 101*, 6 (2007), 255–261.

[53] AMD Xilinx. Content Addressable Memory (CAM). `https://www.xilinx.com/products/intellectual-property/ef-di-cam.html`.

[54] Anand, P. A., et al. Design of high speed crc algorithm for ethernet on fpga using reduced lookup table algorithm. In *2016 IEEE Annual India Conference (INDICON)* (2016), IEEE, pp. 1–6.

[55] Anandakumar, N. N., Peyrin, T., and Poschmann, A. A very compact FPGA implementation of LED and PHOTON. In *International Conference on Cryptology in India* (2014), Springer, pp. 304–321.

[56] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)* (2017), pp. 1093–1110.

[57] Aumasson, J.-P., and Bernstein, D. J. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India* (2012), Springer, pp. 489–508.

[58] Aumasson, J.-P., Neves, S., Wilcox-O'Hearn, Z., and Winnerlein, C. Blake2: simpler, smaller, fast as md5. In *Applied Cryptography and Network Security: 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings 11* (2013), Springer, pp. 119–135.

[59] Bagnulo, M., Matthews, P., and van Beijnum, I. Stateful nat64: Network address and protocol translation from ipv6 clients to ipv4 servers. Tech. rep., 2011.

[60] Banik, S., Pandey, S. K., Peyrin, T., Sasaki, Y., Sim, S. M., and Todo, Y. GIFT: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems* (2017), Springer, pp. 321–345.

[61] Bao, Z., Chakraborti, A., Datta, N., Guo, J., Nandi, M., Peyrin, T., and Yasuda, K. Photon-beetle authenticated encryption and hash family. *NIST Lightweight Compet. Round 1* (2019), 115.

[62] Basat, R., Einziger, G., Mitzenmacher, M., and Vargaftik, S. Faster and more accurate measurement through additive-error counters. In *arXiv:2004.10332 [cs.DS]* (2020).

[63] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., and Wingers, L. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference* (2015), pp. 1–6.

[64] Beierle, C., Biryukov, A., dos Santos, L. C., Grossschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q., and Biryukov, A. Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round 2* (2019).

[65] Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., and Sim, S. M. Skinny-aead and skinny-hash. *IACR Transactions on Symmetric Cryptology* (2020), 88–131.

[66] Bender, M. A., and et al. Don't thrash: how to cache your hash on flash. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)* (2011).

[67] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2010), IMC '10, Association for Computing Machinery, p. 267–280.

[68] Bex, L., Turan, F., Van Beirendonck, M., and Verbauwhede, I. Mining cryptonight-haven on the varium c1100 blockchain accelerator card. *arXiv preprint arXiv:2212.05033* (2022).

[69] Black, S., and Kim, Y. An overview on detection and prevention of application layer ddos attacks. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)* (2022), IEEE, pp. 0791–0800.

[70] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970).

[71] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[72] Bobda, C., Mbongue, J. M., Chow, P., Ewais, M., Tarafdar, N., Vega, J. C., Eguro, K., Koch, D., Handagala, S., Leeser, M., et al. The future of fpga acceleration in datacenters and the cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS) 15*, 3 (2022), 1–42.

[73] Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., and Horowitz, M. Forwarding

metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[74] Breslow, A. D., and Jayasena, N. S. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment 11*, 9 (2018), 1041–1055.

[75] Britannica. Denial of service attack. `https://www.britannica.com/technology/denial-of-service-attack`, 2023.

[76] Cao, Z., and Wang, Z. Flow identification for supporting per-flow queueing. In *Proceedings Ninth International Conference on Computer Communications and Networks (Cat. No. 00EX440)* (2000), IEEE, pp. 88–93.

[77] Carter, J. L., and Wegman, M. N. Universal classes of hash functions. *Journal of computer and system sciences 18*, 2 (1979), 143–154.

[78] Celis, P., Larson, P.-A., and Munro, J. I. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)* (1985), IEEE, pp. 281–288.

[79] Charikar, M., Chen, K., and Farach-Colton, M. Finding frequent items in data streams. In *Automata, Languages and Programming* (2002), Springer Berlin Heidelberg, pp. 693–703.

[80] Chen, H., Liao, L., Jin, H., and Wu, J. The dynamic cuckoo filter. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (2017), IEEE, pp. 1–10.

[81] Chen, M., Chen, S., and Cai, Z. Counter tree: A scalable counter architecture for per-flow traffic measurement. *IEEE/ACM Transactions on Networking 25*, 2 (2017), 1249–1262.

[82] Chen, S., Chen, M., and Xiao, Q. *Traffic measurement for big network data.* Springer, 2017.

[83] Chiosa, M., Preusser, T. B., and Alonso, G. Skt: A one-pass multi-sketch data analytics accelerator. *Proceedings of the VLDB Endowment 14*, 11 (2021), 2369–2382.

[84] Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., et al. Serving dnns in real time at datacenter scale with project brainwave. *iEEE Micro 38*, 2 (2018), 8–20.

[85] CISCO. CISCO IOS NetFlow Version 9. http://www.cisco.com/c/en/us/products/ios-nx-os-software/netflow-version-9/index.html, 2015.

[86] Cohen, S., and Matias, Y. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (2003), pp. 241–252.

[87] Cormode, G., and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[88] Cormode, G., and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms 55*, 1 (2005), 58–75.

[89] Daemen, J., Hoffert, S., Assche, G. V., and Keer, R. V. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology* (2018), 1–38.

[90] Dang, Q. Recommendation for Applications Using Approved Hash Algorithms. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-107r1.pdf, 2023.

[91] de Neira, A. B., Kantarci, B., and Nogueira, M. Distributed denial of service attack prediction: Challenges, open issues and opportunities. *Computer Networks 222* (2023), 109553.

[92] Deshmukh, R. V., and Devadkar, K. K. Understanding ddos attack & its effect in cloud environment. *Procedia Computer Science 49* (2015), 202–210.

[93] Dharmapurikar, S., Attig, M., and Lockwood, J. Design and implementation of a string matching system for network intrusion detection using FPGA-based Bloom filters. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '04)* (2004).

[94] Dharmapurikar, S., Krishnamurthy, P., Sproull, T. S., and Lockwood, J. W. Deep packet inspection using parallel Bloom filters. *IEEE Micro, vol. 24, no. 1* (2004), 52–61.

[95] Diehl, W., Farahmand, F., Yalla, P., Kaps, J.-P., and Gaj, K. Comparison of hardware and software implementations of selected lightweight block ciphers. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)* (2017), IEEE, pp. 1–4.

[96] Donnet, B., Baynat, B., and Friedman, T. Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *Proceedings of the 2006 ACM CoNEXT conference* (2006), pp. 1–12.

[97] Duarte, F. Amount of Data Created Daily (2024). `https://explodingtopics.com/blog/data-generated-per-day`, 2023.

[98] Dworkin, M., Barker, E., Nechvatal, J., Foti, J., Bassham, L., Roback, E., and Dray, J. Advanced Encryption Standard (AES), 2001-11-26 2001.

[99] Einziger, G., Fellman, B., and Kassner, Y. Independent counter estimation buckets. In *IEEE INFOCOM* (2015), pp. 2560–2568.

[100] Einziger, G., Fellman, B., and Kassner, Y. Independent counter estimation buckets. In *2015 IEEE Conference on Computer Communications (INFOCOM)* (2015), IEEE, pp. 2560–2568.

[101] Einziger, G., Fellman, B., and Kassner, Y. Independent counter estimation buckets. In *IEEE INFOCOM* (2015), pp. 2560–2568.

[102] Ender, M., Moradi, A., and Paar, C. The unpatchable silicon: a full break of the bitstream encryption of xilinx 7-series {FPGAs}. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 1803–1819.

[103] Estan, C., and Varghese, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Transactions on Computer Systems 21* (2003), 270–313.

[104] Estébanez, C., Saez, Y., Recio, G., and Isasi, P. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience 44*, 6 (2014), 681–698.

[105] Estébanez, C., Sáez, Y., Recio, G., and Isasi, P. Performance of the most common non-cryptographic hash functions. *Software: Practice and Experience 44* (06 2014).

[106] et al., K. H. A multi-partitioning approach to building fast and accurate counting Bloom filters. *IEEE 27th International Symposium on Parallel and Distributed Processing* (2013), 1159–1170.

[107] Falsafi, B., Dally, B., Singh, D., Chiou, D., Joshua, J. Y., and Sendag, R. Fpgas versus gpus in data centers. *IEEE Micro 37*, 1 (2017), 60–72.

[108] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014).

[109] Fan, L., Cao, P., Almeida, J., and Broder, A. Z. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking 8*, 3 (2000).

[110] Flajolet, P., and Martin, G. N. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences 31*, 2 (1985), 182–209.

[111] Forrié, R. The strict avalanche criterion: spectral properties of boolean functions and an extended definition. In *Conference on the Theory and Application of Cryptography* (1988), Springer, pp. 450–468.

[112] Fowler, G., Noll, L. C., and Vo, K.-P. The FNV non-cryptographic hash algorithm. `https://tools.ietf.org/html/draft-eastlake-fnv-03`, 2012.

[113] Fowler, G., Noll, L. C., Vo, K.-P., Eastlake, D., and Hansen, T. The FNV non-cryptographic hash algorithm. *Ietf-draft* (2011).

[114] Fu, P., Luo, L., Li, S., Guo, D., Cheng, G., and Zhou, Y. The vertical cuckoo filters: A family of insertion-friendly sketches for online applications. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)* (2021).

[115] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. 2004.

[116] Geravand, S., and Ahmadi, M. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks 57*, 18 (2013), 4047–4064.

[117] Gong, J., Yang, T., Zhang, H., Li, H., Uhlig, S., Chen, S., Uden, L., and Li, X. Heavykeeper: An accurate algorithm for finding top-k elephant flows. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 909–921.

[118] Goudarzi, D., Jean, J., Kölbl, S., Peyrin, T., Rivain, M., Sasaki, Y., and Sim, S. M. Pyjamask: Block Cipher and Authenticated Encryption with Highly Efficient Masked Implementation. *IACR Transactions on Symmetric Cryptology* (2020), 31–59.

[119] Graf, T. M., and Lemire, D. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA) 25* (2020), 1–16.

[120] Graf, T. M., and Lemire, D. Binary fuse filters: Fast and smaller than xor filters. *Journal of Experimental Algorithmics (JEA) 27*, 1 (2022), 1–15.

[121] Grochol, D., and Sekanina, L. Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs. In *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)* (2018), IEEE, pp. 257–263.

[122] Guo, D., Wu, J., Chen, H., Yuan, Y., and Luo, X. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering 22*, 1 (2009), 120–133.

[123] Guo, J., Peyrin, T., and Poschmann, A. The PHOTON family of lightweight hash functions. In *Annual Cryptology Conference* (2011), Springer, pp. 222–239.

[124] Guo, L., Lau, J., Ruan, Z., Wei, P., and Cong, J. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2019), pp. 127–135.

[125] HardwareBee. The Ultimate Guide to FPGA Architecture. `https://hardwarebee.com/the-ultimate-guide-to-fpga-architecture/`, 2023.

[126] Harwayne-Gidansky, J., Stefan, D., and Dalal, I. FPGA-based SoC for real-time network intrusion detection using counting Bloom filters. *IEEE Southeastcon 2009, Atlanta, GA* (2009), 452–458.

[127] Hassan, M. On the off-chip memory latency of real-time systems: Is ddr dram really the best option? In *2018 IEEE Real-Time Systems Symposium (RTSS)* (2018), IEEE, pp. 495–505.

[128] Hassan, M., Sateesan, A., Vliegen, J., Picek, S., and Mentens, N. Evolving non-cryptographic hash functions using genetic programming for high-speed lookups in network security applications. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)* (2023), Springer, pp. 302–318.

[129] He, Z., Korolija, D., and Alonso, G. Easynet: 100 gbps network for hls. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)* (2021), IEEE, pp. 197–203.

[130] Hoque, T., Yang, K., Karam, R., Tajik, S., Forte, D., Tehranipoor, M., and Bhunia, S. Hidden in plaintext: An obfuscation-based countermeasure against fpga bitstream tampering attacks. *ACM Trans. Des. Autom. Electron. Syst. 25*, 1 (nov 2019).

[131] Hu, C., Liu, B., Zhao, H., Chen, K., Chen, Y., Wu, C., and Cheng, Y. Disco: Memory efficient and accurate flow statistics for network measurement. In *2010 IEEE 30th International Conference on Distributed Computing Systems* (2010), pp. 665–674.

[132] Hu, C., Wang, S., Tian, J., Liu, B., Cheng, Y., and Chen, Y. Accurate and efficient traffic monitoring using adaptive non-linear sampling method. In *IEEE INFOCOM* (2008), pp. 26–30.

[133] Hua, N., Xu, J. J., Lin, B., and Zhao, H. C. Brick: A novel exact active statistics counter architecture. *IEEE/ACM Transactions on Networking 19*, 3 (2011), 670–682.

[134] Huang et al., Q. SketchVisor: Robust network measurement for software packet processing. *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)* (2017), 113–126.

[135] iana. IEEE 802 Numbers. `https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml`, 2023.

[136] infrastructure security agency, C. . BrickerBot Permanent Denial-of-Service Attack. `https://www.cisa.gov/news-events/ics-alerts/ics-alert-17-102-01a`, 2017.

[137] Intel. Comparing fpgas, structured asics, and cell-based asics. Available at `https://www.intel.com/content/www/us/en/products/programmable/fpga-vs-structured-asic.html`.

[138] Intel. Fpga vs. gpu for deep learning. Available at `https://www.intel.com/content/www/us/en/artificial-intelligence/programmable/fpga-gpu.html`.

[139] Intel. Shifting from software to hardware for network security. `https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/wp/wp-01261-shifting-from-software-to-hardware-for-network-security.pdf`, 2016.

[140] Jenkins, B. Algorithm alley. `https://www.drdobbs.com/database/algorithm-alley/184410284`, 1997.

[141] Jiang, W., He, Z., Zhang, S., Zeng, K., Feng, L., Zhang, J., Liu, T., Li, Y., Zhou, J., Zhang, C., et al. Fleetrec: Large-scale recommendation inference on hybrid gpu-fpga clusters. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (2021), pp. 3097–3105.

[142] Jiang, W., Korolija, D., and Alonso, G. Data processing with fpgas on modern architectures. In *Companion of the 2023 International Conference on Management of Data* (2023), pp. 77–82.

[143] Jungk, B., Lima, L. R., and Hiller, M. A systematic study of lightweight hash functions on FPGAs. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)* (2014), IEEE, pp. 1–6.

[144] K. Locke. *Parameterizable Content-Addressable Memory.* `https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf`.

[145] Kaya, I., and Kocak, T. A low power lookup technique for multi-hashing network applications. *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures* (2006).

[146] Kiefer, M., Poulakis, I., Bress, S., and Markl, V. Scotch: Generating fpga-accelerators for sketching at line rate. *Proc. VLDB Endow. 14*, 3 (2020), 281–293.

[147] Kim, S., Jung, C., Jang, R., Mohaisen, D., and Nyang, D. Countless: A counting sketch for the data plane of high speed switches. *arXiv preprint arXiv:2111.02759* (2021).

[148] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467.

[149] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. *Random Struct. Algorithms 33*, 2 (2006), 187–218.

[150] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467.

[151] Kulkarni, A., Chiosa, M., Preusser, T. B., Kara, K., Sidler, D., and Alonso, G. Hyperloglog sketch acceleration on fpga. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)* (2020), IEEE, pp. 47–56.

[152] Kumar, A., Xu, J., and Wang, J. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications 24*, 12 (2006).

[153] Kumar, A., Xu, J., and Wang, J. Space-code bloom filter for efficient per-flow traffic measurement. *IEEE Journal on Selected Areas in Communications 24*, 12 (2006), 2327–2339.

[154] Kumar, S., Turner, J., and Crowley, P. Peacock hashing: Deterministic and updatable hashing for high performance networking. In *IEEE INFOCOM 2008-The 27th Conference on Computer Communications* (2008), IEEE, pp. 101–105.

[155] Kushner, D. The real story of stuxnet. *IEEE Spectrum 50*, 3 (2013), 48–53.

[156] Kwon, M., Reviriego, P., and Pontarelli, S. A length-aware cuckoo filter for faster ip lookup. In *2016 IEEE Conference on Computer Communications Workshops*.

[157] Lai, Y.-K., Huang, P.-Y., Lee, H.-P., Tsai, C.-L., Chang, C.-S., Nguyen, M. H., Lin, Y.-J., Liu, T.-L., and Chen, J. H. Real-time DDoS attack detection using sketch-based entropy estimation on the NetFPGA SUME platform. In *2020 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)* (2020), IEEE.

[158] Lai, Y.-K., Wang, N.-C., Chou, T.-Y., Lee, C.-C., Wellem, T., and Nugroho, H. T. Implementing on-line sketch-based change detection on a NetFPGA platform. In *1st Asia NetFPGA Developers Workshop* (2010).

[159] Lara-Nino, C. A., Morales-Sandoval, M., and Diaz-Perez, A. Small lightweight hash functions in FPGA. In *2018 IEEE 9th Latin American Symposium on Circuits & Systems (LASCAS)* (2018), IEEE, pp. 1–4.

[160] Le Jeune, L., Sateesan, A., Rabbani, M. M., Goedemé, T., Vliegen, J., and Mentens, N. Sok-network intrusion detection on FPGA. In *International Conference on Security, Privacy, and Applied Cryptography Engineering* (2021), Springer, pp. 242–261.

[161] Li, T., Chen, S., and Ling, Y. Fast and compact per-flow traffic measurement through randomized counter sharing. In *2011 Proceedings IEEE INFOCOM* (2011), IEEE, pp. 1799–1807.

[162] Li, Y., Wu, H., Pan, T., Dai, H., Lu, J., and Liu, B. CASE: Cache-assisted stretchable estimator for high speed per-flow measurement. In *IEEE INFOCOM* (2016), pp. 1–9.

[163] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 1–13.

[164] Liu, Z., Manousis, A., Vorsanger, G., Sekar, V., and Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)* (2016), 101–114.

[165] Liu, Z., Namkung, H., Nikolaidis, G., Lee, J., Kim, C., Jin, X., Braverman, V., Yu, M., and Sekar, V. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches. In *30th USENIX Security 21)* (2021).

[166] Liu et al., Z. Nitrosketch: Robust and general sketch-based monitoring in software switches. *Proc. ACM SIGCOMM* (2019), 334–350.

[167] Lu, J., Wan, Y., Li, Y., Zhang, C., H, D., Wang, Y., G.Zhang, and Liu, B. Ultra-fast Bloom filters using SIMD techniques. *Proc. IEEE/ACM IWQoS* (2017).

[168] Lu, J., Yang, T., Wang, Y., Dai, H., Jin, L., Song, H., and Liu, B. One-hashing Bloom filter. *Proc. IEEE/ACM IWQoS* (2015).

[169] Lu, Y., and Prabhakar, B. Robust counting via counter braids: An error-resilient network measurement architecture. In *IEEE INFOCOM* (2009), pp. 522–530.

[170] Luinaud, T., Stimpfling, T., da Silva, J. S., Savaria, Y., and Langlois, J. P. Bridging the gap: Fpgas as programmable switches. In *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)* (2020), IEEE, pp. 1–7.

[171] Luo, L., Guo, D., Rottenstreich, O., Ma, R. T., Luo, X., and Ren, B. The consistent cuckoo filter. In *IEEE INFOCOM 2019* (2019), IEEE, pp. 712–720.

[172] Lyons, M. J., and Brooks, D. The design of a Bloom filter hardware accelerator for ultra low power systems. *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design* (2009), 371–376.

[173] Manku, G. S., and Motwani, R. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases* (2002), Elsevier, pp. 346–357.

[174] Martinasek, Z., Hajny, J., Smekal, D., Malina, L., Matousek, D., Kekely, M., and Mentens, N. 200 Gbps hardware accelerated encryption system for FPGA network cards. In *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security* (2018), pp. 11–17.

[175] Martinasek, Z., Hajny, J., Smekal, D., Malina, L., Matousek, D., Kekely, M., and Mentens, N. 200 Gbps hardware accelerated encryption system for FPGA network cards. In *ASHES* (2018), ACM, p. 11–17.

[176] Mclean, M. Must-know cyber attack statistics and trends. `https://www.embroker.com/blog/cyber-attack-statistics/`, 2023.

[177] Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. *Handbook of applied cryptography*. CRC press, 2018.

[178] Metwally, A., Agrawal, D., and El Abbadi, A. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory* (2005), Springer, pp. 398–412.

[179] Miró-Llinares, F. The impact of covid-19 on cybercrime trends. `https://rm.coe.int/presentation-fernando-miro-llinares-the-impact-of-covid-19-on-cybercri/1680a1e42f`, 2022.

[180] Mitzenmacher, M., Pontarelli, S., and Reviriego, P. Adaptive cuckoo filters, 2020.

[181] Mitzenmacher, M., Reviriego, P., and Pontarelli, S. OMASS: One memory access set separation. *IEEE Transactions on Knowledge and Data Engineering 28*, 7 (2016), 1940–1943.

[182] Mitzenmacher, M., and Upfal, E. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[183] Mitzenmacher, M. D., and Vocking, B. The asymptotics of selecting the shortest of two, improved.

[184] Morris, R. Counting large numbers of events in small registers. *Commun. ACM* (1978), 840–842.

[185] Netcope. . www.netcope.com/getattachment/bb2b8efa-9925-438d-b895-897d7c1e4745/NFB-200G2QL-product-brief.aspx.

[186] Netcope. *www.netcope.com/getattachment/bb2b8efa-9925-438d-b895-897d7c1e4745/NFB-200G2QL-product-brief.aspx*.

[187] Nguyen, D., Memik, G., Memik, S. O., and Choudhary, A. Real-time feature extraction for high speed networks. In *International Conference on Field Programmable Logic and Applications* (2005), pp. 438–443.

[188] NVIDIA, Vingelmann, P., and Fitzek, F. H. Cuda, release: 10.2.89, 2020.

[189] of Standards, N. I., and (NIST), T. Cybersecurity. https://csrc.nist.gov/glossary/term/cybersecurity, 2023.

[190] of Standards, N. I., and (NIST), T. Information security. https://csrc.nist.gov/glossary/term/information_security, 2023.

[191] Ozdal, M. M. Emerging accelerator platforms for data centers. *IEEE Design & Test 35*, 1 (2017), 47–54.

[192] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004).

[193] Pandey, P., Bender, M. A., Johnson, R., and Patro, R. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data* (2017), pp. 775–787.

[194] Pandey, P., Conway, A., Durie, J., Bender, M. A., Farach-Colton, M., and Johnson, R. Vector quotient filters: Overcoming the time/space trade-off in filter design. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 1386–1399.

[195] Proulx, A., Chouinard, J.-Y., Fortier, P., and Miled, A. A survey on fpga cybersecurity design strategies. *ACM Transactions on Reconfigurable Technology and Systems 16*, 2 (2023), 1–33.

[196] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News 42*, 3 (2014), 13–24.

[197] Putze, F., Sanders, P., and Singler, J. Cache-, hash-and space-efficient bloom filters. In *International Workshop on Experimental and Efficient Algorithms* (2007).

[198] Qiao, Y., Li, T., and Chen, S. Fast bloom filters and their generalization. *IEEE Transactions on Parallel and Distributed Systems 25*, 1 (2013), 93–103.

[199] Qiao, Y., Li, T., and Chen, S. Fast Bloom filters and their generalization. *IEEE TPDS 25*, 1 (2014), 93–103.

[200] Ramakrishna, M. Practical performance of bloom filters and parallel free-text searching. *Communications of the ACM 32*, 10 (1989), 1237–1239.

[201] Ramakrishna, M. V., Fu, E., and Bahcekapili, E. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers 46*, 12 (1997), 1378–1381.

[202] Reviriego, P., Christensen, K., and Maestro, J. A comment on fast Bloom filters and their generalization. *IEEE Transactions on Parallel and Distributed Systems 27*, 1 (2016), 303–304.

[203] Richa, A. W., Mitzenmacher, M., and Sitaraman, R. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization 9* (2001), 255–304.

[204] Rivest, R. The md5 message-digest algorithm. Tech. rep., 1992.

[205] Roberts, L. The arpanet and computer networks. In *ACM Conference on history of personal workstations* (1986).

[206] Rothenberg, C. E., Macapuna, C. A., Verdi, F. L., and Magalhaes, M. F. The deletable bloom filter: a new member of the bloom family. *IEEE Communications Letters 14*, 6 (2010).

[207] Roy, P., Khan, A., and Alonso, G. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, 2016), SIGMOD '16, Association for Computing Machinery, p. 1449–1463.

[208] Saavedra, A., Hernández, C., and Figueroa, M. Heavy-hitter detection using a hardware sketch with the countmin-cu algorithm. In *2018 21st Euromicro Conference on Digital System Design (DSD)* (2018), IEEE.

[209] Saavedra, A., Hernández, C., and Figueroa, M. Heavy-hitter detection using a hardware sketch with the Countmin-CU algorithm. In *Euromicro DSD* (2018), pp. 38–45.

[210] Sateesan, A., Biesmans, J., Claesen, T., Vliegen, J., and Mentens, N. Optimized algorithms and architectures for fast non-cryptographic hash functions in hardware. *Microprocessors and Microsystems 98* (2023), 104782.

[211] Sateesan, A., Sinha, S., KG, S., and Vinod, A. A survey of algorithmic and hardware optimization techniques for vision convolutional neural networks on fpgas. *Neural Processing Letters 53* (2021), 2331–2377.

[212] Sateesan, A., Vliegen, J., Daemen, J., and Mentens, N. Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications. In *2020 23rd Euromicro Conference on Digital System Design (DSD)* (2020), IEEE, pp. 262–269.

[213] Sateesan, A., Vliegen, J., Daemen, J., and Mentens, N. Hardware-oriented optimization of bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. *Microprocessors and Microsystems 93* (2022), 104619.

[214] Sateesan, A., Vliegen, J., and Mentens, N. An analysis of the hardware-friendliness of amq data structures for network security. In *International Conference on Security, Privacy, and Applied Cryptography Engineering* (2022), Springer, pp. 287–313.

[215] Sateesan, A., Vliegen, J., Scherrer, S., Hsiao, H.-C., Perrig, A., and Mentens, N. Speed records in network flow measurement on fpga. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, pp. 219–224.

[216] Scherrer, S., Vliegen, J., Sateesan, A., Mentens, N., Hsiao, H.-C., and Perrig, A. ALBUS: a probabilistic monitoring algorithm to counter burst-flood attacks. In *2023 42th International Symposium on Reliable Distributed Systems (SRDS)* (2023), IEEE.

[217] Scherrer, S., Wu, C.-Y., Chiang, Y.-H., Rothenberger, B., Asoni, D. E., Sateesan, A., Vliegen, J., Mentens, N., Hsiao, H.-C., and Perrig, A. Low-rate overuse flow tracer (loft): An efficient and scalable algorithm for detecting overuse flows. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)* (2021), IEEE, pp. 265–276.

[218] Schweller, R., Li, Z., Chen, Y., Gao, Y., Gupta, A., Zhang, Y., Dinda, P. A., Kao, M., and Memik, G. Reversible sketches: Enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking 15*, 5 (2007), 1059–1072.

[219] sFlow. Traffic Monitoring using sFlow. `http://www.sflow.org/sFlowOverview.pdf`, 2003.

[220] Shen, Q. C., Vega, J. C., and Chow, P. Parallel crc on an fpga at terabit speeds. In *2022 International Conference on Field-Programmable Technology (ICFPT)* (2022), IEEE, pp. 1–6.

[221] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., and Rexford, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2017), SOSR '17, Association for Computing Machinery, p. 164–176.

[222] Specht, S., and Lee, R. Taxonomies of distributed denial of service networks, attacks, tools and countermeasures. *CEL2003-03, Princeton University, Princeton, NJ, USA* (2003).

[223] Stanojevic, R. Small active counters. In *IEEE INFOCOM* (2007), pp. 2153–2161.

[224] Szabo-Wexler, E. Approximate membership of sets : A survey.

[225] Tang, L., Huang, Q., and Lee, P. P. C. A fast and compact invertible sketch for network-wide heavy flow detection. *IEEE/ACM Transactions on Networking 28*, 5 (2020), 2350–2363.

[226] Teubner, J., Mueller, R., and Alonso, G. FPGA acceleration for the frequent item problem. In *IEEE ICDE* (2010), pp. 669–680.

[227] Tong, D., and Prasanna, V. High throughput sketch based online heavy hitter detection on FPGA. *ACM SIGARCH Computer Architecture News 43*, 4 (2016).

[228] Tong, D., and Prasanna, V. K. High throughput sketch based online heavy hitter detection on FPGA. *SIGARCH Comput. Archit. News 43* (2016), 70–75.

[229] Tong, D., and Prasanna, V. K. Sketch acceleration on FPGA and its applications in network anomaly detection. *IEEE TPDS 29*, 4 (2017).

[230] Tong, D., and Prasanna, V. K. Sketch acceleration on fpga and its applications in network anomaly detection. *IEEE Transactions on Parallel and Distributed Systems 29*, 4 (2018), 929–942.

[231] Tsidon, E., Hanniel, I., and Keslassy, I. Estimators also need shared values to grow together. In *IEEE INFOCOM* (2012), pp. 1889–1897.

[232] Wada, T., Matsumura, N., Nakano, K., and Ito, Y. Efficient byte stream pattern test using Bloom filter with rolling hash functions on the FPGA. *2018 Sixth International Symposium on Computing and Networking* (2018), 66–75.

[233] Wang, M., and Zhou, M. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment* (2019).

[234] Wang, Z., Huang, H., Zhang, J., Wu, F., and Alonso, G. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022), pp. 967–986.

[235] Weerasinghe, J., Abel, F., Hagleitner, C., and Herkersdorf, A. Enabling fpgas in hyperscale data centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)* (2015), IEEE, pp. 1078–1086.

[236] Weerasinghe, J., Polig, R., Abel, F., and Hagleitner, C. Network-attached fpgas for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)* (2016), IEEE, pp. 36–43.

[237] Wellem, T., Lai, Y., Huang, C., and Chung, W. A hardware-accelerated infrastructure for flexible sketch-based network traffic monitoring. In *IEEE HPSR* (2016), pp. 162–167.

[238] Wellem, T., Lai, Y.-K., Huang, C.-Y., and Chung, W.-Y. A hardware-accelerated infrastructure for flexible sketch-based network traffic monitoring. In *IEEE 17th HPSR* (2016), IEEE.

[239] Wilding, N. An introduction to hardware cyber security: how hard it can be? `https://www.nexor.com/an-introduction-to-hardware-cyber-security-how-hard-can-it-be/`, 2019.

[240] Winderickx, J., Braeken, A., and Mentens, N. Enhanced end-to-end security through symmetric-key cryptography in wearable medical sensor networks. *Health and Technology 11* (2021), 511–523.

[241] Wu, H., Hsiao, H.-C., and Hu, Y.-C. Efficient large flow detection over arbitrary windows: An algorithm exact outside an ambiguity region. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), pp. 209–222.

[242] Wu, H., and Huang, T. Tinyjambu: A family of lightweight authenticated encryption algorithms. the nist lightweight cryptography (lwc) standardization project (a round-2 candidate), 2019. *URL https://csrc. nist. gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/TinyJAMBU-spec-round2. pdf*.

[243] Xiao, Q., Cai, X., Qin, Y., Tang, Z., Chen, S., and Liu, Y. Universal and accurate sketch for estimating heavy hitters and moments in data streams. *IEEE/ACM Transactions on Networking* (2023).

[244] Xie, Z., Ding, W., Wang, H., Xiao, Y., and Liu, Z. D-ary cuckoo filter: A space efficient data structure for set membership lookup. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems*.

[245] Xilinx. XUP Vitis Network Example (VNx). `https://github.com/Xilinx/xup_vitis_network_example`.

[246] Xilinx. Alveo U250 Data Center Accelerator Card. `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html`, 2023.

[247] Xilinx. Vitis unified software platform documentation: Application acceleration development (ug1393). https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration, 2023.

[248] Xilinx, A. UltraScale Architecture DSP Slice User Guide (UG579). `https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp`, 2021.

[249] Xilinx User Guide. *Ternary Content Addressable Memory (TCAM) Search IP for SDNet.* `https://www.xilinx.com/support/documentation/ip_documentation/tcam/pg190-tcam.pdf`.

[250] Yang, T., Gao, S., Sun, Z., Wang, Y., Shen, Y., and Li, X. Diamond sketch: Accurate per-flow measurement for big streaming data. *IEEE Transactions on Parallel and Distributed Systems 30*, 12 (2019), 2650–2662.

[251] Yang, T., Liu, L., Yan, Y., Shahzad, M., Shen, Y., Li, X., Cui, B., and Xie, G. Sf-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (2017), pp. 103–106.

[252] Yang, T., Xu, J., Liu, X., Liu, P., Wang, L., Bi, J., and Li, X. A generic technique for sketches to adapt to different counting ranges. In *IEEE INFOCOM* (2019), pp. 2017–2025.

[253] Yang et al., T. Elastic sketch: Adaptive and fast network-wide measurements. *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)* (2018), 561–575.

[254] Yu, M., Jose, L., and Miao, R. Software defined traffic measurement with opensketch. In *NSDI* (2013), vol. 13, pp. 29–42.

[255] Zadnik, M., Canini, M., Moore, A. W., Miller, D. J., and Li, W. Tracking elephant flows in internet backbone traffic with an FPGA-based cache. In *FPL* (2009), pp. 640–644.

[256] Zazo, J. F., Lopez-Buedo, S., Ruiz, M., and Sutter, G. A single-FPGA architecture for detecting heavy hitters in 100 Gbit/s ethernet links. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)* (2017), IEEE.

[257] Zazo, J. F., Lopez-Buedo, S., Ruiz, M., and Sutter, G. A single-FPGA architecture for detecting heavy hitters in 100 Gbit/s ethernet links. In *ReConFig* (2017), pp. 1–6.

[258] Zhou, Y., Jin, H., Liu, P., Zhang, H., Yang, T., and Li, X. Accurate per-flow measurement with bloom sketch. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2018), pp. 1–2.

[259] Zhou, Y., Liu, P., Jin, H., Yang, T., Dang, S., and Li, X. One memory access sketch: A more accurate and faster sketch for per-flow measurement. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference* (2017), pp. 1–6.

[260] Zhou, Y., Zhang, Y., Ma, C., Chen, S., and Odegbile, O. O. Generalized sketch families for network traffic measurement. *POMACS 3*, 3 (2019).

[261] Zhu, H., Zhang, Y., Zhang, L., He, G., Liu, L., and Liu, N. Sa sketch: A self-adaption sketch framework for high-speed network. *Concurrency and Computation: Practice and Experience 32*, 23 (2020), e5891.

Curriculum Vitae

Arish Sateesan was born on July 4th, 1987, in Varkala, India. He obtained his Bachelor's degree in Electronics & Communication Engineering from Cochin University of Science and Technology, India, in 2009. He worked as a Lecturer at the College of Engineering Kidangoor during 2010-2011 and later joined Gemtech Solutions Pvt Ltd as Systems Engineer in 2011 before commencing his Master's degree in 2013. He obtained his Master's degree in Embedded System Design from the National Institute of Technology Kurukshetra, India. After completing his Master's, he joined IBM as Business Intelligence Application Developer in 2015. Later he moved into academic research at Nanyang Technological University Singapore as Research Associate (Deep Learning and Embedded Systems) in 2018, prior to joining KU Leuven. He joined KU Leuven in April 2019 as a Ph.D. candidate at COSIC (Emerging Technologies, Systems & Security Subgroup), under the supervision of Prof. Nele Mentens and Dr. Jo Vliegen, in collaboration with the Network Security Group, ETH Zurich, on the joint research project ESCALATE. His doctoral research focuses on Network Security, specifically in the acceleration of Large Flow detection algorithms using configurable hardware (FPGA).

List of publications

Journals

1. **Sateesan, A.**, Biesmans, J., Claesen, T., Vliegen, J., and Mentens, N. Optimized algorithms and architectures for fast non-cryptographic hash functions in hardware. Microprocessors and Microsystems 98 (2023), 104782.

2. **Sateesan, A.**, Vliegen, J., Daemen, J., and Mentens, N. Hardware-oriented optimization of bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. Microprocessors and Microsystems 93 (2022), 104619.

Under submission

3. **Sateesan, A.**, Vliegen, J., Scherrer, S., Hsiao, H.C., Perrig, A. and Mentens, N. SPArch: A hardware-oriented sketch-based architecture for all-round network flow measurements.

4. Hassan, M., **Sateesan, A.**, Vliegen, J., Picek, S., Mentens, N., "GENETIC-HASH: Genetic Programming-based, Hardware Oriented, Ultra Fast Hashes for Network Security Applications," IEEE Transactions on Cybernetics.

International conferences

1. Scherrer, S., Vliegen, J., **Sateesan, A.**, Mentens, N., Hsiao, H.-C., and Perrig, A. ALBUS: a probabilistic monitoring algorithm to counter burst-flood attacks. In 2023 42th International Symposium on Reliable Distributed Systems (SRDS) (2023), IEEE.

2. Hassan, M., **Sateesan, A.**, Vliegen, J., Picek, S. and Mentens, N. Evolving Non-cryptographic Hash Functions Using Genetic Programming for High-speed Lookups in Network Security Applications. In International Conference on the Applications of Evolutionary Computation (Part of EvoStar) (2023), Cham: Springer Nature Switzerland.

3. **Sateesan, A.**, Vliegen, J., and Mentens, N. An analysis of the hardware-friendliness of amq data structures for network security. In International Conference on Security, Privacy, and Applied Cryptography Engineering (2022), Springer.

4. Le Jeune, L., **Sateesan, A.**, Rabbani, M. M., Goedemé, T., Vliegen, J., and Mentens, N. Sok-network intrusion detection on FPGA. In Security, Privacy, and Applied Cryptography Engineering: 11th International Conference, SPACE 2021, Kolkata, India (2022), Springer.

5. Scherrer, S., Wu, C.-Y., Chiang, Y.-H., Rothenberger, B., Asoni, D. E., **Sateesan, A.**, Vliegen, J., Mentens, N., Hsiao, H.-C., and Perrig, A. Low-rate overuse flow tracer (loft): An efficient and scalable algorithm for detecting overuse flows. In 2021 40th International Symposium on Reliable Distributed Systems (SRDS) (2021), IEEE.

6. Claesen, T., **Sateesan, A.**, Vliegen, J., and Mentens, N. Novel non-cryptographic hash functions for networking and security applications on fpga. In 2021 24th Euromicro Conference on Digital System Design (DSD) (2021), IEEE.

7. **Sateesan, A.**, Vliegen, J., Scherrer, S., Hsiao, H.-C., Perrig,A., and Mentens, N. Speed records in network flow measurement on fpga. In 2021 31st International Conference on Field-Programmable Logic and Applications (FPL) (2021), IEEE.

8. **Sateesan, A.**, Vliegen, J., Daemen, J., and Mentens, N. Novel Bloom filter algorithms and architectures for ultra-high-speed network security applications. In 2020 23rd Euromicro Conference on Digital System Design (DSD) (2020), IEEE.

Under submission

9. Sponziello, N., **Sateesan, A.**, Rabbani, M., Mentens, N., Dragoni, N., Dushku, E., "ITERATOR: Interruptible Remote attestation through Cuckoo filters," The 14th ACM Conference on Data and Application Security and Privacy (CODASPY 2024).



FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Wetenschapspark 27 bus 15152
3590 Diepenbeek
first.last@cs.kuleuven.be
http://www.cosic.be