Atlas: Application Confidentiality in Compromised Embedded Systems

Pieter Maene, Johannes Götzfried, Tilo Müller, Ruan de Clercq, Felix Freiling, and Ingrid Verbauwhede

Abstract—Due to the requirements of the Internet-of-Things, modern embedded systems have become increasingly complex, running different applications. In order to protect their intellectual property as well as the confidentiality of sensitive data they process, these applications have to be isolated from each other. Traditional memory protection and memory management units provide such isolation, but rely on operating system support for their configuration. However, modern operating systems tend to be vulnerable and cannot guarantee confidentiality when compromised. We present Atlas, a hardware-based security architecture, complementary to traditional memory protection mechanisms, ensuring code and data confidentiality through transparent encryption, even when the system software has been exploited. Atlas relies on its zero-software trusted computing base to protect against system-level attackers and also supports secure shared memory. We implemented Atlas based on the LEON3 softcore processor, including toolchain extensions for developers. Our FPGA-based evaluation shows minimal cycle overhead at the cost of a reduced maximum frequency.

Index Terms—Trusted computing, security hardware, embedded systems, confidentiality

1 INTRODUCTION

Embedded systems are a core component of many products and they are increasingly networked, driven by the development of the Internet of Things (IoT). However, this exposes them to a much larger attack surface, explaining the need for lightweight security mechanisms to protect them. For instance, modern cars rely on microcontrollers, interconnected by a Controller Area Network (CAN), for a variety of functions from controlling the brakes and engine to on-board entertainment. Driven by the increasing complexity of microcontrollers, and in an effort to simplify architecture design and save cost, manufacturers are integrating functionality onto a smaller number of those microcontrollers [28]. This means that sensitive applications now run alongside non-critical ones, increasing the need for security mechanisms to protect confidentiality and integrity. Among others, the engine control algorithms are important Intellectual Property (IP), and its parameters ensure that the car runs as designed.

However, Operating Systems (OSs) have been shown to be vulnerable in the past, leading to code and data compromise in some cases. For instance, Dirty COW [29] is a privilege escalation vulnerability based on a bug in the way Linux handled copy-on-write memory, allowing an attacker to gain write access to otherwise read-only memory. At a lower level, Google’s Project Zero discovered a vulnerability in the Wi-Fi stack of Broadcom chips [3], enabling a remote adversary to execute arbitrary code on its ARM Cortex R4 running the firmware. Furthermore, this exploit eventually led to code execution in the kernel running on the host de-

vice’s main processor [4]. These Wi-Fi chips run a very basic OS (HND RTE), and while the attackers did not compromise it directly, it also does not feature many common security features, allowing memory allocation bugs to be exploited. Therefore, lightweight protection mechanisms are needed to protect the confidentiality of those algorithms, even when an attacker compromises the system’s OS and can tamper with any software running on the device.

In this paper, we focus on protecting the confidentiality of code and data against system-level attackers through transparent memory encryption. Our solution is designed to be complementary to traditional Memory Protection Units (MPUs), which are configured by the OS in order to isolate the memory regions of different applications. However, when the OS has been compromised, security, and especially confidentiality of code and data, can no longer be guaranteed. We ensure confidentiality even in the event of a system compromise, which necessarily requires hardware-based solutions. Once applications start using these hardware-assisted protection mechanisms, there also needs to be a way for them to communicate reliably and securely. In addition, compared to existing trusted computing mechanisms for these lightweight processors, e.g., based on boundary registers [27], our solution has lower area overhead, which is fixed for any number of applications.

Our Contributions

This paper introduces Atlas, a hardware-based security mechanism protecting application confidentiality against system-level attackers, with a fixed overhead that is independent of the number of applications running on the system. Furthermore, Atlas enables the use of shared memory as a lightweight and easy-to-use secure communication channel. In detail, our contributions are:

- We propose the use of hardware-based memory encryption to protect application confidentiality in embedded

• P. Maene, R. de Clercq, and I. Verbauwhede are with imec-COSIC, Department of Electrical Engineering (ESAT), KU Leuven, Belgium.
E-mail: {pieter.maene, ruan.declercq, ingrid.verbauwhede}@esat.kuleuven.be

• J. Götzfried, Tilo Müller, and Felix Freiling are with the Department of Computer Science, FAU Erlangen-Nürnberg, Germany.
E-mail: {johannes.goetzfried, tilo.mueller, felix.freiling}@cs.fau.de

systems. In particular, our solution protects confidentiality in the event of system compromise, including a potentially compromised OS.

- We ensure that neither code nor data leaks to any other application or the OS, relying on a zero-software Trusted Computing Base (TCB). Since there is no need to keep track of state information per application, our solution scales to an unlimited number of applications.
- We provide confidential shared memory, which can be used as a communication channel between multiple applications, without the need for a dynamic key exchange.
- We designed and implemented Atlas by extending the open source LEON3 processor. This includes a host toolchain to compile C programs for our architecture.
- We evaluated the software and hardware implementation of Atlas regarding performance and area. Atlas has 0.031% cycle overhead compared to an unmodified binary for a real-world signing application, at the cost of a four times slower maximal clock and 46.595% area increase.

All code we developed to run applications on the modified core, including the hardware, toolchain, and software implementations are open source and can be downloaded from <https://esat.kuleuven.be/cosic/software/atlas/>.

2 ARCHITECTURE

This section first presents our attacker model (Section 2.1). Next, Section 2.2 discusses Atlas’ system model, and finally the design of its architecture is detailed in Section 2.3.

2.1 Attacker Model

In our model, we assume the attacker wants to extract confidential IP (e.g., proprietary algorithms) from the application’s code. Furthermore, he is also looking to obtain confidential data processed by it, which was either statically compiled or dynamically calculated at runtime. The attacker has system-level privileges, i.e., he can exploit any piece of software running on the device, including the OS. As long as the OS has not been compromised, an MPU ensures that applications only access their own memory. When an attacker has obtained system-level privileges, though, he can read from and write to any memory location. Denial-of-Service (DoS) attacks are considered to be out of scope. Following the Dolev-Yao model [8], the cryptographic primitives used in our scheme cannot be broken, but protocol-level attacks are allowed.

In addition to controlling any software, the attacker can physically probe main memory. However, we assume he does not have access to the CPU’s internal registers or caches. Invasive attacks where the chip is decapsulated are therefore excluded. This is a reasonable assumption, since such attacks require a high level of technical skill, expensive equipment, and take a long time to plan and execute. For example, Tarnovsky’s attack on the Infineon SLE 66 microcontroller took six months from planning to execution [35].

2.2 System Model

Encrypting memory transparently under a single key is not sufficient to protect against a system-level attacker, as such a system could not track ownership and would return any requested data in plaintext. Therefore, the device’s system model has to meet two requirements. First, all calls to any confidential application have to pass through its entry point, and applications therefore need to know each other’s location. Second, an application should not be able to relocate itself to the entry point of another protected application, as this would give it access to that application’s confidential code and data. The entry point corresponds to the first instruction being executed when an application is called. Atlas satisfies the first constraint by creating a static layout of all applications running on a single device. Since decryption will fail when an attacker moves his application and because it is hard for him to generate a correctly encrypted binary himself, the code encryption mitigates the second issue. Note that applications are expected to yield control when finished, as preemption is not supported.

In addition to the device key K_D , the current implementation of Atlas also uses a tweak key F (see Section 3.1). Both keys are unique for each device, and generated by the system integrator. They are hardwired in the silicon, e.g., by blowing fuses of the manufactured device.

The secure shared memory feature relies on pre-shared secrets. Because a confidential application’s static data is encrypted, the communication keys can be stored securely in memory and decrypted when necessary. Generating these keys, defining the regions where the applications can read and write securely shared data, and updating the binary with these parameters are also done by the integrator.

2.3 Architecture Design

Atlas’ encryption unit protects the confidentiality of applications sharing the same address space. Once memory protection mechanisms relying on software support have been compromised, applications can read from or write to any given address. However, the entry point is used as a unique Initialization Vector (IV), binding dynamically encrypted data to its application. While a system-level attacker has the ability to read any location, he will be unable to recover the correct plaintext when trying to access protected memory.

When the OS has been compromised, the MPU can no longer be trusted to protect against an attacker modifying memory. As shown in Section 2.2, code encryption prevents an attacker from relocating his code to another application’s entry point. Although an adversary can now write to any memory location, data encryption cannot be configured independently and thus, code needs to be encrypted as well. This increases the attack complexity, as any instruction manipulating memory needs to be encrypted. Since the attacker does not know the encryption key, it is hard for him to obtain the instruction’s ciphertext. Consequently, Atlas protects the confidentiality of code and data against all software attacks including relocation attacks.

2.3.1 Encryption Unit Properties

The encryption unit is considered to have the following properties: first, in order to protect the confidentiality of

Fig. 1. The memory hierarchy was modified to include an encryption unit. Encryption and decryption take place right before code and data enter or leave the cache, manipulating the values read from and written to memory before they are communicated over the bus.

different applications, it is able to identify the application to which the current memory bus request belongs. Second, as one of the design goals is to build a scalable architecture, it has to be stateless. Finally, to support secure shared memory, it should be possible to dynamically reconfigure the symmetric key used for data encryption to one that is shared among the communicating applications. We will discuss the implementation of these properties in Section 3.1.1.

2.3.2 Hardware Architecture

As shown in Figure 1, the encryption unit is inserted between the cache and main memory. Once it is turned on, confidential instructions will be automatically decrypted when read, and data will be decrypted and encrypted transparently when entering or leaving the cache. Remember that it is assumed to be impossible for attackers to read the processor’s caches or internal registers (Section 2.1). To prevent leakage, our hardware and toolchain respectively take care of flushing both caches, and clearing all registers when the encryption unit mode is changed (e.g., when turning encryption on).

The encryption unit is controlled through custom instructions that were added to the Instruction Set Architecture (ISA). They are executed by the application itself and can be used to turn encryption on or off, e.g., when it does not need confidentiality or in case it wants to access unprotected memory. Additional instructions are available to configure and use secure shared memory.

2.3.3 Software Architecture

In order to decrypt encrypted code and dynamically protect data, the currently executing application has to be identifiable. An entry point is therefore created for each application, which is the very first instruction that has to be called when execution of an application is started, and takes care of setting the application’s identity and switching the encryption context. Since all local and global functions are encrypted, as well as its static data, they will not be decrypted correctly unless the application was called through its entry point. During secure execution, an application can turn off data encryption, e.g., to write out a final result, but code encryption remains switched on until the application exits. Furthermore, applications are able to call unprotected code, but then any affected data will be processed in clear. Protected applications therefore cannot rely on shared libraries to handle sensitive data, but have to include the required functionality in their own binary, i.e., link against those libraries statically.

Fig. 2. The encryption unit was added to the LEON3’s cache. When encryption is turned off, the original instruction and data signals are sent to the bus; otherwise, they are routed through the encryption unit. Note that only the control signals for the encryption unit are shown.

Applications are not tied to a specific region, but instead code and data of each application can be spread over the entire address space. In particular, the stack is shared between applications, and the registers of each application are saved to and restored from this single stack. Due to encryption context switching, stack data, including saved registers, is encrypted with a different IV for each application.

3 IMPLEMENTATION

We implemented Atlas by modifying the LEON3 processor from Gaisler, a 32-bit SPARCv8 architecture with a seven-stage pipeline and instruction and data caches. Furthermore, a software toolchain was developed to provide the required functionality to compile applications for our platform.

3.1 Hardware

The hardware implementation of Atlas consists of two main parts: first, a newly designed encryption unit with the properties described in Section 2, and second, custom instructions were added to the integer unit to configure and control memory encryption. Figure 2 shows how the LEON3 architecture was modified.

3.1.1 Encryption Unit

So far, the encryption unit was described as a building block which satisfies three properties: it can identify the currently running application, encrypts data without storing state, and has a reconfigurable key (Section 2). Our implementation stores the identifier of the active application in a dedicated register, which can only be updated through a custom instruction. The device key K_D is always used, except when the encryption unit is configured to secure shared memory. In that case, the unit switches to the secure shared memory key K_S , which is stored in a dynamically configurable dedicated register. Note that this key is only used to encrypt and decrypt shared data, with K_D still being used to decrypt protected code.

Figure 3 shows a diagram of the encryption unit. The LRW tweakable mode of operation [24] is used to realize stateless encryption of a single 32-bit word. The tweak ensures that every message is unique. In this mode, the ciphertext C is calculated as follows:

$$C = E_K(P \oplus X) \oplus X$$

$$X = F \otimes I$$

Fig. 3. The encryption unit uses SIMON 32/64 in the LRW tweakable mode of operation. The tweak is a multiplication in the finite field $GF(2^{64})$ of a tweak key F and IV, which is the concatenation of the application identifier and the current memory address. The encryption key can be switched from the fixed device key K_D to a configurable pre-shared key K_S when secure shared memory is used.

where P is the plaintext, X the tweak, E_K encryption with key K , F the tweak key, and I the IV. Atlas uses the concatenation of the application identifier and the memory address that is being read from or written to as the IV. Both values are 32-bit, so therefore the tweak key F also has to be 64 bits long and the finite field used for the multiplication is $GF(2^{64})$. X is then truncated to 32 bits before XORing it with the plaintext and output of the cipher respectively.

Since any block cipher can be used in this mode of operation, the choice of algorithm is determined by the word size of the CPU architecture. The LEON3 is a 32-bit architecture where values are read from and written to memory at word granularity. In order to reduce the complexity of the memory controller, a 32-bit block cipher was selected. Additionally, a low-latency single-cycle implementation was used to ensure there is no additional cycle overhead for memory accesses, and to keep the critical path as short as possible. SIMON 32/64 [2] was shown to be the fastest and smallest algorithm with 32-bit blocks [25]. Currently, none of the alternatives with longer keys have low latencies (e.g., KATAN supports 80-bit keys, but has a two times longer critical path). Although 64-bit keys offer short term protection against small organizations [10], we recommend using PRINCE [6] in the case of a 64-bit architecture. PRINCE has 64-bit blocks and 128-bit keys, and is the fastest single-cycle cipher currently available, with very competitive area [25].

LRW is a tweakable mode of operation, like XTS which is now widely used to encrypt block devices like hard disks [12], [11]. The reason for choosing LRW over XTS was that the latter passes through the block cipher twice for each block, which would result in a longer critical path. LRW has a known weakness when the plaintext contains the tweak key F . Since the tweak key register is not accessible directly from software, this is not an issue in our design. In contrast to other modes of operation (e.g., CTR mode or CFB), LRW requires an implementation of the cipher's decryption function.

Since the memory is never read and written at the same time, it is possible to reuse encryption components for decryption as an optimization. SIMON is a Feistel cipher,

where decryption is almost identical to encryption, except that the inputs have to be swapped and the key schedule has to be reversed. Furthermore, SIMON's key expansion is linear, thus it can also be performed in parallel to the round functions for decryption. Therefore, Atlas also includes a *decryption key* consisting of the last four subkeys in order to initialize the key expansion when decrypting. For Feistel ciphers where the key expansion is not linear (e.g., SIMECK [38]), it cannot be calculated in parallel to the round functions, and either all subkeys should be fixed in hardware or they would have to be calculated before the round functions are applied. The former would negatively impact the implementation's area, while the latter would significantly increase the critical path. In general, we suggest the use of a block cipher where encryption and decryption share functionality, and where low-latency single-cycle implementations can be built. Note that this does incur the area and latency cost of additional multiplexers where signals are driven differently when the unit is respectively encrypting or decrypting.

3.1.2 Custom Instructions

Atlas extends the LEON's integer unit with eight new instructions to give software developers access to the new security features:

ENCENTER stores the current value of the program counter in the identifier register and turns on encryption. It is the first instruction that has to be called at the entry point of any confidential application.

ENCEXIT clears all registers of the encryption unit and turns off encryption. It has to be called whenever there is an exit from a confidential application.

ENCPAUSE turns *data* encryption off without clearing any registers. An application which wants to write to unprotected memory needs to call this instruction first.

ENCRESUME turns *data* encryption on with the currently saved settings, usually resuming confidential execution of the currently running confidential application.

ENCSHMON turns on shared memory encryption. This instruction switches the data encryption key to K_S and uses zeros instead of the application identifier.

ENCSHMOFF turns off shared memory encryption without clearing K_S and resumes isolated execution by switching back to K_D .

ENCSETKEY **ENCSETKEY** and **ENCSETDKEY** are used to set the encryption and decryption key for the SIMON cipher used in secure shared memory. The full 64-bit key is passed within two general purpose registers.

To prevent data leakage, the hardware ensures that the instruction and data cache are always flushed when encryption is enabled or disabled, i.e., when **ENCENTER** and **ENCEXIT** are dispatched. The data cache is not flushed during **ENCPAUSE**, **ENCRESUME**, **ENCSHMON**, or **ENCSHMOFF**, as they are executed by protected code which can be assumed to not leak confidential information. Finally, this also means that except for **ENCENTER**, these instructions will always be encrypted in the binary.

3.2 Software

In order to use Atlas' features, the new instructions need to be dispatched at some point. To this end, we developed

a toolchain to expose the functionality to programmers as transparently as possible. With our toolchain, usual C programs can be compiled and linked for the modified core, while the programmer only needs to properly divide the functionality into confidential and unprotected code. On a high level, we use ELF rewriting with relocatable object files and executable files, i.e., no compiler patch is needed. Our toolchain can therefore be easily combined with other existing toolchains.

3.2.1 Confidential Applications

Code and data of a confidential application is transparently protected by the encryption unit. With our toolchain, the programmer can define which files constitute such a confidential application. The remaining functionality of all other source files is considered to be unprotected. Each application can be written in standard C code, and programmers have the ability to annotate their code with macros. These enable them to call into other confidential applications without data leakage besides the supplied parameters.

3.2.2 Control Flow Rewriting

After each confidential application has been compiled, our toolchain parses all relocatable object files and identifies calls from unprotected code to a confidential application, or vice versa. These calls are then rewritten to go through entry and exit routines, which take care of switching the encryption context. Identifiers for the target function as well as the originating function and application are passed in registers, preserving the original control flow.

The context of a confidential application, i.e., all callee-saved registers, is saved and cleared before the context switch, and restored afterwards. Caller-saved registers which are not used for passing arguments are cleared to ensure that no data leaks.

3.2.3 Encryption

Since our toolchain supports standard C code, we also provide built-in support for encrypting confidential applications. The code and static data of each application are both placed in separate text and data sections, except for the entry and exit stubs. After the linking step, our toolchain parses the executable file for both sections and transparently encrypts them. Furthermore, it locates all stubs belonging to the application in the main text section, and also encrypts those.

One implementation aspect we would like to discuss explicitly is the encryption of the GCC integer library routines. On platforms where hardware support for certain mathematical functionality is not available, the compiler automatically inserts code implementing the missing operators. This only happens during the final link stage, and these routines are therefore only inserted into the binary once. Since this is done transparently to the programmer, they could be called from confidential code as well. One solution would be to keep these functions in unprotected code, and perform the same control flow rewriting as for usual unprotected functions. However, this would incur the overhead of switching the encryption context on every call and also mean that their parameters are passed in clear.

Our toolchain therefore ensures copies of these functions are added to each protected module by partially linking its sources first. The compiled object is then encrypted like any other code in the protected application.

3.2.4 Atlas Library

While most of our software implementation is part of the toolchain, we also provide a library for programmers. Besides macros for annotation, we provide library functions for copying data between confidential applications and unprotected code, and template functions for opening and accessing secure shared memory sections between different applications. Helper functions are provided to set a shared precomputed key and to copy from and to these sections. Furthermore, we provide a generator to create these routines for an arbitrary number of applications.

4 EVALUATION

In this section, Atlas is evaluated regarding performance and area. We obtained results for the Digilent Atlys and Xilinx ML605 development boards, which have Xilinx Spartan 6 and Virtex 6 FPGAs respectively. Xilinx ISE 14.7 was used for synthesis, place, and route. Next, Section 4.3 will informally argue the security of our design.

4.1 Performance

4.1.1 Critical Path

Single-cycle implementations of encryption algorithms result in long combinational circuits which impact the critical path. Since the memory hierarchy is part of a processor's critical path, the maximum clock frequency of our design is reduced compared to the original design. On the Atlys, the original design can run at a maximum frequency of 78.57 MHz, whereas Atlas can be clocked at 19.05 MHz. We saw similar results on the ML605, where the original maximum frequency of 109.09 MHz was reduced to 31.58 MHz. Embedded systems, however, are typically designed for low power, and therefore not clocked at the maximum possible frequency [7]. Consequently, the actual overhead depends on the application. If the maximum possible frequency of the current design would not be sufficient, the cipher could be serialized to improve performance, trading latency for delay on memory operations.

4.1.2 Microbenchmark

Two microbenchmarks have been run on our evaluation platform to measure the performance impact of our toolchain. The first is an application which invokes a confidential one that simply returns. To show the overhead between entering a confidential application and a regular call, we compiled this application with a vanilla GCC toolchain as well as our modified one. The former finishes in 87 cycles, while the latter executes in 227 cycles. The secure context switch and cache flush, which ensure that no confidential data will leak, are responsible for this overhead.

The second benchmark copies 1 KB of data from a confidential application to unprotected memory. This requires encryption to be switched off and on repeatedly, as each data element needs to be loaded into a register while encryption

is enabled and written back to memory after it has been disabled. This operation is 4.557 times slower than `memcpy`, which is again caused by the cache flushes.

4.1.3 Macrobenchmark

To demonstrate the overhead Atlas imposes on real world applications, we wrote an example signing application, which consists of a confidential application with static encrypted data and unprotected code. A message is passed from unprotected code to the confidential application, where it is signed with an asymmetric private key stored securely in the static data section. The signed message is then passed back to unprotected code, where the signature is verified with the corresponding public key. In addition to the overhead imposed by the confidential application call, the message has to be copied from unprotected memory to protected and vice versa. The TweetNaCl [5] library is used to generate and verify the signature.

We compiled this application with an unmodified GCC toolchain and our modified one. When the LEON3 issues partial writes, only the modified bytes are sent over the bus, breaking encryption which requires the full word. Therefore, `stb` or `sth` cannot be used in our current prototype. Consequently, the benchmark was run with data encryption disabled. However, since all other modifications to the core remained in place (e.g., cache flushes) and as the cipher implementation is single-cycle with the design clocked at the same frequency, the performance results are not affected. For both binaries, the execution time was measured with and without copying to and from protected memory. The binary which has all Atlas features enabled imposes an overall overhead of 0.031% compared to the GCC-compiled binary without any secure copies. When secure copies are disabled in the binary compiled with our toolchain, execution takes on average 449 cycles longer than the 1,625,595 cycles of the reference binary. When compiled with GCC and secure copies enabled, the overhead is equal to 0.019%. Recall that both caches are flushed during the execution of `ENCENTER` and `ENCEXIT`, which contribute significantly to the reported overhead. For comparison, when the toolchain-compiled binary compiled with copies enabled is executed on an Atlas core where these flushes were removed, the overhead drops to 0.021%.

4.2 Area

The area usage of Atlas was measured after Xilinx ISE finished place and route. An unmodified LEON3 synthesized with the same settings occupies 2,496 slices on the Atlys. Atlas occupies 3,659 slices, resulting in an overhead of 46.595% (Table 1). To reduce the number of required gates, the same cipher core is reused for encryption and decryption (Section 3.1.1). Although SIMON is the smallest cipher currently available, cryptography remains expensive in terms of area, especially in case of single-cycle implementations. As mentioned earlier, a serialized implementation could also further improve the area requirements.

4.3 Security

The goal of Atlas is to protect the confidentiality of code and data on embedded systems, even when the device’s OS has

TABLE 1
Area in terms of registers, Look-Up Tables (LUTs) and occupied slices of an unmodified LEON3, compared to our core on FPGA.

	Unmodified	Atlas	Overhead
Digilent Atlys			
Slices	2,496	3,659	46.6%
Registers	3,070	3,333	8.6%
LUTs	6,261	9,726	55.3%
Xilinx ML605			
Slices	5,519	7,970	44.4%
Registers	11,021	12,046	9.3%
LUTs	13,070	18,482	41.4%

been compromised. This is realised by adding an encryption unit to the memory hierarchy, which transparently encrypts any data leaving the processor and decrypts incoming transfers. The encryption unit is controlled through a set of dedicated instructions (Section 3.1.2). As discussed, `ENCENTER` is the first instruction of protected applications and the only instruction stored in plain. When called, the current value of the program counter is copied to the dedicated identifier register. Since this instruction has to be executed for this register to be set, an attacker or malicious OS cannot directly control its value. Consequently, this prevents the encryption unit from being used as a decryption oracle. Furthermore, an attacker cannot replace the code following `ENCENTER`, e.g., to read out secrets included in the binary, as this requires knowledge of K_D . Finally, note that cleartext code and data are stored in the processor’s caches. Considering that an attacker cannot generate correctly encrypted code, he would first have to turn off encryption if he were to try and access cached code or data. However, recall that all caches are flushed from hardware when `ENCEXIT` is executed (Section 2.3.2).

When the secure shared memory functionality is used, the encryption unit operates differently. In particular, the application identifier is set to zero and K_S is used for encryption, which can be set dynamically. The security of this mode hinges on the fact that each application accessing the secure shared memory region includes K_S as static data, which is encrypted using K_D and the application identifier. The attacker therefore is not able to learn this key, as it is secured by the encryption unit.

Lastly, we also protect against some classes of physical attacks, specifically main memory probing. This relies on the fact that the encryption unit is inserted between main memory bus and the caches (Figure 2). Code and data are therefore only decrypted within the processor’s boundaries and there is no point for a probing attacker where he can tap cleartext from the bus nor for him to read confidential data directly from main memory.

5 RELATED WORK

Many solutions guaranteeing code and data confidentiality have already been proposed. This section first discusses software-based memory encryption approaches, and then presents hardware-based architectures.

5.1 Software-Based Memory Encryption

Software-based memory encryption solutions [22] can be used to ensure confidentiality of code and data. This has been done at different levels of the memory hierarchy, from protecting only swap spaces [33], to process memory ranges [9], [17], and even the whole RAM [32], [14]. While software-based memory encryption has the advantage of compatibility, it also negatively impacts performance and, more importantly, can only prevent memory probing attacks. Furthermore, it cannot protect applications from a system-level attacker.

CPU-based encryption is somewhat related to our work but only protects a small fraction of sensitive data. Symmetric encryption schemes range from register-based approaches as an OS patch [36], [34] to solutions relying on hypervisors [15] and cache-based schemes [30]. There are even schemes for asymmetric encryption algorithms, as it turned out that asymmetric keys can be recovered from memory as well [21], [31]. In particular, RSA implementations exist that are either register-based [13], [18] or rely on hardware transactional memory [19]. However, all these solutions just keep the encryption key and intermediate data out of memory but not any other sensitive information, because they only have limited secure storage available. In contrast, our encryption unit is inserted directly in the memory hierarchy between the cache and main memory, ensuring that confidential code or data is protected as soon as it leaves the processor package.

As mentioned before, full-disk encryption [12], [11] uses similar cryptographic mechanisms to protect data at rest, as they address a similar problem. However, these solutions deal with much larger storage sizes than Atlas and also have very different latency requirements. Current solutions typically rely on the XTS mode of operation [23], [26]. XTS is almost identical to LRW, its main differences being that the tweak i is first encrypted and that the second multiplicand is equal to α^j , where j is the IV. In addition, if the last plaintext block is smaller than the block size, it is padded with bits from the previous ciphertext. Finally, when applied to standard sector-level disk encryption, data units typically correspond to logical blocks [23]. Note that disk encryption solutions are length-preserving and therefore do not authenticate the encrypted data, instead relying on the fact that the ciphertext is not malleable [12]. Atlas was similarly designed to transparently encrypt and decrypt memory, but does use application-specific keys to prevent different applications from accessing unauthorized data.

5.2 Hardware-Based Memory Encryption

Recently, hardware-supported security mechanisms have seen a lot of interest. Such so-called Protected Module Architectures (PMAs) strictly isolate applications from each other and the OS by performing certain checks on every memory access. Intel recently announced Software Guard Extensions (SGX) [1], which provides a general hardware base for strict isolation of applications on x86. To protect the confidentiality of applications in untrusted memory, the Memory Encryption Engine (MEE) dynamically encrypts code and data leaving the cache [20]. SGX uses multiple configuration structures which are stored in memory, nor is

its hardware overhead considered lightweight. In contrast, Atlas focuses on guaranteeing confidentiality of applications running on embedded devices.

Researchers at IBM also proposed an architecture called SecureBlue++ [37], protecting the confidentiality and integrity of an application's cache lines when they are evicted to main memory. Although SecureBlue++ provides integrity in addition to confidentiality, an important difference compared to Atlas is that its confidentiality protection mechanism relies on hardware implementations of several cryptographic primitives, thus drastically increasing the memory controller's complexity. Binaries are encrypted using an *executable key*, which is itself encrypted asymmetrically and decrypted in hardware when entering the secure mode. While more flexible in terms of key distribution compared to Atlas, this means that an expensive hardware implementation of an asymmetric algorithm is required as well.

For embedded systems, many solutions build on the concept of PMAs. Sancus [27] is a security architecture for lightweight devices, providing isolation and attestation. Its memory protection mechanism consists of a combinational circuit which checks the current memory address against a set of boundary registers. Two pairs of registers are added, storing the start and end addresses of the text and data section respectively. Access to the memory regions specified by the registers is then restricted based on the current value of the processor's program counter. Soteria [16] further extends Sancus, protecting intellectual property at load time through encryption and during runtime with the help of Sancus' dedicated memory access logic in hardware.

All these lightweight solutions have in common that they need to maintain state per application. Furthermore, on similar FPGAs, the overhead of Atlas in terms of LUTs, is comparable to the fixed LUT overhead of Sancus. Atlas requires significantly fewer registers, but has a greater impact on the critical path, at least until ciphers with lower latency become available. Finally, in contrast to Atlas, most solutions relying on PMAs cannot be used with more complex memory hierarchies including caches.

6 DISCUSSION

In this section, we first discuss current limitations of Atlas, followed by possible future improvements.

6.1 Limitations

Atlas does not support SPARC register windows, requiring software to be compiled flatly. The reason is that overflowing or underflowing register windows triggers an interrupt, which currently cannot be handled by Atlas. Enabling interrupts in the current design would violate our security policy, as they circumvent the encryption context switch.

Furthermore, function pointers cannot be used for calls between applications. Since it is impossible to reliably determine the destination address of calculated calls at compile or link time, our toolchain could not rewrite control flow to jump through the application's entry point which initializes the encryption unit.

6.2 Future Work

Encrypting on word granularity leads to small block sizes of 32 bits, which would change when porting our design to a 64-bit architecture, allowing stronger algorithms to be used (e.g., PRINCE [6]). Alternatively, two words could be encrypted simultaneously, but this would significantly complicate the encryption unit's design and impact performance, as reads would require both words to be fetched. Similarly, writes might incur a read, because encryption always needs to be performed with both words. Finally, encryption could be done on cache line granularity, respectively encrypting and decrypting when lines are flushed and loaded.

As was mentioned before, serializing the cipher would improve the clock frequency overhead and further reduce the area requirements of the design. This would come at a cycle cost for each memory access, because the processor would have to wait for the encryption unit to finish. Therefore, a good tradeoff would have to be found.

7 CONCLUSION

We presented Atlas, a scalable security architecture which provides code and data confidentiality for applications through hardware-based memory encryption. Atlas protects IP against system-level attackers in the event of a complete system compromise, using unique IVs for each application. Furthermore, it has a zero-software TCB and also protects against physical attacks on main memory. Our FPGA implementation on the SPARC LEON3 shows that an existing microcontroller can be extended to include our proposed features with negligible cycle overhead, at the cost of a reduced maximum clock frequency and increased area.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported in part by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89), the KU Leuven Research Council through C16/15/058, and ERC Advanced Grant 695305. Pieter Maene is an SB PhD fellow at Research Foundation - Flanders (FWO).

REFERENCES

- [1] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," 2013.
- [2] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK Families of Lightweight Block Ciphers," Cryptology ePrint Archive, Report 2013/404, 2013.
- [3] G. Beniamini, "CVE-2017-6956," MITRE, 2017.
- [4] —, "CVE-2017-6975," MITRE, 2017.
- [5] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, "TweetNaCl: A Crypto Library in 100 Tweets," in *Progress in Cryptology - LatinCrypt 2014*, 2014.
- [6] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knežević, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalcın, "PRINCE: A Low-latency Block Cipher for Pervasive Computing Applications," Cryptology ePrint Archive, Report 2012/529, 2012.
- [7] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power CMOS Digital Design," *IEICE Transactions on Electronics*, vol. 75, no. 4, 1992.
- [8] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, 1983.
- [9] G. Duc and R. Keryell, "CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection," in *Proceedings of the 22nd Computer Security Applications Conference*, 2006.
- [10] ECRYPT II, "Yearly Report on Algorithms and Keysizes," 2012.
- [11] N. Ferguson, "AES-CBC+ Elephant Diffuser: A Disk Encryption Algorithm for Windows Vista," 2006.
- [12] C. Fruhwirth, "New Methods in Hard Disk Encryption," Tech. Rep., 2005.
- [13] B. Garmany and T. Müller, "PRIME: private RSA Infrastructure for Memory-Less Encryption," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.
- [14] J. Götzfried, N. Dörr, R. Palutke, and T. Müller, "HyperCrypt: Hypervisor-based Encryption of Kernel and User Space," in *11th International Conference on Availability, Reliability and Security (ARES'16)*, 2016.
- [15] J. Götzfried and T. Müller, "Mutual Authentication and Trust Bootstrapping towards Secure Disk Encryption," *Transactions on Information and System Security*, vol. 17, 2014.
- [16] J. Götzfried, T. Müller, R. de Clercq, P. Maene, F. Freiling, and I. Verbauwhede, "Soteria: Offline Software Protection within Low-cost Embedded Devices," in *Proceedings of the 31st Computer Security Applications Conference*, 2015.
- [17] J. Götzfried, T. Müller, G. Drescher, S. Nürnberger, and M. Backes, "RamCrypt: Kernel-based Address Space Encryption for User-mode Processes," in *Proceedings of the 11th Conference on Computer and Communications Security*, 2016.
- [18] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with Private Keys without RAM," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.
- [19] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *Proceedings of the 36th Symposium on Security and Privacy*, 2015.
- [20] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," Cryptology ePrint Archive, Report 2016/204, 2016.
- [21] N. Heninger and H. Shacham, "Reconstructing RSA Private Keys from Random Key Bits," in *Proceedings of the 29th Annual International Cryptology Conference*, 2009.
- [22] M. Henson and S. Taylor, "Memory Encryption: A Survey of Existing Techniques," *ACM Computer Surveys*, vol. 46, no. 4, 2013.
- [23] "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices," IEEE, Standard, 2008.
- [24] M. Liskov, R. L. Rivest, and D. Wagner, "Tweakable Block Ciphers," *Journal of Cryptology*, vol. 24, no. 3, 2011.
- [25] P. Maene and I. Verbauwhede, "Single-cycle Implementations of Block Ciphers," in *Lightweight Cryptography for Security and Privacy*, ser. LNCS, 2015.
- [26] "Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices," National Institute of Standards and Technology, Special Publication, 2010.
- [27] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. V. Herrewwege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.
- [28] R. Obermaisser, P. Peti, and F. Tagliabo, "An Integrated Architecture for Future Car Generations," *Real-Time Systems*, vol. 36, no. 1, 2007.
- [29] P. Oester, "Dirty COW (CVE-2016-5195)," MITRE, 2016.
- [30] J. Pabel, "Frozen Cache," <https://frozenspace.blogspot.com>, 2009.
- [31] T. P. Parker and S. Xu, "A method for safekeeping cryptographic keys from memory disclosure attacks," in *Proceedings of the 1st International Conference on Trusted Systems*, 2009.
- [32] P. Peterson, "Cryptkeeper: Improving Security with Encrypted RAM," in *Proceedings of the 10th Conference on Technologies for Homeland Security*, 2010.
- [33] N. Provos, "Encrypting Virtual Memory," in *Proceedings of the 9th USENIX Security Symposium*, 2000.
- [34] P. Simmons, "Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption," *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.
- [35] C. Tarnovsky, "Deconstructing a "Secure" Processor," *Black Hat DC*, 2010.

- [36] Tilo Müller and Felix Freiling and Adreas Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *Proceedings of the 20th USENIX Security Symposium*, 2011.
- [37] P. Williams and R. Boivie, "CPU Support for Secure Executables," in *Proceedings of the 4th Conference on Trust and Trustworthy Computing*, 2011.
- [38] G. Yang, B. Zhu, V. Suder, M. D. Aagaard, and G. Gong, "The Simeck Family of Lightweight Block Ciphers," *Cryptology ePrint Archive*, Report 2015/612, 2015.

Pieter Maene is a research assistant at the COSIC research group at KU Leuven. His research interests include trusted computing architectures, hardware-software co-design, and hardware implementations of cryptographic algorithms.

Johannes Götzfried is a post-doctoral researcher at the chair for IT Security Infrastructures at the Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg. His research interests include trusted computing, system security and physical security.

Tilo Müller is a post-doctoral researcher at the chair for IT Security Infrastructures at the Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg. His research interests include system security, mobile security and software protection.

Ruan de Clercq is a post-doctoral researcher at the COSIC research group at KU Leuven. His research interests include embedded security, computer security architectures, and applied cryptography.

Felix Freiling is professor of computer science at the Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg. His research interests cover theory and practice of dependable systems.

Ingrid Verbauwhede is a professor of electrical engineering at KU Leuven. Her main interest is in the design and the design methods for secure embedded circuits and systems. She is a fellow of IEEE.