ReCoFused partial reconfiguration for secure moving-target countermeasures on FPGAs

Buse Ustaoglu¹ · Kenneth Schmitz¹ · Daniel Große^{1,2} · Rolf Drechsler^{1,2}

Received: 19 January 2020 / Accepted: 4 June 2020 / Published online: 10 July 2020
© Springer Nature Switzerland AG 2020

Abstract

Partial reconfiguration is a versatile technique to modify the functionality of field programmable gate arrays (FPGAs) at run time. When performing partial reconfiguration a dedicated intellectual property (IP) component of the FPGA vendor, i.e., the partial reconfiguration controller (PRC), among a wide range of IP components has to be used. While ensuring the functional safety of FPGA designs is well understood, ensuring hardware security still remains challenging. This applies in particular to reconfiguration-based countermeasures which are intensively used to create a moving target for an attacker. Reconfiguration-based countermeasures against side-channel attacks or differential power analysis (DPA) attacks were implemented. However, from the system security perspective, the above mentioned PRC is a critical component as was noticed by many papers before. In this work, we extend a previously proposed safety mechanism which creates a container around an IP, to encapsulate and thereby to protect and observe the PRC of a FPGA. The proposed encapsulation scheme results in an architecture comprising so-called ReCoFuses (RCFs), each capturing a specific protective goal which have to be fulfilled at any time during PRC operation. The terminology follows the classical electric installation including a *fuse box*. In our scheme we employ formal verification to guarantee the correctness in detecting a security violation. Only after successful verification, the RCFs are integrated into the *ReCoFuse Container*. Experimental results demonstrate the advantage of our approach by preventing attacks on the PRC of a system secured by partial reconfiguration.

Keywords Reconfiguration · Partial reconfiguration controller · Verification · Moving target

1 Introduction

Substantial progress for both, application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs) has been achieved over the last decade. In particular, the programmable nature of FPGAs allows for great flexibility, and the powerful feature of partial reconfiguration pays off in many application fields today [19]. Practical examples include increasing fault tolerance [10], power-aware reconfiguration [25, 27], and area reduction by time division multiplexing [36].

Although the realization of partial reconfiguration varies depending on the FPGA model, it commonly relies on vendor specific proprietary library cells and intellectual property (IP). Due to the black box characteristic of these IP blocks, the internal operation (i.e., source code) cannot be *examined, tested or verified* by the user. As a consequence, integrating these components in a system requires *trust* in the test and verification methodologies of the respective IP vendor and—in worst case—jeopardizes the system's stability. Hence, several approaches to overcome this problem were proposed. Unfortunately, many

This article is an extended version of a conference paper appeared at *Applied Reconfigurable Computing* (ARC) 2019 [31].

✉ Buse Ustaoglu, buse.ustaoglu@dfki.de; Kenneth Schmitz, kenneth.schmitz@dfki.de; Daniel Große, grosse@uni-bremen.de; Rolf Drechsler, drechsler@uni-bremen.de | ¹Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany. ²Institute of Computer Architecture, University of Bremen, 28359 Bremen, Germany.

SN Applied Sciences (2020) 2:1363 | <https://doi.org/10.1007/s42452-020-3003-x>

of these approaches require at least some knowledge of the design sources, which is impractical for the aforementioned scenario. For this reason, the encapsulation of an IP component has been thoroughly investigated in the past. The behavior of the encapsulated component is then monitored, controlled or even fixed by the surrounding logic at runtime. For example, in [4] a “shield” is synthesized which continuously monitors inputs and output of the design and eventually corrects its erroneous outputs. A more general approach has been proposed in [6, 8]. The paper presents the notion of a “container”, in which the IP component is instantiated. The concept was applied in order to monitor and fix bus protocol glitches with automatic synthesis of correction logic from a property specification language. This way the container protects both, the IP and the surrounding system, respectively. A similar principle was applied on a hardware level by implementing an instruction replacement scheme for a modern RISC-V processor IP to circumvent errata and design flaws [30]. In [14] a similar technique has been proposed. Hardware sandboxes are employed for secure integration of non-trusted IPs in modern *System-On-Chips* (SoCs). Only permissible interactions between the IP and the rest of the system are allowed by exposing the IP interface to isolated virtual resources and checking IP signals’ correctness at run time.

Coming back to partial reconfiguration, the safe and secure operation of the overall system heavily depends on the *Partial Reconfiguration Controller* (PRC) of the FPGA which typically initiates the reconfiguration process in the design. In particular, reconfiguration-based countermeasures forming a “moving target” for the attacker may completely collapse, if the underlying IP-based reconfiguration fails or is attacked.

Contribution In this work, we leverage the container principle—originally proposed as a safety mechanism—to the security domain. We present a tailored encapsulation scheme for the PRC with the goal to protect the PRC against attacks when performing for example reconfiguration-based cryptographic operations. The new architecture consists of individual *ReCoFuses*. During PRC operation (i.e., reconfiguration) all properties have to hold at any given time. To guarantee the correctness of each *ReCoFuse*, we require the formal verification of its behavior, i.e., to formally capture which PRC communication is “good” or “bad” and what will be the resulting action in the respective case. Overall, the *ReCoFuses* are integrated into the *ReCoFuse Container*.

For demonstrating the proposed scheme, we consider systems which use reconfiguration-based countermeasures and by this implementing the above mentioned moving target principle. Mentens et al. showed in [23] that introducing temporal jitter based on reconfiguration improves side channel attack resistance significantly. They

also introduce a spatial jitter to improve the countermeasure against fault attacks by randomly changing physical location of functional blocks on-chip area at runtime. In their work, the importance for securing the reconfiguration control (i.e., the PRC) has already been recognized, but was not targeted there (as well as in many following papers). A more recent work which also uses *Partial Reconfiguration* (PR) to protect cryptographic circuits against different types of physical attacks has been introduced in [15]. They generate different physical configurations of the same RTL description of an AES algorithm. Then, the PRC loads the different configurations after each encryption. However, no measures are taken to prevent possible attacks on the PRC which is of course essential for the overall security.

In the case study of this paper, we present three initial *ReCoFuses* to tackle three major vectors of attacks against the system via the PRC:

1. Attack the timing of the reconfiguration by keeping one single reconfiguration active for an extended period of time.
2. Disturb the diversity of individual reconfigurations, such that (in the worst case) the same reconfiguration is used permanently.
3. Alter the reconfiguration such that a different (much simpler) logic block is used in place of the intended (cryptographic) functionality.

In all cases, the moving target becomes a static one making reconfiguration-based countermeasures against physical attacks useless.

Outline The paper is structured as follows: First, Sect. 2 reviews the preliminaries of partial reconfiguration and formal verification. In Sect. 3, related work is presented. Section 4 describes the adversary model we consider in this work. Our proposed encapsulation scheme for the PRC, implemented as *ReCoFuse Container* as well as an extensive case study, that demonstrates the advantages of our scheme for a reconfiguration-based encryption system, is introduced in Sect. 5. The experimental evaluation, i.e., fault injection and resource utilization, is reported in Sect. 6. Finally, the paper is concluded in Sect. 7 and future work is discussed in Sect. 8.

2 Preliminaries

In the course of this work, different technical concepts are introduced to convey the *ReCoFuse*-based protection scheme of this work. The following topics will be covered briefly:

Fig. 1 Overview of partial reconfiguration infrastructure

- Basics of partial reconfiguration with references to more extensive technical explanations on the topic.
- Basic introduction to verification, based on System Verilog Assertions, using the commercial tool Onespin 360 DV-Verify.

2.1 Basics of partial reconfiguration

Partial reconfiguration is implemented with highly proprietary means inside the FPGA depending on the FPGA model. Different manufacturer achieve partial reconfiguration with different components. In the course of this work, we will focus on the specific implementation of partial reconfiguration from Xilinx [43], but our approach is also applicable for other FPGA vendors.

Figure 1 presents the essential components of the partial reconfiguration infrastructure:

- *Reconfigurable Partitions* (RPs) describe the area and position in the FPGA, where *Reconfigurable Modules* (RMs) are placed (see $RP_{0...11}$ in Fig. 1).
- *Reconfigurable Modules* (RMs) represent the actual implementation which serves as replacement at runtime (see dashed squares in Fig. 1). For each additional RM a new partial bitfile is generated which is stored in the memory to be accessed by the PRC. (see MEM in Fig. 1)
- The partial bitfile represents the *actual* configuration data for the RM in the FPGA. This data is stored in on or off chip memory and it contains the configuration of the logic primitives (e.g., LUTs, DSPs, RAMs) and the connections in the RMs. For reconfiguration, such a bitfile is fed to the *Internal Configuration Access Port* (ICAP) by the PRC at runtime.

- The ICAP implements the access to the partial reconfiguration infrastructure (see ICAP connecting the PRC to the RPs in Fig. 1). It is treated as a regular primitive in the tool flow during development, synthesis and place and route.
- The PRC is necessary to control the reconfiguration process in the FPGA. It is attached to the memory (e.g., via AXI4), holding the (partial) bitfiles, as well as to the aforementioned cell primitive. Depending on the manufacturers, different options are available, such as internal or external triggers to perform the reconfiguration.

2.2 Formal verification

Formal verification as used in this work is the task of checking whether a circuit implementation satisfies its specification or not [7]. In industrial practice techniques based on *Bounded Model Checking* (BMC) gained popularity very fast since they can scale to larger designs [3, 11]. For a LTL formula φ (formalized part of the specification) the basic idea of BMC is to search for counter-examples to φ in executions of the system whose length is bounded by k time steps. More formally, this can be expressed as:

$$BMC^k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \neg \varphi^k$$

where $I(s_0)$ denotes the predicate for the initial states, T denotes the transition relation and $\neg \varphi^k$ constraints that the property φ is violated by an execution of length k . In case of simple safety properties of the form AGp where p is a propositional formula, the violation of the property reduces to $\bigvee_{i=0}^k \neg p_i$, where p_i is the propositional formula p at time step i . The overall problem formulation is then transformed into an instance of *Boolean Satisfiability Problem* (SAT). If this instance is satisfiable, a counter-example of length k has been found.

Using SAT as underlying machinery for BMC is feasible as the current state-of-the-art SAT solvers [35] can solve millions of clauses in moderate time. Besides BMC, tools have evolved to use *inductive reasoning* [44] and more recently *Property-Directed Reachability* (PDR) [5, 9] techniques which give vast improvements over the original method.

Several standardized property specification languages are available. *SystemVerilog Assertions* (SVA) is one of the most common property specification languages. In this work, we use SVA in combination with *Timing Diagram Assertion Library* (TiDAL) which comes with the commercial verification tool 360 DV-Verify from OneSpin [26]. TiDAL allows one to specify the temporal properties in a very intuitive way, i.e., (a) the time points when an expression

is evaluated can be explicitly defined and (b) the properties follow a logic implication style.

A simple property example is presented in Listing 1.

```

1  property check_req_ack ;
2  (
3  // Assume part
4  t ##0 req == 1
5  implies
6  // Prove part
7  t ##3 ack == 1
8  );
9  endproperty

```

Listing 1: Example property

This property states that if request is 1 at timepoint $t + 0$ (assume part), then three clock cycles later, i.e., $t + 3$, the acknowledge should be 1 (prove part). Such properties can be verified on the circuit. If a property fails, a counter example is provided, i.e., a wave trace which can be simulated which shows the violation of the property.

In case of larger numbers of properties the time, spent for verification, will increase. However, as we show later in the experiments, property checking of the proposed ReCoFuses can be performed very fast.

3 Related work

The PRC is an IP component of the respective FPGA vendor. Besides this black box realization, researchers have implemented their own PRC with the focus on higher performance [37], better timing wise predictability during reconfiguration [28], multifunctionality [12] and even fault tolerance [34]. Dedicated protection of the PRC has not been considered in these works.

There are many research items for secure FPGA reconfiguration. The focus is given for bitstream security using authenticated encryption [1, 16]. In [17], the external memory where partial bitstreams are stored is considered as unsecure location. Therefore, a novel solution is presented to securely store them. However, these works do not focus on the protection of the reconfiguration itself, particularly the PRC, which is at least as important as the bitstream integrity.

The authors of [18] proposed the secure reconfiguration controller (SeReCon). Semantically, it also provides an additional barrier to the partial reconfiguration infrastructure. This effectively forms an additional anchor of trust in terms of a gateway to the reconfiguration infrastructure in the design, granting more reliability in the case of IP core based reconfigurable FPGA systems. However, the aforementioned work primarily focuses on authentication of IP cores (in this context bitstreams for partial reconfiguration).

Xilinx announced a security monitor based on an IP soft core and more recently a partial bitstream monitor IP which allows monitoring the partial reconfiguration process [39, 41]. To the best of our knowledge, no non-IP-based protection of the PRC for reconfiguration-based countermeasures is offered.

4 Adversary model

The proposed architecture provides increased protection against attacks targeting reconfiguration-based countermeasures. Adversaries are derived from assumptions made in [23] and [21], allowing passive and semi invasive attacks. The malicious user desires to extract confidential information from the system by exploiting available attack measures to circumvent the security mechanisms.

Differential Power Analysis (DPA) represents a passive attack scenario where the malicious user can obtain—possibly a very large number—power consumption measurements of the attacked system over time. Since activity in the design's circuitry correlates to its power draw, DPA allows attacks based on statistical methods (e.g., Welch's T-test [32]) to successfully extract cryptographic secrets. These attacks can be carried out with relatively small investments, since computer-based oscilloscopes are readily available at decreasing price points.

For semi-invasive attacks, we assume an adversary, who can disturb (or deactivate) the reconfiguration procedure, thus leaving the system vulnerable to the aforementioned DPA-based passive attacks. Only on die attacks are assumed for this scenario. If mitigation against DPA is based on partial reconfiguration, directly attacking the PRC is most rewarding, since failing reconfiguration will leave the system unprotected¹. Where a single attack was sufficient in the past, the attacker must now attack at least two places at the same time to break the reconfiguration-based protection.

In practice, injecting faults into multiple wires or positions in the FPGA fabric increases the cost of an attack. Multiple instances of the proposed protection scheme allow mitigation (i.e., out-scale) of attackers, by employing n modular redundancy in terms of ReCoFuses².

A second vector of attack is offered from blackbox IP cores in the design. As motivated in the introduction, malfunctions, flaws or malicious intents can jeopardize the

¹ This work does not cover non-reconfiguration-based countermeasures, such as *Threshold Implementations* which were presented in [24].

² Please note that we advise to distribute (place) the ReCoFuses evenly in the FPGA, while attaching them to different clock buffers or PLLs.

Fig. 2 Original PRC Architecture vs Proposed ReCoFuse Container Architecture

system's stability. Even Trojans in cryptographic hardware blocks were reported in the literature [2]. If the IP core in the design is considered an adversary, it has direct access to signal lines inside the circuitry (e.g., stalling a shared bus). This scenario was reported to be realistic as demonstrated in [45]. The authors demonstrated an on-chip power monitor based on ring oscillators to observe the power consumption of other modules on the FPGA. Furthermore, it allowed a DPA attack against an on-chip (i.e., same FPGA) RSA crypto module, as well as side channel attacks against the CPU of the host system (PCIe based FPGA). The proposed RCFs must be capable to capture malformed communication with the surrounding system and reliably detect malicious behavior during operation (i.e., skipped reconfigurations in this particular use case).

Another vector of attack results from bitstream relocation. Tools such as RapidSmith2 [13] and Bitman [29] can be used to modify partial bitstreams to allow sending them into different FPGA locations. This principle can be used to reduce the memory needs when storing reconfigurations. However, there is also a drawback as a misuse that allows reconfiguring hardware blocks into locations where they were not intended for. This potentially allows an attacker to perform trivial data modifications via not intended reconfigured hardware blocks instead of cryptographic ones. Therefore, this attack would break the security.

5 ReCoFuse container

This section presents our encapsulation-scheme for the *Partial Reconfiguration Controller* (PRC) of a FPGA which implements reconfiguration-based countermeasures against physical attacks. Also it provides to supervise and monitor the reconfiguration process. The scheme is based on two main components: (1) A "container" encapsulating the PRC, and (2) individual ReCoFuses to monitor and react on untrusted communication with the PRC which would compromise the security or stability of the reconfiguration-based countermeasures. The compartmentalization, i.e., the container, is not a novel concept. However, it provides the means to interconnect the ReCoFuses, the PRC

and the supporting logic (e.g., for error handling) and is being used for another task. From an architectural point of view, it is the top-level of the proposed solution and is therefore presented in detail.

In the following, we first introduce the overall architecture of the ReCoFuse Container. Then, we detail the interfacing of the PRC and the ReCoFuse Container which hosts the individual ReCoFuses. Finally, the required formal verification of ReCoFuse behavior is described.

5.1 Architecture of ReCoFuse container

The left part of Fig. 2 depicts the original unprotected PRC architecture. On the right of Fig. 2 the proposed architecture realizing our encapsulation-scheme for the PRC is shown.

The encapsulation-scheme is employed in an effective way by considering to keep the area overhead and impact on maximum frequency of the overall design low. It is independent from the internal structure of the PRC and the to-be-protected design. The necessary data is obtained from inputs and outputs of the PRC that communicates with the reconfiguration memory via AXI4 standard protocol and the ICAP interface. Therefore, there is no need to know or interfere the internal functionality of the PRC.

As can be seen the ReCoFuse Container has several "slots" for individual ReCoFuses (details see next section). The ReCoFuses are denoted as $RCF_{0...n}$ in Fig. 2. Moreover, all outgoing data connections between the main components, i.e., Reconfiguration Memory, PRC and ICAP, are now also fed into the ReCoFuses. Furthermore, a configuration register has been added which allows the user to dynamically enable or disable each RCF.

5.2 Interfaces and ReCoFuses

Listening on all reconfiguration interfaces allows to monitor the reconfiguration operations requested by the reconfiguration-based countermeasures. In Fig. 2, these are the AXI4 and ICAP interfaces. The ICAP provides access to the configuration functions from within the FPGA fabric. Commands and data can be written to and read from the configuration logic of the FPGA array. The PRC has a direct

Table 1 ICAP interface port descriptions [42]

Port	Direction	Width	Function
CLK	Input	1	Clock Input
CSIB	Input	1	Active Low ICAP Enable
I<31:0>	Input	32	Configuration data input bus
O<31:0>	Output	32	Configuration data output bus
RDWRB	Input	1	Read/Write Select input

interface to the ICAP. The ICAP protocol follows a valid/acknowledge scheme where the header of each partial bitfile can be analyzed during data communication. Therefore, hardware debug and ensuring RM initiation is possible by monitoring the ICAP interface via the *Integrated Logic Analyzer* (ILA) core from Xilinx. The port descriptions of the ICAP interface are shown in Table 1. For more details, we refer to the Xilinx 7 Series partial reconfiguration user guide [42]. Table 2 shows an excerpt from the user guide, including the sync word, which initiates every communication between PRC and ICAP.

As can be seen in the architecture in the right side of Fig. 2, the observed input data is sent to the ReCoFuses. A ReCoFuse essentially implements a *Finite State Machine* (FSM), and hence performs state transitions based on the observed data. Figure 3 illustrates the generic FSM structure of a ReCoFuse. The `WAIT_STATE` awaits the initial header from the PRC, which is dependent on the type of the ReCoFuse to start evaluation of the observed data. The `ACTION_STATE` which may include substates determines whether there is a violation or not. When there is no violation, the `ACTION_STATE` always returns to the `WAIT_STATE` and the operation of the PRC is considered as legitimate. If a violation is detected, the `ACTION_STATE` transitions to the `BAD_STATE` and the operation of the PRC is considered as untrustable. The output signal of a ReCoFuse (e.g., in this work an error signal) allows each ReCoFuse to communicate untrusted behavior. As a consequence emergency actions can be executed, for instance to shut down the system. For simplicity, in Fig. 2 on the right we have just ORed all the error signals from each ReCoFuse.

5.3 Verification of ReCoFuses

To guarantee the correctness of each ReCoFuse, we require the formal verification of its behavior. Formal verification

Fig. 3 Generic FSM diagram for a ReCoFuse

allows to prove the correctness of the functional behavior of each ReCoFuse. For this task the RTL code of a ReCoFuse is automatically transformed in a suitable formal model by the property checker. The user only has to specify the temporal properties describing the different state transitions of a ReCoFuse. In other words, these properties are used to prove which PRC communication with the control of the reconfiguration-based countermeasures is untrusted and what will be the resulting action in that case. Typically, a ReCoFuse observes the communication over several clock cycles and finally reaches a “bad” state. An example property for this last proof step basically states the following: If a ReCoFuse enters the bad state, the associated action must be taken in the next clock cycle. The corresponding property is shown in Listing 2.

We have developed the ReCoFuses based on viable attacks from our adversary model on the PRC. As a consequence, we have defined dedicated verification properties. While each ReCoFuse of course realizes specific functional behavior, there is a common property structure as in essence the user has to capture each state transition. In industrial practice there are process models to create strong verification plans which allow to ensure full correctness.

```

1  property raise_error;
2  (
3    t ##0 enter_bad_state()
4  implies
5    t ##1 raise_error_signal()
6  );
7  endproperty
  
```

Listing 2: ReCoFuse Signal Bad State Property

5.4 Individual ReCoFuses

The concrete ReCoFuses are presented in the following three sections.

Table 2 Partial bitfile header

Address	Description	Comment
0xAA995566	Sync Word	Initiation of bitfile transmission
0x30002001	Frame address register	Target Address for partial reconfiguration
0xE3AD7EA5	CRC32 for Payload	Checksum for bitfile payload

Fig. 4 FSM diagram of the timer ReCoFuse

5.4.1 Timer ReCoFuse

Functionality: The timer ReCoFuse (RCF_0) basically keeps track of the time between two consecutive reconfigurations. The FSM diagram of the timer ReCoFuse is shown in Fig. 4. After a successful reconfiguration (`WAIT_RECONFIG`), a timer is started in particular a counter is increased in allowed time duration (`ADVANCE_TIMER`).

If this timer expires *before* a new reconfiguration procedure is initiated, the timer ReCoFuse signals an error (`DETECT_ERROR`). Keeping a specific reconfiguration active for an extended period of time—rendering the moving target principle ineffective—can be detected reliably by this ReCoFuse.

Interface Events The counter of the time-out ReCoFuse is started by a `RP_active`, which is derived from several signals, provided by the proprietary reconfiguration infrastructure. Alternatively, the sync word (cf. Table 2) in combination with the bitfile length could serve for the same purpose.

```

1  property advance_timer;
2  disable iff (rst) (
3    t ##0 (cnt!=TIMEOUT and RP_active)
4    implies
5    t ##1 (cnt== $past(cnt)+1)
6  );
7  endproperty
8
9  property detect_error;
10 disable iff (rst) (
11   t ##0 (cnt==TIMEOUT and RP_active)
12   implies
13   t ##1 (error)
14 );
15 endproperty

```

Listing 3: Example properties for timer

Verification In Listing 3, a subset of the properties for verifying the timeout ReCoFuse RCF_0 are shown. The first property `advance_timer` (Line 1–Line 7) states that the counter (which realizes the timer) advances with each time step after the previous reconfiguration is done. Here, `TIMEOUT` (Line 3) defines the allowed active duration of one Reconfiguration Module (RM), i.e., a concrete AES implementation. The `RP_active` (Line 3) signal is derived from multiple signals from the reconfiguration

infrastructure and captures whether the Reconfigurable Partition (RP) is active, i.e., no reconfiguration is currently performed. In Line 5, the `past()` statement is used to refer to the previous time point.

`detect_error` names the second verification property (Line 9–Line 15 in Listing 3). It ensures that RCF_0 enters the “bad” state (i.e., raising `error`), when the respective RM was not reconfigured in time (i.e., before `cnt` reaching `TIMEOUT`) (Line 11).

5.4.2 Replay ReCoFuse

Functionality: The replay ReCoFuse (RCF_1) contains an individual counter for each Reconfiguration Module (so, different AES implementations in our case study), i.e., functional alternative which is swapped in. Based on the individual counter values the distance of the *Least Frequently Used* (LFU) RM as well as the *Most Frequently Used* (MFU) RM is determined. This distance indicates whether the usage of the available RMs is uniform. Hence, this forms an effective measure to detect if a specific RM is preferred or used continuously, since the corresponding counter will advance *faster*. To illustrate the developed uniformity check, Fig. 5 shows a reconfiguration procedure over 13 reconfigurations (i.e., steps) in form of a bar chart, choosing from four different RMs. In Fig. 5, the y axis shows the four different RMs (i.e., different AES implementations). The x axis shows the frequency, how often the RMs were reconfigured. For example, after 2 time steps only RM_1 and RM_2 have been reconfigured both once; after 6 time steps this changes to respective frequencies of (1, 2, 2, 1) (for RM_1 , RM_2 , RM_3 , RM_4).

A challenge when implementing this ReCoFuse in hardware, was that the logic (i.e., the counters) should not become too costly. The solution was the implementation of a shift window operation which essentially “cuts” all counters (similar to a histogram) at the *bottom*. As a consequence, the least frequently used counter is *zero aligned*. Figure 5 depicts this “cutting” in terms of the shift window operation in the left (highlighted gray), while preserving the distance between the LFU and MFU RM, i.e., shift window reduces the counters from (1, 2, 2, 1) to (0, 1, 1, 0) after $step_6$.

For the example at hand, we allow a distance of 6 between the least frequently used RM and the most frequently used RM. Assuming an attack (e.g., a replay attack) resulting in a more frequent reconfiguration of RM_1 is depicted in the figure: In $step_{13}$ we see a violation of our security condition of $MFU - LFU = 7 - 0 = 7^3$ and hence an error is signaled by the ReCoFuse.

³ The gray boxes have been removed by the shift window operation, so the counters are (7, 1, 1, 0).

Fig. 5 Uniformity check and shift window operation

Interface Events The uniformity check of the replay ReCoFuse is applied between reconfiguration memory and PRC in the AXI communication. A unique identifier of the individual RMs can be derived from the *Frame Address Register* (FAR) value (cf. Table 2) together with its address in the

configuration memory (cf. Table 3). To increment a specific counter, the replay ReCoFuse scans the transmissions on the AXI interface for its respective identifier which can be observed when the respective bitfile is loaded by the PRC.

```

1  property decrease_counter;
2  disable iff(rst) (
3    t ##0 (state==SHIFT_WINDOW and RM_seen==ALL_RM)
4  implies
5    // omitted technicality
6    t ##1 (state==SYNQ and RM_seen==NO_RM and cnt== $past(cnt)-1)
7  );
8  endproperty
9
10 property detect_error;
11 disable iff(rst) (
12   t ##0 (dist==MAX and state==CHECK_ERR) and t ##1 (state==CHECK_ERR)
13 implies
14   t ##1 (error)
15 );
16 endproperty
17
18 property increase_counter;
19 disable iff(rst) (
20   t ##0 (state==CNT_UPDATE and cnt<MAX)
21 implies
22   t ##1 (state==CHECK_CNT and cnt == $past(cnt)+1)
23 );
24 endproperty
25
26 property detect_RMn_ID;
27 disable iff(rst) (
28   t ##0 (state==CNT_UPDATE and RM==RMn_ID and RMn_detected==1'b0 and RM_seen<ALL_RM)
29 implies
30   t ##1 (state==CHECK_CNT and RMn_detected==1'b1 and RM_seen== $past(RM_seen)+1)
31 );
32 endproperty

```

Listing 4: Properties for uniformity check**Table 3** Identifier of reconfigurations

Reconfigurations	Bitstream address
RM ₁	0x00200000
RM ₂	0x00400000
RM ₃	0x00600000
RM ₄	0x00800000

Fig. 6 FSM diagram of the replay ReCoFuse

Verification Figure 6 shows the state diagram of the replay ReCoFuse. It starts with WAIT_FAR and WAIT_SYNC states to detect SYNC_WORD and FAR value. The other state transitions are explained with the verification properties. In Listing 4, a subset of the properties to verify the behavior of the replay ReCoFuse are shown. Please note that the shown 4 properties are checked for each RM since the replay ReCoFuse has per RM an individual counter as explained above. The decrease_counter property is central to the shift window operation ($state == SHIFT_WINDOW$) in hardware. It is ensured that all counters are decreased (i.e., previously mentioned “cutting”, $cnt == \$past(cnt) - 1$) when all RMs were active at least once ($RM_seen == ALL_RM$). In order to immediately capture new reconfigurations, the underlying FSM must transition to the SYNQ state, where it screens the AXI communication for the RM identifier. RM_seen is reset ($RM_seen == NO_RM$) in the next step to allow continuous counter cutting.

The second property in Listing 4 is called detect_error starting from Line 11. Following the idea from Listing 3, the ReCoFuse must raise its error signal, when the maximum allowed distance ($dist == MAX$) is exceeded. A dedicated error checking state (CHECK_ERR) in the FSM checks this violation (Line 13 + 14) and raises the error signal in the next cycle (Line 16). The FSM remains in this error state (i.e., the “bad” state).

The third property increase_counter starting from Line 20 checks counter increase. When RM comes in (CNF_UPDATE) state and if it does not reach the allowed ($cnt == MAX$) value (Line 22), the individual counter of the RM must increase in the next cycle and FSM must transition to (CHECK_CNT) state (Line 24 + 25). (CHECK_CNT) is the main state where the values of all counters are checked in order to determine the next operation. Based on the status of the counters, in the next cycle (CHECK_CNT) might transition to

- (CHECK_ERR) if the maximum allowed distance will exceed ($dist == MAX$).
- (SHIFT_WINDOW) if all RMs will be active at least once ($RM_seen == ALL_RM$).

- (SYNQ) if the upper cases will not occur.

The fourth property detect_RMn_ID has the same state transition like the previous one (from CNF_UPDATE to CHECK_CNT), but it has to be written as a separate property because it checks if a RM is active for the first time. If the RM was not detected before ($RMn_detected == 0$) and all RMs were not active ($RM_seen < ALL_RM$), the RM is marked as detected ($RMn_detected == 1$) and one more RM must be indicated as active ($RM_seen == \$past(RM_seen) + 1$) in the next cycle (Line 31 to Line 35).

5.4.3 Relocation ReCoFuse

Functionality: In order to address the issue of intentionally misplaced RMs, we created a new ReCoFuse, which supervises the functionality of individual bitstreams bound to their location. This ReCoFuse shall provide means to identify, whether a RM is placed to the wrong RP. Ideally, the ReCoFuse would contain an exact implementation as the RM, such that it can be identified, if the module is placed correctly (i.e., comparing inputs vs. outputs). As a consequence a “reduced” version of the module’s logic needs to be accommodated within the ReCoFuse. A simulation model of the circuitry in the design, however, would still remain too expensive, since the entire state space (including data path) would have to be covered. Since Linear Feedback Shift Registers (LFSRs) can be used (in conjunction with a generator polynomial) to generate checksums or to create pseudo random bit sequences, these offer a solution to this problem.

A LFSR advances the signal through the register from one bit to the next most-significant bit. The signal input to

Fig. 7 4-bit LFSR

Fig. 8 Relocation ReCoFuse

Table 4 Look-up table of Relocation RCF

Reconfigurations	Bitstream address	Seed
RM ₁	0x00200000	b
RM ₂	0x00400000	c
RM ₃	0x00600000	d
RM ₄	0x00800000	e

the LFSR is generated by xor-ing or xnor-ing the tap bits; the remaining bits function as a standard shift register. As an example, a 4-bit LFSR is shown in Fig. 7. When such a LFSR is set to its initial value (e.g., initialization vector/seed), it will create a pseudo-random sequence of output values which finally will return to the initial state. A very long cycle sequence patterns can be produced with a well-chosen feedback function (i.e., polynomial).

A LFSR with a known polynomial and a known initialization vector will yield a known bit pattern (or bit) after a known number of steps (i.e., clock cycles). Hence, we decided to integrate a separate LFSR as a pseudo-unique fingerprint alongside the logic in the RM and inside the ReCoFuse. When the two LFSRs are driven with the same data and the same clock source (i.e., similar to lock-step CPUs), the LFSRs will always yield the exact same output / state when checked (XOR gate). Since the XOR gate drives the error signal directly, a mismatch between the golden reference output from the ReCoFuse's LFSR and the RM's LFSR will indicate an error. This functionality allows the construction of countermeasures for the detection of misplaced or altered partial bitfiles.

Each RM owns a LFSR which is assigned to its specific to its implementation (i.e., a different polynomial). We differentiate between functionality and variant. Different functionalities are equipped with different LFSR types, whereas different variants are equipped with same type of LFSR but initialized with different seeds. Each functionality belongs to a predetermined specific location (floor plan of RP). In our case-study the functionality is always the AES core, but we have different variants (i.e., implementations of AES in the form of various netlists). We now illustrate this approach. Figure 8 is composed of the three main parts to show the protection scheme of the RCF:

- On the left hand side of Fig. 8, the different RMs with their LFSR logic can be seen. They are set up with their own seed in the beginning.
- In the middle of Fig. 8, the RCF logic is depicted. It consists of the following three components:
 1. A look-up table contains the predetermined address and seed values of each RM. For our case-study the concrete values are shown in Table 4.
 2. The FSM in the middle manages state transitions. The detailed structure is shown in Fig. 9. The initial states are `WAIT_FAR` and `WAIT_SYNC` to detect `SYNC_WORD` and `FAR` value like replay ReCoFuse. After transition to `READ_SEED` state the id number of a reconfiguration is detected, then the seed value of individual reconfiguration is read. This state also enables the LFSRs via the `lfsr_en` signal after the reconfiguration process has completed to guarantee synchronization.
 3. Finally, each value of the RCF LFSR is compared against the corresponding value of the current RM LFSR via the XOR when FSM is in `COMPARE_LFSRs` state. If output values of the LFSRs do not match, FSM enters `DETECT_ERROR` state. This check allows to identify the attack as an unintended logic block reconfigured to an unintended RM would result in a deviation of the LFSR sequence.
- On the right hand side of Fig. 8, an error-signal is derived to indicate the erroneous state of the system.

Fig. 9 FSM diagram of the relocation ReCoFuse

Interface Events Similar to the replay ReCoFuse, the relocation ReCoFuse is applied between reconfiguration memory and PRC. It also keeps track of the unique identifier of RMs from the respective bitfile in order to read the corresponding seed for each RM.

Verification In Listing 5, a subset of the properties to verify the behavior of the relocation ReCoFuse are shown. The `inactive_error` property (from Line 1 to Line 7) checks when LFSRs are not enabled (reconfiguration was not yet completed). Subsequently the error signal must be inactive.

The `active_error` property (from Line 9 to Line 16) checks when LFSRs are enabled (reconfiguration has completed) and the output values of the ReCoFuse's LFSR and RM's LFSR are not the same, hence the error signal must be raised since not the intended logic has been reconfigured at the current location, but another functionality.

The `active_no_error` property (from Line 18 to Line 25) checks when the LFSRs are enabled (reconfiguration completed) and the output values of the ReCoFuse's LFSR and RM's LFSR are identical, there must be no error.

The `read_seed` property (from Line 27 to Line 34) checks when the identifier of the individual RM was seen and the predetermined seed value of the individual RM should be read in the next clock cycle.

```

1  property inactive_error;
2  disable iff (rst) (
3    t ##0 (!lfsr_en)
4  implies
5    t ##1 (!error)
6  );
7  endproperty
8
9  property active_error;
10 disable iff (rst) (
11   t ##0 (lfsr_en) and
12   t ##0 (o_LFSR1 xor o_LFSR_aes)
13 implies
14   t ##1 (error)
15 );
16 endproperty
17
18 property active_no_error;
19 disable iff (rst) (
20   t ##0 (lfsr_en) and
21   t ##0 (!(o_LFSR1 xor o_LFSR_aes))
22 implies
23   t ##1 (!error)
24 );
25 endproperty
26
27 property read_seed;
28 disable iff (rst) (
29   t ##0 (state == READ_SEED) and
30   t ##0 (RM == RMn.ID)
31 implies
32   t ##1 (seed == seed_RMn)
33 );
34 endproperty

```

Listing 5: Property list for relocation

In Listing 6, there are two common properties for replay and relocation ReCoFuse. These check the state transition after (SYNC_WORD) and (FAR) values (cf. Table 2) were seen.

```

1  disable iff (rst) (
2    t ##0 (state == WAIT_SYNC and SYNC_WORD == 32'hAA995566)
3  implies
4    t ##1 (state == WAIT_FAR)
5  );
6  endproperty
7
8  property detect_FAR;
9  disable iff (rst) (
10   t ##0 (state == WAIT_FAR and FAR == 32'h30002001)
11 implies
12   t ##1 (state == CNT_UPDATE)
13 );
14 endproperty

```

Listing 6: Common properties for replay and relocation

Fig. 10 Behavior of RCF_0 (time-out)

In the following we demonstrate our proposed scheme on a concrete case study.

5.5 Case study

This section demonstrates the proposed encapsulation-scheme for the PRC. AES officially superseded DES and was presented by the NIST in 2001 [33], since DES was not considered secure anymore. Unfortunately, sophisticated side channel attacks allow attackers to extract protected information nonetheless, for which countermeasures were presented [23]. However, these dedicated countermeasures can be attacked again, thus rendering the protection ineffective. Due to this shortcoming and the omnipresence of AES in soft- and hardware, AES-based designs are well suited for evaluating our approach.

The system implements the moving target principle via reconfiguration by switching between different implementations of an AES core. By this, the attacker is not faced with static logic in the FPGA, but with a permanently changing one and hence physical attacks become much harder to achieve.

In the following, we first describe three major attack vectors. Then, we present the ReCoFuse Container and three ReCoFuses. Finally, we consider their verification.

5.5.1 Attack vectors

We assume, the cryptographic system is hardened by employing partial reconfiguration as a countermeasure against DPA. Breaking the countermeasures (i.e., partial reconfiguration) makes the system susceptible against DPA again. In principle DPA uses the fluctuation in a device's power consumption to automatically extract information on the implementation of the design and the processed data. Since the attack is easy to implement and is non-invasive, it poses a significant threat which was recognized by Kocher et al. [20] in 1999.

In order to attack a specific area (e.g., the PRC) in a FPGA, electromagnetism and fault injection based attacks have both been reported to be effective [22] and are viable methods for disturbing the reconfiguration process in hardware.

We identified three major attack vectors on the PRC viable for both adversaries:

1. *Time-out attack*: Forcing the PRC to keep the same reconfiguration active for a too long time would result in no protection. It removes the moving target characteristic of the design and makes it vulnerable to side channel attacks.
2. *Replay attack*: Attacking the PRC to chose a single reconfiguration continuously (or more often) removes the moving target characteristic as well.
3. *Relocation attack*: Since the modification of bitstreams can easily be achieved with tools, such as BitMan, the integrity of a design is not guaranteed. Even if the PRC operates correctly, the design's correct operation is not ensured anymore. The attack scenario introduces bitstream manipulation w.r.t. the previous scenarios. A practical example is the modification of the FAR values, while maintaining correctly computed CRCs. This provides the attacker with the ability to place unsuited PRs into unsuited RPs.

This list, however, is not exhaustive and can be extended by the respective adopter's needs. The next part presents how the proposed ReCoFuse Container helps in protecting against the three attacks.

5.5.2 ReCoFuse container

We encapsulated the PRC in a ReCoFuse Container. It instantiates the PRC and provides connection to the configuration memory via AXI and the ICAP primitive as described in Sect. 5. ReCoFuses are integrated inside the ReCoFuse Container to achieve countermeasures against the time out, replay and relocation attack.

In the following section, we present an experimental evaluation of our approach for our case study.

6 Experiments and results

All experiments have been conducted on a Xilinx Zynq-7000 Series FPGA, more precisely our evaluation platform is a Zedboard featuring a XC7Z020-CLG484-1 FPGA component. More recent FPGA generations feature the same reconfiguration interface, thus our approach maintains applicability in the future. Enhanced capabilities, such as better encryption and authentication however can help to increase the difficulty of attacks further. Our encryption system implements the moving target principle by

Fig. 11 Behavior of RCF₁ (replay) **Fig. 12** Behavior of RCF₂ (relocation)

switching between different implementations of the AES core (based on tiny_AES⁴ IP core from OpenCores.org) via reconfiguration. A dedicated controller in the FPGA (called *SYSCTRL*) initiates the random (i.e., uniform) replacement of a *Reconfiguration Module* (RM), i.e., between the different AES implementations.

We have synthesized the encryption system using Vivado 2016.3. The partial bitfiles are copied from the SD card to the on board DDR3 memory (serves as partial bit-file memory), using a bare metal executable which runs on the ARM core of the FPGA. To access from the programmable logic of the FPGA, we switched the DDR3 memory to AXI slave mode. The PRC is directly attached to the AXI slave DDR3 memory in the design and instantiates the ICAP primitive as well. A ReCoFuse Container encapsulates the PRC as presented in Sect. 5. The three ReCoFuses time out (RCF₀), replay (RCF₁) and relocation (RCF₂), as described in Sect. 5.5.2, are integrated in the ReCoFuse Container to protect against the three attacks as introduced in Sect. 5.5.1.

6.1 Injecting faults

As mentioned above, the controller *SYSCTRL* for reconfiguring between the different AES implementations initiates the random (i.e., uniform) replacement of a RM and for this communicates with the PRC. During normal operation *SYSCTRL* replaces the current RM with a random successor before the timer of RCF₀ expires, such that no ReCoFuse

Table 5 Hardware resource utilization of AES core

Elements	RP block	AES core (Original)	AES core (LFSR added)
Slices	2800	2537	2552
LUT as Logic	11200	9669	9672
LUT as Memory	3000	0	0
LUT as FF Pairs	11200	3657	3708
Slice Registers	22400	5504	5520
Muxes	5600	3600	3600

raises an error. To run the experiments, we attacked the reconfiguration process by injecting faults in the encryption system in order to disturb the operation of the PRC. This was achieved by additional logic on the FPGA. Essentially, we disable the initiation of the replacement at runtime or alternatively remove the randomness from the RM selection. The following results have been obtained from actual hardware as the setup has been introduced Sect. 6. We used Xilinx integrated logic analyzers (ILAs) [40] in order to monitor the internal signals of the ReCoFuses while our design is running on FPGA:

6.1.1 Time-out attack

Figure 10 illustrates the functionality of RCF₀ for the time-out attack. The timer must expire, if a RM is kept active longer than acceptable; we set the TIMEOUT to 10 time steps. For demonstration we captured the activity for 960 ms (i.e., 15 time steps). After 64 ms, RM₁ is loaded, followed by, RM₂ and RM₃ (each active for 1 time step (64ms)). RM₄

⁴ https://opencores.org/project/tiny_aes—configured to provide 128-Bit AES encryption after 21-cycles.

Table 6 Hardware resource utilization of each ReCoFuse

Elements	Timer RCF	Replay RCF	Relocation RCF
Slices	16	54	41
LUT as Logic	15	88	55
LUT as Memory	0	0	0
LUT as FF Pairs	2	27	11
Slice Registers	34	127	107
Muxes	0	0	0

is kept active indefinitely (10 time steps), which exceeds the acceptable period (640 ms), such that the error signal is raised, when the counter value (`ctr`) reaches 640. The `error` signal indicates a violation of the time-out requirement.

6.1.2 Replay attack

Figure 11 shows a sequence of the reconfigurations after 15 steps. We have four different RMs. The maximum allowed distance of the RMs is set to 6 as in Fig. 5. In step 7, the occurrence of RM_4 decreases all counters by the shift window operation, resulting in a *zero alignment* of

all counters. At this point, the PRC is attacked (i.e., internally triggered faults are injected). In the 15th step, loading RM_1 will activate the error signal of the RCF_1 . Since the difference between the most and least frequently used RM exceeds the allowed limit.

6.1.3 Relocation attack

Figure 12 depicts the behavior of the relocation ReCoFuse RCF_2 . Until step 13 no relocation violation is detected. For the three reconfigurations RM_1 , RM_2 and RM_3 (representing different AES implementations) the LFSR values inside the ReCoFuse (`lfsr_RCF` and the `lfsr_RM`) always match.

This is the case, since their respective initialization seeds were initialized to the same value and the implemented polynomials are identical, hence, the sequence of bits to be compared is also identical. In step 12, we conduct an attack to the PRC by means of loading an unintended reconfiguration (RM_4). The seed applied by the RCF to the RM's LFSR is not identical to the seed, which was loaded to the RCF's LFSR. As a consequence, both LFSR values do not match anymore and hence the error signal is triggered in step 13 as required by the verification before.

Table 7 Hardware resource utilization of overall system

Elements	"Moving target AES": Original		ReCoFuse protected			ReCoFuse protected		
	Usage	Util (%)	Usage	Util (%)	Increase (%)	Usage	Util (%)	Increase (%)
Slices	2976	22.38	3036	22.83	2.02	3058	22.99	2.75
LUT as Logic	10520	19.77	10620	19.96	0.95	10704	20.12	1.75
LUT as Memory	203	1.17	203	1.17	0.00	203	1.17	0.00
LUT FF Pairs	4367	8.21	4409	8.29	0.96	4436	8.34	1.58
Slice Registers	6744	6.34	6906	6.49	2.24	7023	6.60	4.13
Muxes	3601	13.54	3601	13.54	0.00	3601	13.54	0.00

Table 8 Run-time and Memory for Formal Verification

ReCoFuse	Property	Run-time (s)	Memory (MB)
Timer	advance_timer	0.78	96
	detect_error	1.35	98
Replay	decrease_counter	0.94	578
	increase_counter	0.88	103
	detect_RMn_ID	0.84	103
	detect_error	1.33	103
Relocation	read_seed	1.18	103
	inactive_error	0.82	102
	active_error	0.75	102
	active_no_error	0.68	235
Replay+Relocation	detect_SYNC	0.87	103
	detect_FAR	1.16	235

In summary, the three experiments based on injecting faults demonstrated the effectiveness of our approach. In the next section, we report the resource utilization of our ReCoFuse Container for the encryption system.

6.2 Resource utilization

All synthesis runs and implementation runs were executed with the same settings via Vivado 2016.3.

Table 5 presents the comparison of the resource utilizations of a pure tiny_AES core and a tiny_AES core where we have added the LFSR for fingerprinting. The first column *Elements* of the table presents the respective resource. The second column *RP block* shows the area reserved for partial reconfiguration. The third column shows the original *tiny_AES core* resource utilization and the fourth column shows *tiny_AES core with LFSR*. The LFSR length is 16-bit width a maximum-length sequence based on [38]. The tiny_AES core with LFSR still easily fits the reserved area. It can be clearly seen that the extra hardware costs are very low.

Table 6 shows the implementation results of each ReCoFuse. The Timer ReCoFuse uses 32-bit counter. The Replay ReCoFuse has 3-bit individual counters as well as auxiliary registers for each RM. The Relocation ReCoFuse includes 16-bit LFSR. In the case study we have four RMs. When we increase the number of RMs, the utilization of the Timer ReCoFuse will not change because it only considers the specific duration of any RM. However, the resource utilization of the Replay and Relocation ReCoFuse that depend on the number of RMs will of course increase, but the costs are very moderate for the achieved protection of the PRC.

Table 7 shows the implementation of our encryption system. The second column *Original:Usage* shows the number of elements used for our encryption system employing reconfiguration based on different tiny_AES cores and the Xilinx reconfiguration infrastructure following the moving target principle. The third column *Original:Util* shows the percentage of the utilization of the device. The fourth and fifth column *ReCoFuse protected: Usage and Util* contain the number and percentage of resource utilization for the encryption system protected with the previously introduced ReCoFuse Container that contains Timer and Replay ReCoFuses. The sixth column *ReCoFuse protected: Increase* shows the negligible overhead compared to the original encryption system. Finally, resource utilization for the encryption system protected with ReCoFuse Container that includes Relocation ReCoFuse besides other ReCoFuses is in the seventh and eighth columns. The last column that shows the increase compared to the original encryption system. The difference of the increase of resource utilization between ReCoFuse Container without (i.e., *Slice Registers increase:2.24%*) and with Relocation ReCoFuse (i.e., *Slice Registers increase:4.13%*) is quite

low. This proves that addition of the proposed method into the protection system still keeps the overall overhead negligible.

6.3 Verification

Finally, we want to demonstrate that the formal verification of the developed ReCoFuses can be performed very fast. Table 8 shows the run-time and memory consumption using OneSpin 360 MV version 2017_06 for property checking. As can be seen each property can be proven in less than 2s using less than 600 MB of main memory. The largest memory consumption occurred for the *decrease_counter* property because it checks all the counters.

7 Conclusions

In this work, we leveraged an originally proposed safety mechanism—which creates a container around an IP—to encapsulate and protect the PRC of a FPGA. This is very important when implementing reconfiguration-based countermeasures against SCA attacks, realizing for example the moving target principle. We introduced ReCoFuses inside our encapsulation-scheme, each capturing a specific property of interest which has to be fulfilled at any time during PRC operation. Formal verification was employed to guarantee the correctness in detecting a security violation. For evaluation of our scheme, we have created a reference design, which we attacked by injecting faults. The experiments showed that the implemented measures—leveraging the proposed scheme—realize an effective and cost efficient protection for reconfiguration-based secured designs. In summary, this work closes the gap of a vulnerable reconfiguration infrastructure as identified in [23] by Mentens et al.

8 Future work

The proposed encapsulation scheme provides the detection of attacks on the PRC, which aim to break reconfiguration-based countermeasures. Our developed architecture is able to raise an alarm (via an error signal), if a ReCoFuse detects a security violation. In this situation, instead of shutting down the system immediately, several alternative strategies can be investigated. When an error is detected by a ReCoFuse, the system could enter an emergency mode and maintain the operation. For this spare reconfigurations, which are not part of the actual reconfiguration module cycle of the system, can be employed. Another alternative is to reconfigure specific logic wrt. the security

violation of a given ReCoFuse, i.e., specific reactions per ReCoFuse become possible. Finally, the raised alarm could also be propagated to the software level which allows a flexible software-controlled adaption against the detected security violation.

Moreover, our flexible architecture allows adding more ReCoFuses (e.g., CRC, additional encryption, hash-based finger printing etc.) easily. The protective measures are dependent on the required degree of protection and can be extended in a flexible manner, since the architecture allows easy integration of new RCFs as presented in this work. Possibly, a full catalog of ReCoFuses can be maintained in the future which allows the augmentation of existing reconfiguration-based designs with ReCoFuses.

Compliance with ethical standards

Conflict of interest On the behalf of all authors, the corresponding author states that there is no conflict of interest.

References

- Abdellatif KM, Chotin-Avot R, Mehrez H (2013) Protecting fpga bitstreams using authenticated encryption. In: IEEE 11th international new circuits and systems conference (NEWCAS), pp 1–4
- Bhasin S, Danger JL, Guilley S, Ngo X, Sauvage L (2013) Hardware trojan horses in cryptographic IP cores. In: Fault diagnosis and tolerance in cryptography (FDTC), pp 15–29
- Biere A, Cimatti A, Clarke EM, Zhu Y (1999) Symbolic model checking without BDDs. In: Tools and algorithms for the construction and analysis of systems, pp 193–207
- Bloem R, Könighofer B, Könighofer R, Wang C (2015) Shield synthesis: runtime enforcement for reactive systems. In: International conference on tools and algorithms for the construction and analysis of systems (TACAS), pp 533–548
- Bradley A (2013) Incremental, inductive model checking. In: International symposium on temporal representation and reasoning, pp 5–6
- Chandrasekharan A, Schmitz K, Kuhne U, Drechsler R (2015) Ensuring safety and reliability of IP-based system design—a container approach. In: International symposium on rapid system prototyping (RSP), pp 76–82
- Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT Press, Cambridge
- Drechsler R, Kühne U (2014) Safe IP integration using container modules. In: 5th international symposium on electronic system design (ISED), pp 1–4
- Een N, Mishchenko A, Brayton R (2011) Efficient implementation of property directed reachability. In: International conference on formal methods in CAD, pp 125–134
- Emmert J, Stroud C, Skaggs B, Abramovici M (2000) Dynamic fault tolerance in FPGAs via partial reconfiguration. In: 8th IEEE international symposium on field-programmable custom computing machines (FCCM), pp 165–174
- Ganai M, Gupta A (2007) SAT-based scalable formal verification solutions (series on integrated circuits and systems). Springer, Berlin
- Guohua W, Dongming L, Fengzhou W, Adetomi A, Arslan T (2017) A tiny and multifunctional icap controller for dynamic partial reconfiguration system. In: NASA/ESA conference on adaptive hardware and systems (AHS), pp 71–76
- Haroldsen T, Nelson B, Hutchings B (2015) Rapidsmith 2: a framework for bel-level cad exploration on Xilinx FPGAs. In: ACM/SIGDA international symposium on field-programmable gate arrays (FPGA), pp 66–69
- Hategekimana F, Whitaker TJ, Pantho MJH, Bobda C (2017) Secure integration of non-trusted ips in SOCs. In: Asian hardware oriented security and trust symposium (AsianHOST), pp 103–108
- Hettwer B, Petersen J, Gehrer S, Neumann H, Güneysu T (2019) Securing cryptographic circuits by exploiting implementation diversity and partial reconfiguration on fpgas. In: Design, automation test in europe conference exhibition (DATE), pp 260–263
- Hori Y, Katashita T, Sakane H, Toda K, Satoh A (2013) Bitstream protection in dynamic partial reconfiguration systems using authenticated encryption. IEICE Trans Inf Syst E 96D(11):2333–2343
- Kashyap H, Chaves R (2016) Compact and on-the-fly secure dynamic reconfiguration for volatile FPGA. ACM Trans Reconfigurable Technol Syst 9(2):1–22
- Kepa K, Morgan F, Kosciuszkiwicz K, Surmacz T (2010) Serecon: a secure reconfiguration controller for self-reconfigurable systems. Int J Crit Comput Based Syst 1(1–3):86–103
- Koch D (2012) Partial reconfiguration on FPGAs: architectures, tools and applications. Springer, Berlin
- Kocher P, Jaffe J, Jun B (1999) Differential power analysis. In: Annual international cryptology conference. Springer, Berlin, pp 388–397
- Lemke-Rust K, Paar C (2006) An adversarial model for fault analysis against low-cost cryptographic devices. In: Fault diagnosis and tolerance in cryptography (FDTC). Springer, pp 131–143
- Li H, Du G, Shao C, Dai L, Xu G, Guo J (2015) Heavy-ion microbeam fault injection into SRAM-based FPGA implementations of cryptographic circuits. IEEE Trans Nucl Sci 62(3):1341–1348
- Mentens N, Gierlichs B, Verbaauwhede I (2008) Power and fault analysis resistance in hardware through dynamic reconfiguration. In: Conference on cryptographic hardware and embedded systems (CHES), pp 346–362
- Nikova S, Rechberger C, Rijmen V (2006) Threshold implementations against side-channel attacks and glitches. In: International conference on information and communications security (ICICS), pp 529–545
- Noguera J, Kennedy IO (2007) Power reduction in network equipment through adaptive partial reconfiguration. In: International conference on field-programmable logic and applications (FPL), pp 240–245
- Onespin (2015) Operational SVA. <https://www.onespin.com/products/dv-verify-apps/operational-sva>. Accessed 8 Mar 2020
- Paulsson K, Hübner M, Bayar S, Becker J (2007) Exploitation of run-time partial reconfiguration for dynamic power management in xilinx spartan III-based systems. In: International symposium on reconfigurable communication-centric systems-on-chip (ReCoSoC), pp 1–6
- Pezzarossa L, Schoeberl M, Spars J (2017) A controller for dynamic partial reconfiguration in FPGA-based real-time systems. In: 20th IEEE international symposium on real-time distributed computing (ISORC), pp 92–100
- Pham KD, Horta E, Koch D (2017) Bitman: A tool and API for FPGA bitstream manipulations. In: Design, automation test in Europe conference exhibition (DATE), pp 894–897
- Schmitz K, Chandrasekharan A, Filho JG, Große D, Drechsler R (2017) Trust is good, control is better: Hardware-based instruction-replacement for reliable processor-IPS. In: 22nd Asia and

- South Pacific design automation conference (ASP-DAC), pp 57–62
31. Schmitz K, Ustaoglu B, Große D, Drechsler R (2019) (ReCo)Fuse your PRC or lose security: finally reliable reconfiguration-based countermeasures on FPGAs. In: International symposium on applied reconfigurable computing, pp 112–126
 32. Schneider T, Moradi A (2016) Leakage assessment methodology. *J Cryptogr Eng* 6(2):85–99
 33. Standard NF (2001) Announcing the advanced encryption standard (AES). *Federal Inf Process Stand Publ* 197(1–51):3–3
 34. Straka M, Kastil J, Kotasek Z (2010) Generic partial dynamic reconfiguration controller for fault tolerant designs based on FPGA. In: IEEE Nordic circuits and systems conference (NORCHIP), pp 1–4
 35. Team O (2018) The international SAT competitions. <http://www.satcompetition.org>. Accessed 13 Apr 2020
 36. Trimberger S, Carberry D, Johnson A, Wong J (1997) A time-multiplexed FPGA. In: IEEE international symposium on field-programmable custom computing machines (FCCM), pp 22–28
 37. Vipin K, Fahmy SA (2014) ZyCAP: Efficient partial reconfiguration management on the xilinx zynq. *IEEE Embed Syst Lett* 6(3):41–44
 38. Xilinx (2007) Linear feedback shift registers in virtex devices. https://www.xilinx.com/support/documentation/application_notes/xapp210.pdf. Accessed 10 Mar 2020
 39. Xilinx (2015) Monitor ip-core product brief. <https://www.xilinx.com/support/documentation/product-briefs/security-monitor-ip-core-product-brief.pdf>. Accessed 5 Jan 2020
 40. Xilinx (2016) Integrated logic analyzer. https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf. Accessed 10 Mar 2020
 41. Xilinx (2018) Partial reconfiguration bitstream monitor v1.0. https://www.xilinx.com/support/documentation/ip_documentation/pr_bitstream_monitor/v1_0/pg304-pr-bitstream-monitor.pdf. Accessed 10 Mar 2020
 42. Xilinx (2018) User guide—7 series fpgas configuration. https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf. Accessed 5 Jan 2020
 43. Xilinx (2018) Xilinx official website—user guide—partial reconfiguration. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf. Accessed 5 Jan 2020
 44. Zhang L, Prasad M, Hsiao M (2004) Incremental deductive inductive reasoning for SAT-based bounded model checking. In: International conference on computer-aided design, pp 502–509
 45. Zhao M, Suh GE (2018) FPGA-based remote power side-channel attacks. In: IEEE symposium on security and privacy (SP), pp 229–244

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.