(ReCo)Fuse Your PRC or Lose Security: Finally Reliable Reconfiguration-based Countermeasures on FPGAs

Kenneth Schmitz*[0000−0001−6618−5907], Buse Ustaoglu*[0000−0002−7469−2260], Daniel Große*†[0000−0002−1490−6175], and Rolf Drechsler*†[0000−0002−9872−1740]

*Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
†Institute of Computer Architecture, University of Bremen, 28359 Bremen
{kenneth.schmitz,buse.ustaoglu}@dfki.de
{grosse,drechsler}@uni-bremen.de

Abstract. Partial reconfiguration is a powerful technique to adapt 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 is still very challenging. This applies in particular to reconfiguration-based countermeasures which are intensively used to form a moving target for the attacker. However, from the system security perspective a critical component is the above mentioned PRC as noticed by many papers implementing reconfiguration-based countermeasures against SCA/DPA attacks. In this work, we leverage a new proposed safety mechanism which creates a container around an IP, to encapsulate and thereby to protect and observe the PRC of an FPGA. The proposed encapsulation scheme results in an architecture consisting of 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 reconfiguration.

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SecRec under grant no. 16K1S0606K, the project SELFIE under grant no. 01IW16001 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

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 strong feature of partial reconfiguration pays off in many application fields today. Practical examples include increasing fault tolerance [1], power-aware reconfiguration [2, 3], and area reduction by time division multiplexing [4].

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) can not 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 have been proposed. Unfortunately, many of these approaches require 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 [5] a "shield" is synthesized which continuously monitors input/output of the design and corrects its erroneous outputs. A more general approach has been proposed in [6, 7]. 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 by 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 also applied on a hardware level by implementing a instruction replacement scheme for a modern RISC-V processor IP to circumvent errata and design flaws [8]. In [9] a similar technique has been proposed. Hardware sand boxes 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 safety mechanism – to the security domain. We present a tailored encapsulation scheme for the PRC. The new architecture consists of individual *ReCoFuses*. Each ***ReCo****nfiguration****Fuse*** (abbreviated as RCF) captures one specific protective property. 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 [10] that introducing temporal jitter based on reconfiguration improves side channel attack resistance significantly. 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). In the case study of this paper, we present two initial ReCoFuses to tackle two major vectors of attacks against the system via the PRC, i.e. to attack

1. the timing of the reconfiguration by keeping one single reconfiguration active for an extended period of time and
2. by disturbing the diversity of individual reconfigurations, such that (in the worst case) the same reconfiguration is used permanently.

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

Related work: The *Partial Reconfiguration Controller* (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 [11], better timing wise predictability during reconfiguration [12] and even fault tolerance [13]. Dedicated protection of the PRC has not been considered in these works.

The authors of [14] 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 bit streams for partial reconfiguration).

Recently, Xilinx announced a security monitor based on a IP soft core which allows monitoring the partial reconfiguration process [15]. To the best of our knowledge, no non-IP-based protection of the PRC for reconfiguration-based countermeasures is offered.

Outline: The paper is structured as follows: First, Section 2 describes the adversary model we consider in this work. In Section 3 the preliminaries of partial reconfiguration and formal verification are reviewed. Our proposed encapsulation scheme for the PRC, implemented as ReCoFuse Container, is introduced in Section 4. Then, Section 5 presents a case study demonstrating the advantages of our scheme for a reconfiguration-based encryption system. The experimental

evaluation, i.e. fault injection and resource utilization, is reported in Section 6. Finally, the paper is concluded in Section 7.

2 Adversary Model

The proposed architecture provides increased protection against attacks targeting reconfiguration-based countermeasures. Adversaries are derived from assumptions made in [10] and [16], 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 [17]) 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 unprotected. 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 ReCoFuses[1].

A second vector of attack is offered from black box IP cores in the design. As motivated in the introduction, malfunctions, flaws or malicious intents can jeopardize the system's stability. Even Trojans in cryptographic hardware blocks were reported in the literature [18]. 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 [19]. 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).

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

3 Preliminaries

In this section we briefly review the basics of partial reconfiguration of FPGAs. Afterwards, an overview on formal verification as used later in order to verify the behavior of the ReCoFuses is provided.

3.1 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 [20], but our approach is also applicable for other FPGA vendors.

Fig. 1 presents the essential components of the partial reconfiguration infrastructure:

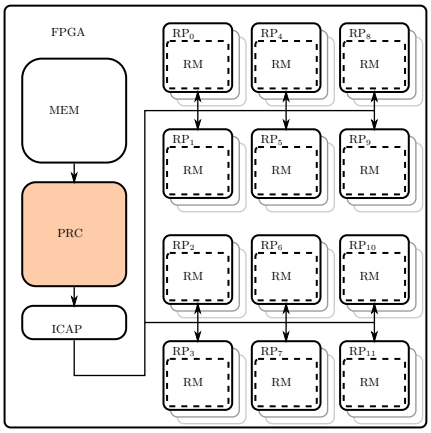

Fig. 1: Overview of partial reconfiguration infrastructure

- *Reconfigurable Partition* (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.

```
1  property check_req_ack;
2    // Assume part          Prove part
3    t ##0 req == 1 implies t ##3 ack == 1);
4  endproperty
```

Listing 1.1: Example property

- 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.

3.2 Formal Verification

Formal verification as used in this work is the task of checking whether a circuit implementation satisfies its specification or not. The specification is thereby expressed with temporal properties. Several standardized property specification languages are available. In this work, we use *SystemVerilog Assertions* (SVA) in combination with *Timing Diagram Assertion Library* (TiDAL) for SVA which comes with the commercial property checking of OneSpin. 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. 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, due to impracticality of full re-verification, our proposed approach still provides a significant advantage.

4 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. 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 of the reconfiguration-based countermeasure.

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.

4.1 Architecture of ReCoFuse Container

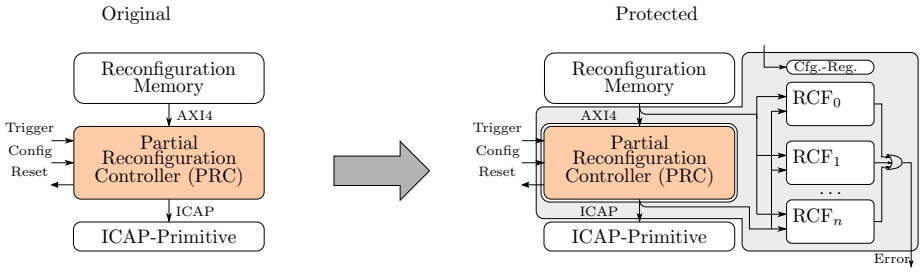

Fig. 2: Original PRC Architecture vs Proposed ReCoFuse Container Architecture

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. As can be seen the ReCoFuse Container has several "slots" for individual ReCoFuses (details see next section). The ReCoFuses are denoted as $\text{RCF}_{0\ldots n}$ 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.

4.2 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 protocol follows a valid/acknowledge scheme where the header of each partial bitfile can be analyzed during data communication. For more details, we refer to the Xilinx 7 Series partial reconfiguration user guide [21].

As can be seen in the architecture, 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. Reaching a predefined "good" or "bad" state determines whether the usage of the PRC is considered as trusted or untrusted. 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

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

Listing 1.2: ReCoFuse Signal Bad State Property

the system. For simplicity, in Fig. 2 on the right we have just ORed all the error signals from each ReCoFuse.

4.3 Verification of ReCoFuses

To guarantee the correctness of each ReCoFuse, we require the formal verification of its behavior. Hence, temporal properties describing the state transitions of a ReCoFuse have to be specified by the user. 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 1.2.

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

5 Case Study

This section demonstrates the proposed encapsulation-scheme for the PRC. As a case study we selected an encryption system using AES. The system implements the moving target principle via reconfiguration by switching between different implementations of the AES. By this, the attacker is not faced with static logic in the FPGA, but permanently changing one and hence physical attacks become much harder.

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

5.1 Attack Vectors

Breaking the reconfiguration-based moving target characteristic of a cryptographic system, allows attackers to extract secret information via side channel leakage. 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 procedure. We identified two major attack vectors on the PRC – viable for both adversaries:

```
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 1.3: Example properties for timer

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:* Forcing the PRC to chose a single reconfiguration continuously (or more often) removes the moving target characteristic as well.

This list, however, is not exhaustive and can must be extended by the respective adopters needs. The next section presents how the proposed ReCoFuse Container helps in protecting against the two attacks.

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 Section 4. ReCoFuses are integrated inside the ReCoFuse Container to achieve countermeasures against the time out and replay attack. The concrete ReCoFuse are presented in the following two sections.

5.3 Timeout ReCoFuse

Functionality: The timeout ReCoFuse (RCF_0) basically keeps track of the time between two consecutive reconfigurations. Hence, after a successful reconfiguration, a timer is started. If this timer expires *before* a new reconfiguration procedure is initiated, the timeout ReCoFuse signals an 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 in combination with the bitfile length could serve for the same purpose.

Verification: In Listing 1.3, 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 1.3). 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 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. 3 shows a reconfiguration procedure over 13 reconfigurations (i.e. steps) in form of a bar chart, choosing from four different RMs. In Fig. 3, 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*. Fig. 3 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 $MFU - LFU = 7 - 0 = 7$[2] and hence an error is signaled by the ReCoFuse.

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

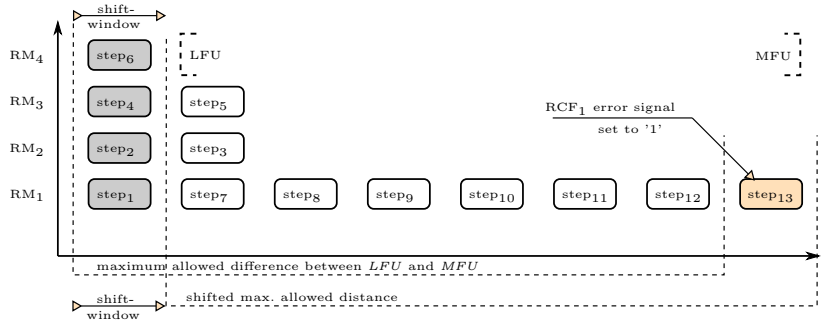

Fig. 3: Uniformity check and shift window operation

```
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
7        and cnt==$past(cnt)-1)
8    );
9  endproperty
10
11 property detect_error;
12   disable iff(rst) (
13     t ##0 (dist==MAX and state==CHECK_ERR)
14       and t ##1 (state==CHECK_ERR)
15   implies
16     t ##1 (error)
17   );
18 endproperty
```

Listing 1.4: Example properties for uniformity check

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 together with its address in the configuration memory. 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.

Verification: In Listing 1.4, a subset of the properties to verify the behavior of the replay ReCoFuse are shown. Please note that the shown 2 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 (`state==SHIFT_WINDOW`) operation 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 1.4 is called `detect_error` starting from Line 11. Following the idea from Listing 1.3, 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).

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

6 Results

All experiments have been conducted on a Xilinx Zync-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 switching between different implementations of the AES core "tiny_AES" from `https://opencores.org/project/tiny_aes` 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 16.04. The partial bitfiles are copied from the SD card to the on board DDR3 memory (serves as partial bitfile 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 Section 4. The two ReCoFuses time out (RCF_0) and replay (RCF_1), as described in Section 5.2, are integrated in the ReCoFuse Container to protect against the two attacks as introduced in Section 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 RCF_0 expires, such that no ReCoFuse 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

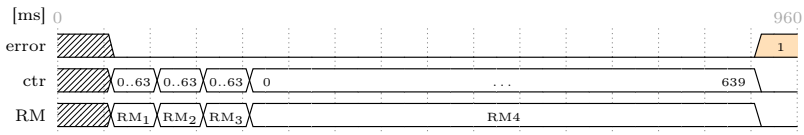

Fig. 4: Behavior of RCF_0 (time-out)

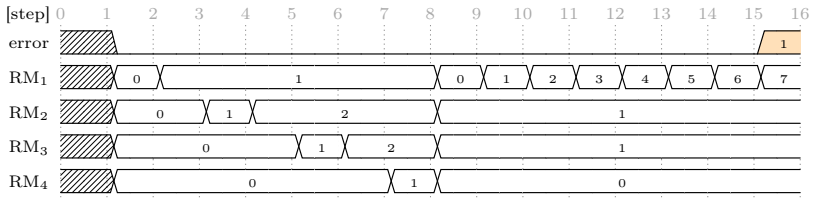

Fig. 5: Behavior of RCF_1 (replay)

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 when using the Xilinx *Integrated Logic Analyzers* (ILAs):

Time-out Attack Fig. 4 illustrates the functionality of RCF_0 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 milliseconds (i.e. 15 time steps). After 64ms, RM_1 is loaded, followed by, RM_2 and RM_3 (each active for 1 time step (64ms)). RM_4 is kept active indefinitely (10 time steps), which exceeds the acceptable period (640ms), 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.

Replay Attack Fig. 5 shows a sequence of the reconfigurations after 15 steps. We have for different RMs. The maximum allowed distance of the RMs is set to 6 as in Fig. 3. 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 15^{th} 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.

In summary, both 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

Table 1 shows the utilization of the FPGA after implementation in Vivado. All synthesis runs and P&R runs were executed with the same settings. The

Table 1: Hardware Resource Utilization

"Moving target AES":	Original	ReCoFuse protected	
Elements	Usage	Usage	Increase
Slices	2976	3036	2.02%
LUT as Logic	10520	10620	0.95%
LUT as Memory	203	203	0.00%
LUT FF Pairs	4367	4409	0.96%

first column *elements* of the table presents the respective resource. The second column *Original* shows the our encryption system employing reconfiguration based on different tiny_AES cores and the Xilinx reconfiguration infrastructure following the moving target principle. Three different AES cores and a blank module (Xilinx recommendation for system initialization) have been included and are randomly chosen for reconfiguration by the *SYSCTRL*. The third column *ReCoFuse protected: Usage* contains the resource utilization for the encryption system protected with the introduced ReCoFuse Container. Finally, the fourth column *ReCoFuse protected: Increase* shows the negligible overhead, caused by our solution.

7 Conclusion and Future Work

In this work, we leveraged an originally proposed safety mechanism which creates a container around an IP, to encapsulate and protect the PRC of an FPGA. 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. 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. Possibly, a full catalog of fuses can be maintained in the future. In summary, this work closes the gap of vulnerable reconfiguration infrastructure as identified in [10] by Mentens et al.

References

1. J. Emmert, C. Stroud, B. Skaggs, and M. Abramovici, "Dynamic fault tolerance in FPGAs via partial reconfiguration," in *FCCM*, 2000, pp. 165–174.
2. K. Paulsson, M. Hübner, S. Bayar, and J. Becker, "Exploitation of run-time partial reconfiguration for dynamic power management in xilinx spartan III-based systems." in *ReCoSoC*, 2007, pp. 1–6.

3. J. Noguera and I. O. Kennedy, "Power reduction in network equipment through adaptive partial reconfiguration," in *FPL*, 2007, pp. 240–245.
4. S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed fpga," in *FCCM*, 1997, pp. 22–28.
5. R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis: - runtime enforcement for reactive systems," in *TACAS*, 2015, pp. 533–548.
6. R. Drechsler and U. Kühne, "Safe ip integration using container modules," in *ISED*, 2014, pp. 1–4.
7. A. Chandrasekharan, K. Schmitz, U. Kühne, and R. Drechsler, "Ensuring safety and reliability of ip-based system design – a container approach," in *RSP*, 2015, pp. 76–82.
8. K. Schmitz, A. Chandrasekharan, J. G. Filho, D. Große, and R. Drechsler, "Trust is good, control is better: Hardware-based instruction-replacement for reliable processor-ips," in *ASP-DAC*, 2017, pp. 57–62.
9. F. Hategekimana, T. J. Whitaker, M. J. H. Pantho, and C. Bobda, "Secure integration of non-trusted ips in socs," in *AsianHOST*, 2017, pp. 103–108.
10. N. Mentens, B. Gierlichs, and I. Verbauwhede, "Power and fault analysis resistance in hardware through dynamic reconfiguration," in *CHES*, 2008, pp. 346–362.
11. K. Vipin and S. A. Fahmy, "ZyCAP: Efficient partial reconfiguration management on the xilinx zynq," *ESL*, vol. 6, no. 3, pp. 41–44, 2014.
12. L. Pezzarossa, M. Schoeberl, and J. Sparsø, "A controller for dynamic partial reconfiguration in FPGA-based real-time systems," in *ISORC*, 2017, pp. 92–100.
13. M. Straka, J. Kastil, and Z. Kotasek, "Generic partial dynamic reconfiguration controller for fault tolerant designs based on FPGA," in *NORCHIP*, 2010, pp. 1–4.
14. K. Kepa, F. Morgan, K. Kosciuszkiewicz, and T. Surmacz, "Serecon: a secure reconfiguration controller for self-reconfigurable systems," *IJCCBS*, vol. 1, no. 1-3, pp. 86–103, 2010.
15. Xilinx, "Monitor ip-core product brief," 2015, https://www.xilinx.com/support/documentation/product-briefs/security-monitor-ip-core-product-brief.pdf.
16. K. Lemke-Rust and C. Paar, "An adversarial model for fault analysis against low-cost cryptographic devices," in *FDTC*. Springer, 2006, pp. 131–143.
17. T. Schneider and A. Moradi, "Leakage assessment methodology," *JCEN*, vol. 6, no. 2, pp. 85–99, 2016.
18. S. Bhasin, J.-L. Danger, S. Guilley, X. Ngo, and L. Sauvage, "Hardware trojan horses in cryptographic IP cores," in *FDTC*, 2013, pp. 15–29.
19. M. Zhao and G. E. Suh, "Fpga-based remote power side-channel attacks," in *S&P*, May 2018, pp. 229–244.
20. Xilinx, "Xilinx official website - user guide – partial reconfiguration," Jan. 2018, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf.
21. ——, "User guide – 7 series fpgas configuration," Mar. 2018, https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.
22. H. Li, G. Du, C. Shao, L. Dai, G. Xu, and J. Guo, "Heavy-ion microbeam fault injection into sram-based fpga implementations of cryptographic circuits," *IEEE Transactions on Nuclear Science*, vol. 62, no. 3, pp. 1341–1348, 2015.