Journal of Cryptographic Engineering manuscript No.
(will be inserted by the editor)

Practical Feasibility Evaluation and Improvement of a Pay-per-Use Licensing Scheme for Hardware IP Cores in Xilinx FPGAs

Jo Vliegen · Nele Mentens · Dirk Koch · Dries Schellekens · Ingrid Verbauwhede

Received: date / Accepted: date

Abstract In earlier published work, Maes et al. present a pay-per-use licensing scheme for hardware Intellectual Property (IP) cores. This scheme focuses on the use of IP cores on SRAM-based FPGAs and is mainly based on the partial reconfigurability property of this type of FPGA. Our work evaluates the practical feasibility of the scheme and the accompanying architecture. As already (partly) indicated by Maes et al., their solution introduces some security and usability issues. Therefore, we present improvements to the scheme and the architecture together with an additional method for decreasing the area overhead. The overall result is the first practical implementation of the pay-per-use licensing scheme occupying 841 slices on a Xilinx XC6S-LX45 FPGA. The small area overhead is mainly achieved by moving the storage of keys from slice flip-flops to configuration memory. Moreover, the implementation would not have been feasible with commercially available tools. We use an academic tool that allows nested partial reconfiguration and flexible IP core placement.

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007) and by the Flemish iMinds projects. In addition, this work is supported in part by the Flemish Government, FWO G.0550.12N, by the Hercules Foundation AKUL/11/19, and by the European Commission through the ICT program under FP7-ICT-2011-8.

J. Vliegen, N. Mentens, D. Schellekens, I. Verbauwhede
KULeuven, ESAT/COSIC & iMinds
Kasteelpark Arenberg 10 - bus 2452,
B-3001 Leuven-Heverlee, Belgium
E-mail: givenname.surname@esat.kuleuven.be
Tel: (+32) 11 180 921

D. Koch
The University of Manchester, School of Computer Science
Oxford Road, Manchester, M13 9PL, United Kingdom
E-mail: dirk.koch@manchester.ac.uk
Tel: (+44) 161 275 61 27

Keywords Hardware IP core licensing · FPGA · Dynamic partial reconfiguration · Key storage · Cryptography

1 Introduction and previous work

Hardware designs are made for a wide variety of applications. The devices for which these designs are tailored, are also getting more complex and more powerful. Additionally, the techniques available for developing hardware designs continue to improve and perfect. This makes the design of efficient hardware an increasingly complex task. Rather than starting each hardware design from scratch, reusing already designed components becomes common practice. With the complexity of individual components increasing, the market of designing and selling Intellectual Property (IP) cores has been introduced around the turn of the century. With reconfigurable hardware, e.g. a Field Programmable Gate Arrays (FPGAs), these IP cores can be implemented easily and dynamically. Moreover, their reconfigurable property allows IP cores to be software and/or hardware in contrast with the firmware updates of microprocessors which only target software updates.

With the pitfall of loosing IP, and therefore money, a number of solutions have been proposed. Simpson and Schaumont describe an offline authentication scheme for embedded software IP modules in FPGAs [16]. Besides this software-oriented approach, a number of hardware solutions were proposed [5][6][7][9]. In [6], a proof-of-concept implementation is presented to reconfigure the majority of the FPGA such that it has a design, containing a specific IP core. However, this does not provide a flexible way of obtaining and implementing one or more IP cores. Although none of the other solu-

tions ([5][7][9]) were implemented in practice, the work of Maes et al. [9] elaborates the most on practical issues. Moreover, their work offers the pay-per-use feature where the system developer pays a price for the IP core per device in which it is instantiated. The scheme of Maes et al. is based on the property of FPGAs to support partial and dynamic reconfiguration.

Although Maes et al. worked out their pay-per-use licensing scheme to a detailed level, it still leaves considerable practical aspects untouched. In this paper, we evaluate the practical feasibility of their scheme. As already partly indicated by the authors of [9], the proposed architecture is vulnerable for side-channel attacks. Additionally, commercially available tools do not allow nested and flexible placement of IP cores. We present an improved scheme and architecture that give a solution to the nesting and flexibility issues. In addition, we describe a novel technique to decrease the area overhead. In order to achieve a more secure, practically implementable, and smaller solution, we use a tool flow based on the academic tool: GoAhead [1]. The result is a working FPGA implementation with a small overhead in area. This is the first implementation of a pay-per-use licensing scheme for hardware IP cores.

This paper first describes the Xilinx SRAM-based FPGAs in Sect.2 to introduce or refresh the readers comprehension of FPGAs. Subsequently, this work gives an overview of the original licensing scheme and the issues it introduces in Sect. 3. The implementation of the scheme, including the solutions to the problems and a modified version of the scheme, are thoroughly explained in Sect. 4. Finally, Sect. 5 gives the results of the prototype implementation and Sect. 6 concludes this work.

2 The Xilinx SRAM-based FPGA in a nutshell

2.1 The FPGA chip

An FPGA can be seen as dual-layer chip consisting of the reconfigurable fabric and the configuration memory. Depending on the technology used for storing the configuration bits, the three most common technologies for FPGAs are: SRAM-based, Flash-based and antifuse-based FPGAs. For more information and a comparison between these technologies we refer to [3]. The work of Maes et al. [9], hence this work as well, focuses on Xilinx SRAM-based FPGAs. The reconfigurable fabric mainly consists of Configurable Logic Blocks (CLBs) and routing components, where the CLBs contain Look-Up Tables (LUTs), multiplexers and flip-flops. The output of a LUT can be configured to any function of the LUT's inputs. Furthermore, next to flip-flops, there are two types of storage primitives: Block RAMs and distributed memory. Block RAM (BRAM) is a dedicated memory component on the silicon of the FPGA. Distributed RAM consists of a combination of many small RAM blocks in the LUTs. while the flip-flops provide storage. The routing components in the reconfigurable fabric are use to achieve the desired routing.

2.2 The FPGA configuration

To configure an FPGA with a certain design a 'bitstream' is to be created first. The different steps in generatring a bitstream for a Spartan-6 FPGA are done through the ISE design software of Xilinx. Their current tool, Vivado, does not provide support for Spartan-6 devices and is hence not used.

The first step is to describe the design in a Hardware Description Language (HDL), like VHDL or Verilog, or through schematic entry. The design has to go through the synthesis tool to be transformed into a netlist *.ngc*. This netlist then gets combined with any other cores in the Build step, which results in a *.ngd* file. This *.ngd* file contains a logical description in terms of logic elements like AND gates, flip-flops, and similar gates. Thereafter, this *.ngd* file gets mapped on the hardware primitives, which are the building blocks of the reconfigurable fabric of the targeted FPGA, resulting in a *.ncd* file. The subsequent step is to place and route the mapped design. Placing is determining where each primitive of the mapping phase is placed in the chip, while routing tries to make every required connection through the routing lines on the chip. These steps result in in a routed *.ncd* file which, finally, can be used to generate a bitstream for the FPGA.

By loading the configuration memory with a bitstream, the behavior of the reconfigurable fabric is determined. Commonly an FPGA is configured in its entirety, but partial reconfiguration is a technique that allows the reconfiguration of a certain partition of an FPGA. A bitstream that contains the configuration of a partition of the FPGA is referred to as a partial bitstream. The partition that stays unchanged is referred to as the 'static partition' while the one or more other partitions are referred to as 'reconfigurable partitions'. If the operation of the static partition continues uninterruptedly during the reconfiguration of a reconfigurable partition, this is referred to as 'dynamic' reconfiguration. The possibility exists to use encrypted bitstreams to configure FPGAs. An on-chip decryption core is present to perform bitstream decryption, but up to the latest Xilinx series [19] this feature is not available for partial bitstreams. Moreover, this decryption

core is not available for any purpose other than bitstream decryption.

To access the configuration memory from the reconfigurable fabric only the Internal Configuration Access Port (ICAP) primitive provides an interface. Reading back as well as writing the configuration memory can be performed through the ICAP. Note that the ICAP even allows read-back if the FPGA was configured with an encrypted bitstream. The storage of the encryption key and the Initial Value can be done in battery-backed RAM (BBRAM) or in eFUSE. The former is preferred over the latter because the eFUSE can be erased [20].

Table 1 summarizes which recent Xilinx FPGA families offer certain features. This table learns that authentication, which guarantees the FPGA configuration is not tampered with, is only available on the most recent devices. All device families mentioned in Table 1 can be partially reconfigured using the vendor tools, except the Spartan 6 family.

Fig. 1 Interactions between the entities in [9], where FV, MA, CV and SD stand for FPGA vendor, metering authority, IP core vendor and system developer, respectively.

3 The original licensing scheme and architecture

This section summarizes the relevant parts of the licensing scheme presented in [9], discusses the originally proposed architecture, and points out the possible security and usability issues.

3.1 The original licensing xcheme

There are four entities that participate in the scheme: the FPGA vendor (FV), the metering authority (MA), the IP core vendor (CV) and the system developer (SD). The interactions between the different entities are shown in Fig. 1. The scheme consists of an initialization phase and a design phase. Both are described below in detail.

3.1.1 Initialization phase

When the FPGA vendor produces an FPGA F_i^*, the device can be sent to a metering authority for registration (transaction 1 in Fig. 1). The metering authority generates a random device key k_i^F and metering key k_i^M and stores these keys in a database together with the ID of the FPGA: ID(F_i). Further, the metering authority stores the device key k_i^F in the secure non-volatile memory of the FPGA. The metering authority will additionally perform an encryption of a bitstream, using the key k_i^F. This bitstream is called the metering bitstream and it contains the metering design M which includes a register that stores the metering key k_i^M. The metering bitstream $B_i(M, k_i^M)$ is computed as $B_i(M, k_i^M) = Enc[b(M, k_i^M)]_{k_i^F}$, where $Enc[x]_y$ stands for a symmetric encryption of x using key y; and $b(x, y)$ stands for a plaintext bitstream that implements hardware components x and y. The uninitialized FPGA F_i^* is now transformed into a registered FPGA F_i, which is handed back to the FPGA vendor, together with $B_i(M, k_i^M)$ (transaction 2 in Fig. 1).

When a system developer buys an FPGA, possibly as component on a development board, which is enabled to use the licensing scheme, the FPGA F_i is delivered, together with $B_i(M, k_i^M)$ (transaction 3 in Fig. 1).

IP core providers also need to register their cores through the metering authority. They have to register every offered IP core by providing the metering authority with an ID of the IP core, ID(IP_j), together with a key, k_j^{IP} (transaction 4 in Fig. 1). Both the metering authority and the IP core provider store ID(IP_j) and k_j^{IP} in a database.

3.1.2 Design phase

When a system developer wants to obtain and use an IP core in a specific FPGA, the following interactions occur. The system developer requests the IP core identified by ID(IP_j) from the IP core vendor (transaction 5 in Fig. 1). This results in the IP core vendor sending the bitstream of the IP core to the system developer, encrypted with the key k_j^{IP}, i.e. $B(IP_j) = Enc[b(IP_j)]_{k_j^{IP}}$ (transaction 6 in Fig. 1). After transactions 5 and 6 the system developer cannot use the IP core because the decryption key is not yet available.

Table 1 Feature overview of recent Xilinx FPGAs

feature	Virtex-4 & Virtex-5	Spartan-6	Virtex-6 & 7-series
year of commercial release	2004 & 2006	2009	2010 & 2011
encrypted full bitstream	AES256	AES256[1]	AES256
encrypted partial bitstream	no	no	no
key storage	BBRAM	BBRAM / eFUSE	BBRAM / eFUSE
ICAP readback with encrypted bitstream	yes	yes	yes
ICAP support with encrypted bitstream	yes[2]	yes	yes
bitstream authentication	no	no	HMAC

[1] not in LX, SX and FX12
[2] for Virtex-4: only in LX75(T), SLX100(T), LX150(T)

When the system developer wants to integrate the core in his design, the IDs of the FPGA (ID(F_i)) and the IP core (ID(IP_j)) are sent to the metering authority (transaction 7 in Fig. 1). The metering authority generates a license $K_{i,j}^{IP}$, with $K_{i,j}^{IP} = Enc[k_j^{IP}]_{k_i^M}$, which is sent to the system developer (transaction 8 in Fig. 1).

The system developer then configures the FPGA F_i with bitstream $B_i(M, k_i^M)$, which allows the key k_i^M to get on the FPGA without another entity being able to read that key. Using k_i^M, the system decrypts the license ($K_{i,j}^{IP}$) to obtain k_j^{IP}, which is used to decrypt the encrypted IP core $B(IP_j)$ on the FPGA, in order to configure the IP core in the system developer's design through partial reconfiguration.

3.2 The original architecture

During the design phase, after all transactions in Fig. 1 have been performed, the architecture implemented on the FPGA is altered a few times. This is depicted in Fig. 2, where the white area represents the reconfigurable resources of the FPGA, while the gray area holds additional dedicated components available on the die of the FPGA. It is pointed out that the ICAP is permanently present in the reconfigurable part. In every step of Fig. 2 incoming data is used to update registers or to reconfigure the FPGA. The targeted components of the incoming data are indicated by the bold arrows.

The top image visualizes how $B_i(M, k_i^M)$ configures the metering design and the metering key in the FPGA through a full configuration. This configuration uses the on-chip bitstream decryption core which is available on the FPGA, using the key k_i^F, which is stored in Non-Volatile Memory (NVM). The partial reconfigurations of the FPGA that follow in a later phase, do not alter the metering design. Therefore, we refer to the part of the FPGA that holds the metering design as the 'static part', while the rest of the FPGA is reserved for the system developer's design together with the IP cores.

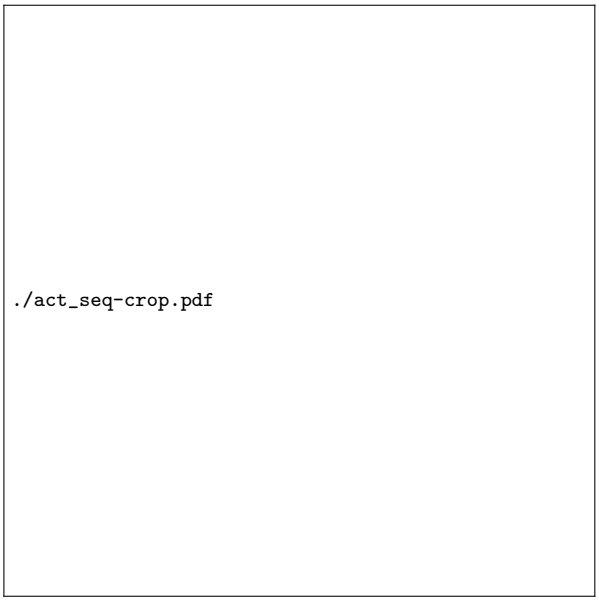

Fig. 2 The evolution of the FPGA architecture in [9] during the design phase. The Non-Volatile Memory (NVM) stores the device key for the on-chip decryption core.

The second image in Fig. 2, shows the second step in which the metering design, consisting of a custom AES decryption core and two registers (of which one contains the metering key k_i^M), is already present in the static part of the FPGA. The license $K_{i,j}^{IP}$ is decrypted by the custom decryption core on the reconfigurable fabric of the FPGA, using key k_i^M, which results in the initialization of the already implemented register for k_j^{IP}. It should be noted that this step performs no configuration, but the initialization of a key in a register.

The bottom image shows the incoming, encrypted bitstream $B(IP_j)$, containing the obtained IP core. This bitstream is decrypted on the custom decryption core, using key k_j^{IP}, and is routed to the ICAP. This results in a partial reconfiguration of the FPGA to implement the design of the IP core.

3.3 Issues in implementing the licensing scheme

As already partly indicated by Maes et al., the implementation of the licensing scheme leads to a number of practical issues. In this subsection we explain the two most important issues.

3.3.1 Side-channel security of the embedded AES core

The original scheme uses the dedicated AES bitstream decryptor in the die of the FPGA for the full configuration of the initial system. In [9], Maes et al. already mention the work of Moradi et al., that presents a side-channel attack on the AES core in CBC mode [11]. The attack reveals the key that is used to decrypt encrypted bitstreams. In the licensing scheme, this means that k_i^F can be revealed during the decryption of $B_i(M, k_i^M)$. With k_i^F, the attacker can decrypt $B_i(M, k_i^M)$ outside the FPGA to obtain k_i^M. This would reveal k_j^{IP} from the license, which can finally be used to decrypt the partial bitstream containing the obtained IP core. This would directly lead to a loss of revenue for the CV. Moreover, an in-depth study of this bitstream can reveal implementation details of the IP core. Note that Maes et al. use a custom AES decryption core for the partial bitstreams, since decryption of a partial bitstream was not supported by the on-chip core.

3.3.2 Nested and flexible integration of IP cores

As shown in Fig. 2, the metering design resides in the static part of the FPGA. The system developer's design is a partial module that covers the remaining part of the FPGA. Since the obtained IP core will be placed inside the system designers' hardware, the IP core needs to be nested as a partial module inside the system developer's design. This requires the ability of performing nested partial reconfiguration, which is not supported by the commercial tools. Moreover, flexible placement of IP cores is not allowed either, which means that an IP core can only be implemented at a predetermined location on the FPGA. Maes et al. support this by having the IP core vendor tailor the obtained IP core to the system developer's needs. However, this slows down the design time, increases the IP core price and results in a non-flexible, non-scalable solution.

4 Improvement and implementation

This section explains how the issues in Sect. 3 can be overcome. In addition, we propose a method for decreasing the area overhead of the architecture. Finally, the novel implementation and tool flow are discussed.

4.1 Overcoming the issues

4.1.1 Side-channel security of the embedded AES core

The original architecture in [9] uses the on-chip AES decryption core for full configuration and uses a custom AES core in the reconfigurable logic of the FPGA in order to decrypt licenses and partial bitstreams, as explained in Sect. 3.2. To solve the side-channel security issues of the on-chip AES core, there are two solutions. Either the on-chip AES core needs to be replaced by a side-channel secure core or a work-around needs to be found based on the existing FPGA technology. Because the former lies with the FPGA vendor and because we want to offer a solution for existing FPGAs, we propose not to use the on-chip AES core. Instead we only use the custom AES core in the reconfigurable logic for all sensitive decryptions. This core needs to be configured in the FPGA which is done with an encrypted bitstream, B_{init}, that also contains a connection to the ICAP and a storage unit for the metering key k_i^M and the IP core key k_j^{IP}. This static bitstream B_{init} is to be generated by the MA and handed back to the FV when the FPGA is registered with the MA (transaction 2 in Fig. 1). Next, the encrypted bitstream $B_i(M, k_i^M)$ is sent to the FPGA and gets decrypted by the custom AES core. According to the original scheme, $B_i(M, k_i^M)$ contains a register that holds k_i^M, an AES decryption component, and a connection to the ICAP. Since the AES decryption component and the connection to the ICAP were already implemented by B_{init}, the only new component in $B_i(M, k_i^M)$ is the metering key k_i^M. The encrypted bitstream $B_i(M, k_i^M)$ is decrypted with key k_i^F, which is available inside the FPGA after the initialization phase.

Note that no precautions have been taken to prevent an attacker from altering the encrypted bitstream B_{init} through the techniques of Moradi et al. [11]. By doing so, an attacker can modify the decrypted bitstream by making connections between the key storage and the outside world to simply eavesdrop every key in a later phase. Some tools exist to reverse engineer a bitstream [2,14], but these tools often focus on a specific device family. The results of these tools look promissing but are not yet optimal and flexible enough to take bitstream reverse enginering for granted. Therefore, we do not take precautions to prevent this attack in this work. The evolution of the FPGA architecture using this novel approach, is vizualised in Fig. 3.

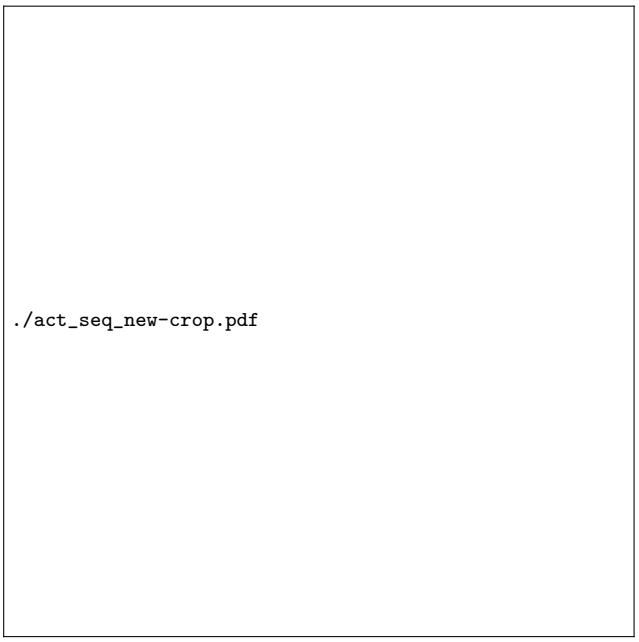

Fig. 3 The evolution of the FPGA architecture in the improved scheme, during the design phase. The KSwU stands for the key switching unit and KStU for the key storage unit.

4.1.2 Nested and flexible integration of IP cores

Commercially available tools for partial reconfiguration do not allow partial modules to be nested. Koch et al. developed an academic tool called ReCoBus-Builder [8] that evolved into the GoAhead tool [1], which allows nested partial reconfiguration. This tool heavily relies on the Xilinx Design Language (XDL) [18] to achieve its unique features. Therefore, the use of the tool binds the scheme to Xilinx FPGAs. Further, GoAhead also allows the flexible placement of reconfigurable modules in comparison to a predetermined location of modules using commercial tools. This makes our solution more practical, cheaper and less devious for both the IP core vendor and the system developer. An additional benefit of using GoAhead is that partial reconfiguration on the Spartan 6 FPGA family becomes possible, in contrast with commercial tools' as mentioned in Sect. 2.2.

4.2 Additional improvements to the architecture

In order to decrease the resource occupation of the static partition that handles the licensing scheme, an alternative method for key storage is proposed. In the original scheme, only the metering key (k_i^M) and the IP core key (k_j^{IP}) are stored in the flip-flops of the FPGA, since these are the keys used for the custom decryption core. The device key (k_i^F) is stored in a memory element that drives the built-in AES decryptor. As explained in Sect. 4.1.1 our solution uses the custom AES core for all decryptions, which means three keys have to be stored. Sect. 4.2.1 describes how traditional storage of the keys can be done, while 4.2.2 explains how our novel approach allows the storage of three keys using only half the area compared to the storage of two keys using the traditional method. Moreover, in the traditional setting, two additional reconfigurable partitions are needed for the storage of the keys, while our approach does not need any further partitioning in the static partition. This leads to additional savings in area and timing, since the interconnection of partial modules with the rest of the FPGA introduces an overhead in area and timing.

4.2.1 The traditional way: key storage in slice flip-flops

Upon initial configuration, the registers for storing k_i^M and k_j^{IP} are empty. During two partial reconfiguration steps the partial bitstream $B_i(M, k_i^M)$ and the license $K_{i,j}^{IP}$ store k_i^M and k_j^{IP} in these registers. Assuming AES-128 is used for encryption, each register holds a 128-bit key. Given that one slice contains four flip-flops (for a Xilinx Virtex-5 FPGA), the two registers require 64 slices.

4.2.2 The improved way: key storage in configuration memory

In order to reduce the overhead used by the key registers, the storage of the keys is moved from the slice flip-flops to the "configuration memory" of the FPGA by using the configuration bits of the LUTs.

Fig. 4 shows the architecture of a LUT6 which is available in a Spartan-6 FPGA. This LUT6 can be configured as a single 6-to-1 LUT or as two 5-to-1 LUTs. Any function with n inputs can be configured in a LUT, resulting in 2^n possible functions. Among these functions are two functions that map any given input to '0' or to '1', respectively. Configuring both 5-to-1 LUTs through the SRAM memory to one of these two functions, turns a single LUT6 in a 2-bit ROM. The value of the ROM is stored in the configuration memory. Since the AES core needs 128 key bits in parallel, we need 64 LUTs or 16 slices for the storage of one key.

As explained earlier in this section, there are three keys to be stored, namely k_i^F, k_i^M and k_j^{IP}. By altering the truth table of the LUT-as-2-bit-ROM as shown in Table 2, the single LUT can hold 2 bits of all three keys. This means that the 16 occupied slices can store all three keys. From a functional point of view, the achieved behavior could be represented as shown in Fig. 5. The configuration bits of the LUT determine

Fig. 4 Architecture of a LUT6 in a Spartan-6 FPGA

the value of the three 2-bit ROMs and therefore determine 2 bits of each key.

Table 2 Truth table of the LUTs in the key storage unit

A1	A2	A3	A4	A5	A6	O5	O6
0	X	X	X	X	X	$k_i^F[n]$	$k_i^F[n+1]$
1	0	X	X	X	X	$k_i^M[n]$	$k_i^M[n+1]$
1	1	X	X	X	X	$k_j^{IP}[n]$	$k_j^{IP}[n+1]$

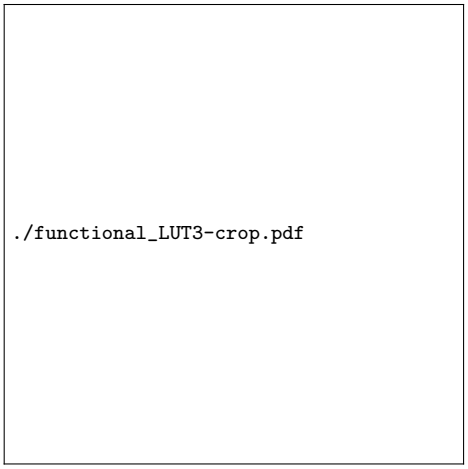

Fig. 5 A functional representation of the LUT configuration in the key storage unit

The group of 16 slices which stores the three keys is referred to as the "key storage unit". To prevent the design tool from optimizing this construction away, the key storage unit is implemented as a hard macro [17]. When instantiating the key storage unit, all key bits are initialized to '0'. An update of the keys is achieved through the reconfiguration of the 16 slices of the key storage unit.

In order to be able to switch between the three keys for the AES decryption core, the selection inputs $A1$ and $A2$ of the multiplexers in Fig. 5 have to be altered.

To be able to switch between keys through reconfiguration, the same approach of using a LUT as a 2-bit ROM can be used again. The truth table of this LUT is shown in Table 3 and its functional representation is shown in Fig. 6. The slice in which this LUT resides is referred to as the "key switching unit".

Table 3 Truth table of the LUT in the key switching unit

A1	A2	A3	A4	A5	A6	O5	O6
X	X	X	X	X	X	A1	A2

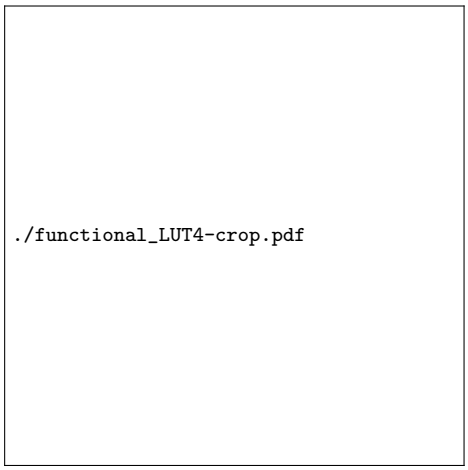

Fig. 6 A functional representation of the LUT configuration in the key switching unit

Combining the key storage and key switching units would functionaly look as depicted in Fig. 7.

This novel technique of storing the different keys also alters the protocol of changing between keys. In the original licensing scheme of Maes et al. [9], receiving a new key occurs in a single update. This is possible because both k_i^M and k_j^{IP} are stored in two separate registers. Using the proposed technique requires two updates for storing a new key: 1) storing the key in the key storage unit and 2) updating the selection bits in the key switching unit.

In the first step, the incoming partial bitstream (containing the new key k_{new}) is decrypted with the currently used key k_{active} and forwarded to the ICAP to update the key storage unit. This has no effect on the value of k_{active}, used for decrypting the incoming bitstream, because the key switching unit is not yet updated. This first step is depicted in Fig. 8.

It is only upon receiving an update on the key switching unit, that k_{new} gets used in the AES core. Naively using the same method as updating the key storage unit

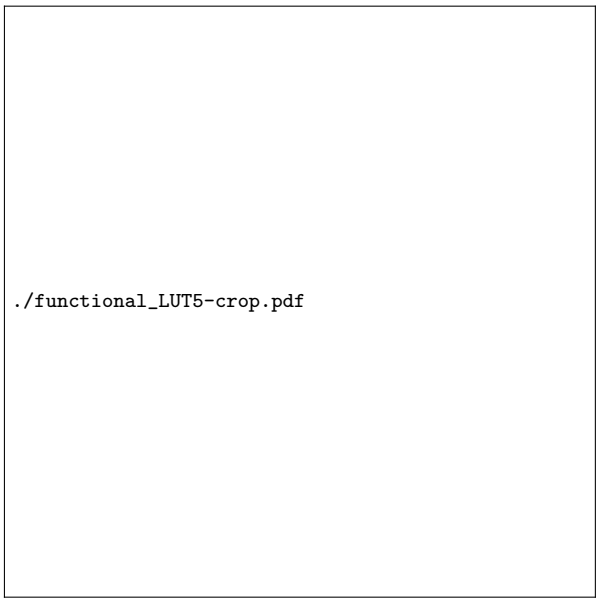

Fig. 7 A functional representation of the LUT configuration in the key switching unit

Fig. 8 The first step: Updating the key storage unit

would result in switching the key **during** decryption of the incoming partial bitstream. This corrupts the key among other, as is depicted in the top half of Fig. 9. To prevent this, the partial bitstream to update the key switching unit has to be received, decrypted and temporarily stored **before** routing it to the ICAP. Achieving this can be done by adding a FIFO that stores the decrypted partial bitstream. This is depicted in the bottom half of Fig. 9. Because the partial bitstream for updating the key switching unit (1 slice) is small, the FIFO is small as well.

Fig. 9 The second step: Updating the key switching unit. The top half depicts the naive approach and the bottom half depicts the correct approach.

4.3 Novel architecture and tool flow

4.3.1 Side-channel robust decryption core

As mentioned in Sect. 4.1.1, a side-channel robust AES decryption core is required in the novel architecture. We follow the approach of Moradi et al. [12] in which the authors describe a twofold contribution. First there is the very compact implementation of AES and secondly have they implemented the threshold countermeasure, presented by Nikova et al. [13]. With this countermeasure, Nikova et al. achieve provable security against differential power analysis and higher-order differential power attacks. The discussion of these attacks and countermeasures fall out of the scope of this work, but the interested reader should consult [10], [4], [13], and [12].

The very small overhead of this AES128 implementation is beneficial for our work as well. The smaller the cost of the static partition, the larger the amount of reconfigurable resources for the single reconfigurable partition. The countermeasure presented in [13] protects implementations against side-channel attacks based on logic glitches.

4.3.2 Novel architecture

The architecture residing in the static partition, handling the licensing scheme, is depicted in Fig. 10. The key switching unit sends the selection signals $A1$ and $A2$ to the key storage unit in order to determine which

of the three keys (k_i^F or k_i^M or k_j^{IP}) is used in the AES core. The output of the custom AES core is forwarded through a bit swapper to the ICAP. The bit swapper makes sure the bits of the decrypted bitstreams are routed in the correct order to the ICAP. The need for the FIFO is explained in Sect. 4.2.2. In our proof-of-concept implementation, encrypted bitstreams are sent to the AES core over a UART [15]. In an industrial implementation the communication interface needs to be replaced by an interface with a higher throughput that is accessible through the Internet.

Fig. 10 The architecture residing in the static partition, handling the improved licensing scheme.

Fig. 11 shows a floorplan for the nested partial reconfiguration of the FPGA. The largest block (Static) consists of all reconfigurable resources on the FPGA. The two rightmost components are the key switching unit (KSwU) and the key storage unit (KStU). These units reside in the static partition. The darker colored component on the left is a reconfigurable partition (Design), which holds the design of the system developer. Up until this point there is a single static partition with a single reconfigurable partition. The IP core which is acquired for the design of the system developer is a reconfigurable partition as well, encapsulated in another reconfigurable partition (Design). The first level of the reconfigurable partitions contains the Design and the second level contains the obtained IP cores.

4.3.3 Novel tool flow

Xilinx offers the PlanAhead tool for partial reconfiguration. The tool generates a partial bitstream for every possible configuration of the static and reconfigurable partitions. For example, a design with two reconfigurable partitions for which the first reconfigurable partition has two available designs and the second reconfigurable partition has three available designs, ends up with six full bitstreams and six times two partial bitstreams. Additionally to this large processing overhead, generating bitstreams for nested reconfigurable partitions is not feasible in PlanAhead.

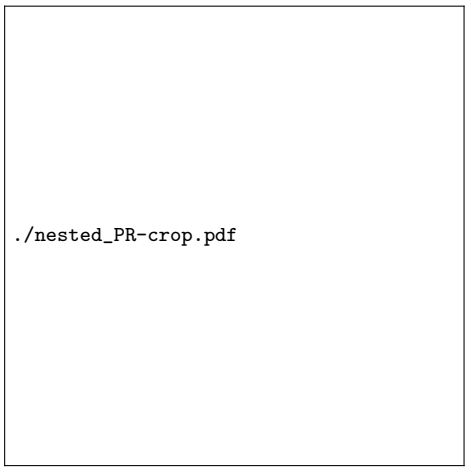

Fig. 11 Graphical representation of the nesting levels, where the Static partition occupies one level; the Design partition, the key storage unit (KStU) and key switching unit (KSwU) form the first level and the IP core(s) form the second nesting level.

In order to reduce the processing overhead and to allow nested reconfigurable partitions, we use the GoAhead tool [1]. This tool originated from the ReCoBus Builder [8], developed by Koch and Beckhoff.

In order to reconfigure the key storage unit and the key switching unit which both reside in the static partition through partial reconfiguration, the key bits and the key selection bits need to be inserted through partial bitstreams. These partial bitstreams are generated by making differential bitstreams with respect to the initial bitstream, b_{init}, and to $b(M_i, k_i^M)$, respectively for k_i^M and k_j^{IP}. To use the technique of making differential bitstreams, first a full bitstream with uninitialized hard macros for the key storage and key switching unit is generated. Then, the second full bitstream, containing the desired modification to one of both units, is generated. Both full bitstreams have to be restricted in the area which can be used. To achieve this, GoAhead applies restrictions on the placement and routing tools to force the exclusive usage of primitives in a certain area. The router tool has to generate routing in this same area, which is achieved by blocking out all connections outside the restricted area.

The initial full bitstream is then generated for the first placed-and-routed bitstream. Instead of performing the bitstream generation step on the initialized configuration like in the traditional tool flow, this step is to be slightly altered to make a differential bitstream with respect to the initial full bitstream. This results in a partial bitstream for the first level of the reconfigurable parition which only updates the frames which differ in the second full bitstream from the first full bitstream. For more details and a more in-depth explanation about generating the full and partial bitstreams using GoAhead, we refer to the documentation [1].

5 Results

The architecture has been implemented on a Xilinx XC6S-LX45 FPGA. The FPGA resources occupied for the implementation of the licensing scheme, are shown in Table 4. The table illustrates that the majority of resources is dedicated to the AES core. Because the system developer needs as much free space as possible for the implementation of his/her own design in combination with licensed IP cores, the number of remaining resources available on the FPGA is important. Therefore, the table shows the total number of relevant resources for the FPGA we used for our proof-of-concept implementation, but also for the smallest and the largest member of the XC5V FPGA family. We can conclude that the relative overhead in area is acceptably small, except for the smallest FPGA on which the static partition not even fits.

Table 4 FPGA resources occupied by the licensing scheme

	AES	static partition	Relative usage of the static partition in a XC6S LX 4	45	150T
Slice FF	2486	3019	63 %	6 %	2 %
Slice LUTs	1729	2636	110 %	10 %	3 %
Occ. Slices	646	841	140 %	12 %	4 %
BRAM8	0	4	13 %	2 %	1 %

Next to the overhead in area, we also report on the execution speed of the licensing scheme. A full bitstream for the XC6X LX45 has a size of 1'484'785 bytes. The partial bitstream to update a key in the key storage unit, including padding for communication and encryption, is 2'232 bytes. The partial bitstream to alter the output of the key switching unit is 1'432 bytes. In our proof-of-concept implementation, a UART interface with a baud rate of 115'200 Bd is used. This results in a duration of 193.750 ms and 124.306 ms for the communication of the two partial bitstreams, respectively. For an update of the key in the key storage unit, 140 AES decryptions have to be performed, which takes 0.462 ms (at a speed of 3.3 μs per decryption of a 128-bit block). For an update of the key switching unit, 90 AES decryptions have to be performed, which takes 0.297 ms. In total a key update takes 194.212 ms (193.750 ms + 0.462 ms) and a key switch takes 124.603 ms (124.306 ms + 0.297 ms). In our proof-of-concept implementation, the communication speed forms the bottleneck. It is clear that a communication channel with a higher bandwidth is necessary for the practical enrollment of the system. To place the timing cost of the presented solution in perspective, we measured the duration of a single, full configuration of the XC6X LX45 which takes 7.94 seconds. Assuming an IP core occupies 12.5% of the device, Table 5 illustrates the absolute and relative impact on the duration to reach to a working system, with respect to a single full FPGA configuration.

Table 5 Absolute and relative timing with respect to a single full FPGA configuration, for a system with 1, 3 or 5 IP cores.

Operation	duration [ms]	Quantity for 1 IP cores	3 IP cores	5 IP cores
B_{init}	7940.000	1	1	1
KStU update	194.212	2	4	6
KSwU update	124.603	2	2	2
IP core	993.000	1	3	5
absolute duration		9.57	11.95	14.32
relative duration		1.21	1.51	1.81

KStU: key storage unit
KSwU: key switching unit

As could be expected, from Table 5 it is clear that there is an increasing overhead with the number of IP cores. However, this additional cost in timing is still acceptable and could, in an extreme case, double the duration. Obviously, this relates the number and the size of the IP cores.

6 Conclusion and future work

This work describes a practical evaluation of the licensing scheme presented by Maes et al. in [9]. We tackle a number of feasibility and usability issues that occur in the licensing scheme and the accompanying architecture. Further, we present additional improvements that decrease the area overhead of the implementation, where the novelty consists of moving the key storage

from the slice flip-flops to the configuration memory. The novel architecture was implemented on a Xilinx FPGA, with a very small area overhead. The tool flow is based on the academic GoAhead tool, that allows nested partial reconfiguration and flexible IP core placement.

Future work consists of implementing a state-of-the-art mode of operation around the AES cipher. Such a mode of operation can, additionally, provide data authentication and data integrity to the conversed bitstream. This, respectively, encompasses that the FPGA is ensured the data comes from a trusted entity and is not tampered with during the conversation. The second issue is the storage of the 'root' key in non-volatile memory. In current versions, this is based on fuses, in future versions, newer technologies such as PUF-based key storage could be used. Further, a communication channel with a higher bandwidth and Internet connectivity should be included to make the solution usable in a real-life setting.

References

1. C. Beckhoff, D. Koch, and J. Tørresen. Go Ahead: A Partial Reconfiguration Framework. In *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2012.
2. F. Benz, A. Seffrin, and S. A. Huss. Bil: A tool-chain for bitstream reverse-engineering. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 735–738. IEEE, 2012.
3. A. Braeken, S. Kubera, F. Trouillez, A. Touhafi, J. Vliegen, and N. Mentens. Secure FPGA technologies and techniques. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 560–563. IEEE, 2009.
4. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *Advances in Cryptology — CRYPTO' 99*, pages 398–412. Springer Berlin Heidelberg, 1999.
5. S. Drimer, T. Güneysu, M. G. Kuhn, and C. Paar. Protecting multiple cores in a single FPGA design, 2008.
6. T. Feller, S. Malipatlolla, D. Meister, and S. A. Huss. TinyTPM: A Lightweight Module aimed to IP Protection and Trusted Embedded Platforms. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 6–11. IEEE Computer Society, 2011.
7. T. Guneysu, B. Möller, and C. Paar. Dynamic Intellectual Property Protection for Reconfigurable Devices. In *International Conference on Field-Programmable Technology (ICFPT)*, pages 169–176. IEEE, 2007.
8. D. Koch, C. Beckhoff, and J. Teich. ReCoBus-Builder; A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAs. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 119–124. IEEE, 2008.
9. R. Maes, D. Schellekens, and I. Verbauwhede. A Pay-per-Use Licensing Scheme for Hardware IP Cores in Recent SRAM based FPGAs. *IEEE Transactions on Information Forensics and Security*, 7(1):98–108, 2012.
10. S. Mangard, T. Popp, and B. M. Gammel. Side-Channel Leakage of Masked CMOS Gates. In A. Menezes, editor, *CT-RSA*, pages 351–365. Springer, 2005.
11. A. Moradi, M. Kasper, and C. Paar. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures - An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA)*, pages 1–18. Springer, 2012.
12. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: a very compact and a threshold implementation of AES. In *Advances in Cryptology–EUROCRYPT 2011*, pages 69–88. Springer, 2011.
13. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of non-linear functions in the presence of glitches. In *Information Security and Cryptology–ICISC 2008*, pages 218–234. Springer, 2009.
14. J.-B. Note and E. Rannaud. From the Bitstream to the Netlist. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, page 264–264. ACM, 2008.
15. A. Osborne. *An Introduction to Microcomputers: Basic concepts*. McGraw-Hill, 2nd edition, 1980.
16. E. Simpson and P. Schaumont. Offline Hardware/Software Authentication for Reconfigurable Platforms. In *Proceedings of the 8th International Workshop Cryptographic Hardware and Embedded Systems (CHES)*, pages 311–323. Springer, 2006.
17. Xilinx. Xilinx Implementation Strategies using FPGA Editor, 2010.
18. Xilinx. Xilinx Partial Reconfiguration User Guide (UG702), 2010.
19. Xilinx. 7 Series FPGAs Configuration (UG470), 2013.
20. Xilinx. Spartan-6 FPGA Configuration User Guide (UG380), 2013.