Reconfigurable LUT: A Double Edged Sword for Security-Critical Applications

Debapriya Basu Roy[1], Shivam Bhasin[3], Sylvain Guilley[2,4], Jean-Luc Danger[2,4], Debdeep Mukhopadhyay[1], Xuan Thuy Ngo[2], and Zakaria Najm[2]

[1] Secured Embedded Architecture Laboratoty (SEAL)
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur
{deb.basu.roy,debdeep}@cse.iitkgp.ernet.in
[2] Institut MINES-TELECOM, TELECOM ParisTech, CNRS LTCI (UMR 5141).
{sylvain.guilley,danger,znajm}@enst.fr,
xuan-thuy.ngo@telecom-paristech.fr
[3] Temasek Laboratories, NTU, Singapore.
sbhasin@ntu.edu.sg
[4] Secure-IC S.A.S., 80 avenue des Buttes de Coësmes, 35 700 Rennes, FRANCE

Abstract. Modern FPGAs offer various new features for enhanced reconfigurability and better performance. One of such feature is a dynamically Reconfigurable LUT (RLUT) whose content can be updated internally, even during run-time. There are many scenarios like pattern matching where this feature has been shown to enhance the performance of the system. In this paper, we study RLUT in the context of secure applications. We describe the basic functionality of RLUT and discuss its potential applications for security. Next, we design several case-studies to exploit RLUT feature in security critical scenarios. The exploitation are studied from a perspective of a designer (e.g. designing countermeasures) as well as a hacker (inserting hardware Trojans).

Keywords: Reconfigurable LUT (RLUT), FPGA, CFGLUT5, Hardware Trojans, Side-Channel Countermeasures, Secret Ciphers.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have had a significant impact on the semiconductor market in recent years. FPGAs came into the VLSI industry as successor of programmable read only memories (PROMs) and programmable logic devices (PLDs) and has been highly successful due to its reconfigurable nature. A standard FPGA can be defined as islands of configurable logic blocks (CLBs) in the sea of programmable interconnects. However, with time, FPGAs have become more sophisticated due to the addition of several on-chip features such as high-density block memories, DSP cores, PLLs, etc. These features coupled with their core advantage of reconfigurability and low-time to market have made FPGA an integral part of the semiconductor industry, as

an attractive economic solution for low to medium scale markets like defense, space, automotive, medical, etc. The key parameters for FPGA manufacturers still remain area, performance and power. However, during these recent years, FPGA manufacturers have started considering security as the fourth parameter. Most recent FPGAs support bitstream protection by authentication and encryption schemes [1]. Other security features like tamper resistance, blocking bitstream read-back, temperature/voltage sensing, etc. are also available. FPGA has also been a popular design platform for implementations of cryptographic algorithms due to its reconfigurability and in house security. Apart from the built-in security features, designers can use FPGA primitives and constraints to implement their own designs in a secure manner. In [2], authors show several side-channel countermeasures which could be realized on FPGAs to protect one design. Another work [3] demonstrates the efficient use of block RAMs to implement complex countermeasures like masking and dual-rail logic. DSPs in FPGAs have also been widely used to design public-key cryptographic algorithms like ECC [4, 5] and other post-quantum algorithms [6]. Moreover, papers like [7] have used FPGA constraints like *KEEP, Lock_PINS* or language like *XDL* to design efficient physical countermeasures.

The basic building block of an FPGA is logic slices. Typically a logic slice contains look up tables (LUTs) and flip-flops. LUTs are used to implement combinational logics whereas flip-flops are used to design sequential architectures. Every LUT contains an *INIT* value which is basically the truth table of the combinational function implemented on that LUT. This *INIT* value is set during the programming of the FPGA through bitstream. Generally this *INIT* value is considered to be constant until the FPGA is reprogrammed again. However, in recent years, a new feature has been added to the FPGAs which allows the user to modify the *INIT* value of some special LUTs in the run time, without any FPGA programming. These special LUTs are known as reconfigurable LUTs or RLUTs as they can be reconfigured during the operation phase to change the input-output mapping of the LUT. To the best of our knowledge, RLUTs have found relevant use in pattern matching and filter applications [8]. Side channel protection methodology using RLUT is presented in [9] where the authors have combined different side channel protection strategies with RLUTs and have developed leakage resilient designs. However, in that work the authors have concentrated mainly on constructive use of RLUTs, not on destructive applications which is covered by our paper.

In this paper, we aim to study the impacts and ramifications of these RLUTs on cryptographic implementations. We have provided a detailed study of RLUTs and have deployed it in many security related applications. We propose several industry-relevant applications of RLUT both of constructive and destructive nature. For example, an RLUT can be easily (ab)used by an FPGA IP designer to insert a hardware Trojan. On the other hand, using RLUT, a designer can provide several enhanced features like programming secret data on client-side. The contribution of the paper can be listed as follows:

– This paper provides a detailed analysis of RLUTs and how it can be exploited to create extremely stealthy and serious hardware security threats like hardware Trojans (destructive applications).
– Moreover, we also propose design methodologies which uses RLUTs to redesign efficient and lightweight existing side channel countermeasures to mitigate power based side channel attacks (constructive applications)
– Thus, in this paper we show that how RLUTs provide a gateway of creating efficient designs for both adversary and normal users and act as double-edged swords for security applications. To the best of our knowledge, this is the first study which provides a detailed security analysis of RLUTs from both constructive and destructive points of view.

The rest of the paper is organized as follows: Sec. 2 describes the rationale of an RLUT and discusses its advantages and disadvantages. Thereafter several destructive and constructive applications of RLUT are demonstrated in Sec. 3 and Sec. 4 respectively. Finally conclusions are drawn in Sec. 5.

2 Rationale of the RLUT

RLUT is a feature which is essentially known to be found in *Xilinx* FPGAs. A *Xilinx* RLUT can be inferred into a design by using a primitive cell called *CFGLUT5* from its library. This primitive allows to implement a 5-input LUT with a single output whose configuration can be changed. *CFGLUT5* was first introduced in Virtex-5 and Spartan-6 families of *Xilinx* FPGAs. As we will show later in this section, the working principle of *CFGLUT* is similar to the shift register or the more popularly known *SRL* primitives. Moreover, some older families of Xilinx which do not support *CFGLUT5* as a primitive, can still implement RLUT using the *SRL16* primitive. In the following, for sake of demonstration, we stick to the *CFGLUT5* primitives. Nevertheless the results should directly apply to its alternatives as well.

As stated earlier, a RLUT can be implemented in Virtex-5 FPGAs using a *CFGLUT5* primitive. The basic block diagram of CFGLUT5 is shown in Fig. 1. It is a 5-input and a 1-output LUT. Alternatively, a CFGLUT5 can also be modeled as a 4-input and 2-output function. The main feature of CFGLUT5 is that it can be configured dynamically during the run-time. Every LUT is loaded with a $INIT$ value, which actually represents the truth table of the function implemented on that LUT. A CFGLUT5 allows the user to change the $INIT$ value at the run-time, thus giving the user power of dynamic reconfiguration internally. This reconfiguration is performed using the CD_I port. A 1-bit reconfiguration data input is shifted serially into $INIT$ in each clock cycle if the reconfiguration enable signal (CE) is set high. The previous value of $INIT$ is flushed out serially through the CD_O port, 1-bit per clock cycle. Several CFGLUT5 can be cascaded together using reconfiguration data cascaded output port (CD_O).

The reconfiguration property of CFGLUT5 is illustrated in Fig. 2 with the help of a small example. In this figure, we show how the value of $INIT$ gets modified:

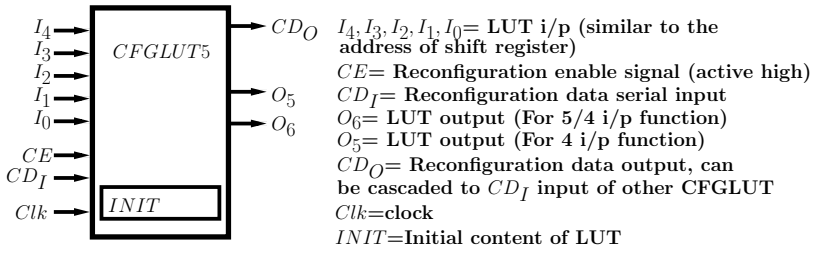

Fig. 1. Block diagram of CFGLUT5

– from value $O = (O_0, O_1, O_2, \ldots, O_{30}, O_{31})$,
– to a new value $N = (N_0, N_1, N_2, \ldots, N_{30}, N_{31})$.

This reconfiguration requires 32 clock cycles. As it is evident from the figure, reconfiguration steps are basic shift register operations. Hence if required, reconfiguration of LUT content can be executed by using shift register primitives ($SRL16E\_1$) in earlier device families. The CD_O pin can also be fed back to the CD_I pin of the same *CFGLUT5*. In this case, the original $INIT$ value can be restored after a maximum of 32 clock cycles without any overhead logic. We will exploit this property of RLUT later to design hardware Trojans.

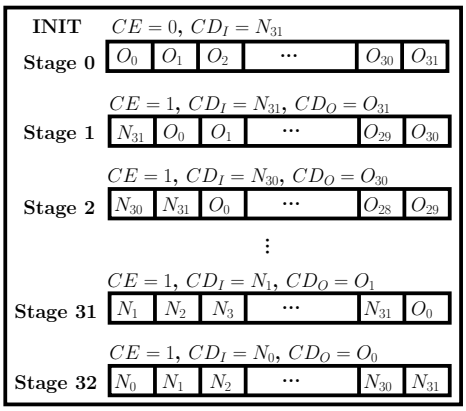

Fig. 2. $INIT$ value reconfiguration in CFGLUT5

There are two different kinds of slices in a Xilinx FPGA i.e., *SLICE_M* and *SLICE_L*. Whereas a simple LUT can be synthesized in either of the slices, CFGLUT5 can be implemented only in *SLICE_M*. *SLICE_M* contains LUTs which can be configured as memory elements like shift register, distributed memory along with combinational logic function implementation. The LUTs of *SLICE_L* can only implement combinational logic. CFGLUT5, when instantiated, is essen-

tially mapped into a *SLICE_M*, configured as shift register ($SRL32$) as shown in Fig 3.

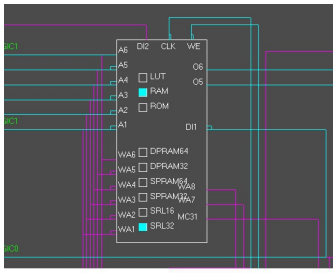

Fig. 3. CFGLUT5 mapped in LUT as SRL32 as shown from Xilinx FPGA Editor

2.1 Comparison With Dynamic Configuration

Another alternative to reconfigure FPGA in run-time is to use partial or dynamic reconfiguration. This reconfiguration can also be exploited to implement secure architectures [10]. In partial reconfiguration, a portion of the implemented design is changed without disrupting operations of the other portion of the FPGA. This operation deploys an Internal Configuration Access Ports (ICAP) and the design needing reconfiguration must be mapped into a special *reconfigurable region* [11]. Reconfiguration latency is in order of milliseconds. Partial reconfiguration is helpful when significant modification of the design is required. However, for small modification, using RLUT is advantageous as it has very small latency (maximum 32 clock cycles) compared to partial reconfiguration. RLUT is configured internally and no external access to either JTAG or Ethernet ports are required for reconfiguring RLUTs. Additionally, traditional DPR (Dynamic Partial Reconfiguration) requires to convey an extra bit file which is not required in case of RLUT, making RLUT ideal for small reconfiguration of the design, in particular for Trojans.

2.2 RLUT and Security

Since we have described the functioning of RLUT in detail, we can clearly recognize some properties which could be helpful or critical for security. A typical problem of cryptographic implementations is its vulnerability to statistical attacks like Correlation Power Analysis (CPA) [12]. For instance, CPA tries to extract secret information from static cryptographic implementations by correlating side-channel leakages to estimated leakage models. A desirable feature to protect such implementations is reconfiguration of few internal features. A RLUT would be a great solution in this case as it has the power to provide reconfigurability at minimal overhead and with no external access. It is important to reconfigure internally to avoid the risk of any eavesdropping. On the other

hand, RLUT can also be used as a security pitfall. For example, an efficient designer can simply replace a LUT with RLUT in a design keeping the same *INIT* value. Until reconfiguration, RLUT would compute normally. However upon reconfiguration, the RLUT can be turned into a potential Trojan. The routing of the design is actually static, only the functionality of the LUT is modified upon reconfiguration. In the following sections, we would show some relevant applications of constructive or deadly nature. Of course it is only a non-exhaustive list of RLUT applications into security.

3 Destructive Applications of RLUT

In earlier sections, we have presented the basic concepts of RLUTs with major emphasis on *CFGLUT5* of Xilinx FPGAs. Though *CFGLUT5* provides user unique opportunity of reconfiguring and modifying the design in run-time, it also gives an adversary an excellent option to design efficient and stealthy hardware Trojan. In this section, we focus on designing tiny but effective hardware Trojan exploiting reconfigurability of RLUTs.

A hardware Trojan is a malevolent modification of a design, intended for either disrupting the algorithm operation or leaking secret information from it. The design of hardware Trojan involves efficient design of Trojan circuitry (known as payload) and design of trigger circuitry to activate the Trojan operation. A stealthy hardware Trojan should have negligible overhead, ideally zero, compared to the original *golden* circuit. Moreover, probability of Trojan getting triggered during the functional testing should be very low, preventing accidental discovery of the Trojan. The threat of hardware Trojans is very realistic due to the fabless model followed by the modern semiconductor companies. In this model, the design is sent to remote fabrication laboratories for chip fabrication. It is very easy for an adversary to make some small modification in the design without violating the functionality of the design. The affected chip will give desired output in normal condition, but will leak sensitive information upon being triggered. More detailed analysis of hardware Trojans can be found in [13–15].

Researchers have shown that it is possible to design efficient hardware Trojans on FPGAs. In [16] the authors have designed a Trojan on a Basys FPGA board which get triggered depending upon the 'content and timing' of the signals. On the other hand, authors in [17] have designed a hardware Trojan which can be deployed on the FPGA via dynamic partial reconfiguration to induce faults in an AES circuitry for differential fault analysis.

In this section, we will focus on effective design of hardware Trojan payload using RLUT. But before going into the design methodologies of payload using RLUTs, we will first describe the other two important aspects of the proposed hardware Trojans: Adversary model and Trigger methodologies.

3.1 Adversary Model

It is a common trend in the semiconductor industry to acquire proven IPs to reduce time to market and stay competitive. We consider an adversary model where a user buys specific proven IPs from a third party IP vendor. By proven IPs, we mean IPs with well-established performance and area figures. Let us consider that the IP under consideration is a cryptographic algorithm and the

target device is an FPGA. An untrusted vendor can easily insert a Trojan in the IP which can act as a backdoor to access sensitive information of other components of the user circuit. For instance, an IP vendor can provide a user with an obfuscated or even encrypted netlist (encrypted *EDIF* (Electronic Design Interchange Format)). Such techniques are popular and often used to protect the rights of the IP vendor. A Trojan in an IP is very serious for two major reasons. First, the Trojan will affect all the samples of the final product and secondly it is almost impossible to get a golden model. Moreover, research in Trojan detection under the given attack model is quite limited. The user does not have a golden circuit to compare, thus making hardware Trojan detection using side channel methodology highly unlikely. Additionally, this adversary model also makes the Trojan design challenging. Generally, before buying an IP, user will analyze IPs from different IP vendors for performance comparison. This competitive scenario does not leave a big margin (gate-count) for Trojans.

Using RLUT, we can design extremely lightweight hardware Trojan payload as we can reconfigure the same LUTs, used in the crypto-algorithm implementation, from correct value to malicious value. This reduces the overhead of the hardware Trojan and makes it less susceptible to detection techniques based on visual inspection [18]. We can also restore the original value of RLUT to remove any trace of Trojan, of course, at minor overheads. An IP designer can easily replace a normal LUT with RLUT. In this case, the designer has only one restriction of replacing a LUT implemented in *SLICE_M*. It is not difficult to find such a LUT in a medium to big-scale FPGA which is often the case with cryptographic modules. Moreover, if the designer chooses to insert the trojan at RTL level, the present restriction would not even apply. Additionally, if the access to the client bitstream is available, the adversary can reverse engineer the bitstream [19] and can replace a normal LUT with RLUT.

Instantiation of *CFGLUT5* does not report any special element in the design summary report, **but a LUT modeled as *SRL32***. A shift register has many usages on the circuit. For example, a counter can be very efficiently designed on a shift register using one hot encoding. Moreover, lightweight ciphers employs extensive usage of shift registers for serialized architectures. Thus any suspicion of malicious activity will not arise in the user's mind by seeing the design report.

The only requirement is efficient triggering and a reconfiguration logic which will generate the malicious value upon receiving trigger signal. However, in this paper we will show that once triggered, **malicious value for the hardware Trojan can be generated without any overhead**, thus giving us extremely lightweight and stealthy design of hardware Trojans. The basic methodology is same for all the Trojans, which can be tabulated as follows:

- Choose a sensitive sub-module of the crypto-algorithm. For example, one can choose a 4×4 Sbox (can be implemented using 2 LUTs) as the sensitive sub module.
- Replace the LUTs of the chosen sub-module with *CFGLUT5s* without altering the functionality. A 4×4 Sbox can also be implemented using two *CFGLUT5*.

– Modify the *INIT* value upon trigger. As shown in Fig. 1, reconfiguration in *CFGLUT5* takes place upon receiving the *CE* signal. By connecting the trigger output to the *CE* port, an adversary can tweak the *INIT* value of *CFGLUT5* and can change it to a malicious value. For example, the 4×4 Sboxes, implemented using *CFGLUT5* can be modified in such a way that non-linear properties of the Sboxes get lost and the crypto-system becomes vulnerable to standard cryptanalysis. The malicious *INIT* value can be easily generated by some nominal extra logic. However, in the subsequent sections, we will show that it is possible to generate the malicious *INIT* value without any extra logic.
– Upon exploitation, restore original *INIT* value.

3.2 Trigger Design the Hardware Trojans

A trigger for a hardware Trojan is designed in a way that the Trojan gets activated in very rare cases. The trigger stimulus can be generated either through output of a sensor under physical stress or some well controlled internal logic. The complexity of trigger circuit also depends on the needed precision of the trigger in time and space. Several innovative and efficient methods were introduced as a part of Embedded Systems Challenge (2008) where participants were asked to insert Trojans on FPGA designs. For instance, one of the the proposition was *content & timing* trigger [16], which activates with a correct combination of input and time. Such triggers are considered practically impossible to simulate. Other triggers get activated at a specific input pattern. A more detailed analysis with example of different triggering methodologies and their pros and cons can be found in [20].

Moreover, modern devices are loaded with physical sensors to ensure correct operating conditions. It is not difficult to find voltage or temperature sensors in smart-cards or micro-controllers. Similarly, FPGA also come with monitors to protect the system for undesired environmental conditions, Virtex-5 FPGAs contain *system monitor*. Though system monitor is not a part of cipher, they are often included in the SoC for tamper/fault/ temperature variation detection. These sensors are programmed to raise an alarm in event of unexpected physical conditions like overheating, high/low voltage etc. Now an adversary can use this system monitor to design an efficient and stealthy hardware Trojan trigger methodology. The trick is to choose a trigger condition which is less than threshold value but much higher than nominal conditions. For instance, a chip with nominal temperature of $20°\text{C} - 30°\text{C}$ and safety threshold of $80°\text{C}$, can be triggered in a small window chosen from the range of $40°\text{C} - 79°\text{C}$. Similarly, user deployed sensors like the one proposed in [21] can also be used to trigger a Trojan. In our case study, we used the temperature sensor of Virtex-5 FPGAs *system monitor* to trigger the Trojan, more precisely on SASEBO-GII boards. The heating required to trigger the Trojan can be done by a simple $5 hair-dryer easily available in the market. The triggering mechanism is explained in Appendix A. In the following to not deviate from the topic, we focus mainly on the payload design of the Trojan using RLUT. We let the designer choose any

of the published techniques (including one proposed in Appendix A) or innovate one. We precisely propose the design of the Trojan and the required triggering conditions.

3.3 Trojan Description

Before designing Trojan payload for a given hardware, we first demonstrate the potential of RLUT in inserting malicious activity. Let us consider a buffer which is a very basic gate. Buffers are often inserted in a circuit by CAD tools to achieve desired timing requirements. For FPGA designers, another equivalent of buffer is route-only LUT. These buffers can be inserted in any sensitive wires without raising an alarm. In fact, sometimes the buffers might already exist.

These buffers are implemented in a *LUT6* with `INIT=0xAAAAAAAAAAAAAAAA` and can be easily replaced by *CFGLUT5*. A simple Trojan would consist in changing the *INIT* value of CFGLUT5 to `0xAAAAAAAA` and feedback CD_O output to CD_I input (see Fig 1). The CE input is connected to the trigger of the Trojan. Now, when the Trojan is triggered once (one clock), *INIT* value changes to `0x55555555` which changes the functionality of the gate to **inverter**. Another trigger brings back the *INIT* value to `0xAAAAAAAA` i.e., a **buffer**. The operations are illustrated in Fig. 4, where red block shows Trojan inverter and black blocks show a normal buffer. Thus by precisely controlling the trigger, an adversary can interchange between a buffer and inverter. Such a Trojan can be used in many scenarios like injecting single bit faults for Differential Fault Attacks [22] or controlling data multiplexers or misreading status flags, etc.

In the above example, we see how a buffer can be converted to an inverter by reconfiguring the *CFGLUT5* upon receiving the trigger signal. One important observation is that we do need need any extra reconfiguration logic to modify the *INIT* value of the *CFGLUT5*. The modification of the *INIT* value is achieved by the connecting the reconfiguration input port CD_I to the reconfiguration data output port CD_O. In other words, we can define the malicious *INIT* value in following way

$$INIT_{malicious} = CS_i(INIT_{normal})$$

where CS_i denotes cyclic right shift by i bits. The approach of RLUT is harder to detect because the malicious payload does not exist in the design. It is configured when needed and immediately removed upon exploitation. In normal LUT, the malicious design is hardwired (requires extra logic) and risk detection, whereas RLUT modifies existing resources and enables us to design design hardware Trojans without any extra reconfiguration logic. We will use similar methodologies for all the proposed hardware Trojans in this paper.

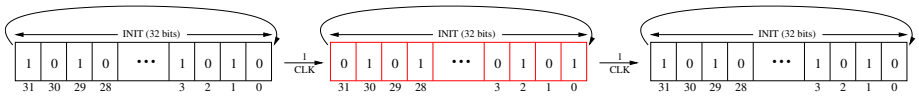

Fig. 4. Operations of CFGLUT5 to switch from a buffer to inverter and back

Next, we target a basic *AES* IP as a Trojan target. The architecture of the AES design is shown in Fig. 5. The AES takes 128 bits of plaintext and key as input and produce 128 bit cipher-text in 11 clock cycles. The control unit of the AES encryption engine is governed by a 4 bit mod-12 counter and generates three different control signals which are as follows:

1. *load:* It is used to switch between plaintext and MixColumns output. During the start of the encryption, this signal is made high to load the plaintext in the AES encryption engine.
2. *S.R/M.C:* It is used to switch between the ShiftRows and MixColumns output in the last round of AES.
3. *done:* It is used to indicate the end of encryption.

These signals are set high for different values of the counter. In our Trojan design, we mainly target the control unit of the AES architecture to disrupt the flow of the encryption scheme so that we can retrieve the AES encryption key. For this, we have developed four different Trojans and have deployed them on the *AES* implementation. The objective of the developed Trojan is to retrieve the *AES* key with only one execution of hardware Trojan or single bad encryption. Indeed, it has been shown that only one faulty encryption, if it is accurate in time, suffices to extract a full 128-bit key [23]. Triggering conditions can be further relaxed if several bad encryptions are acceptable. Each Trojan has trigger with different pulse-width or number of clock cycles. For different payloads, the RLUT content varies, hence variation in the trigger.

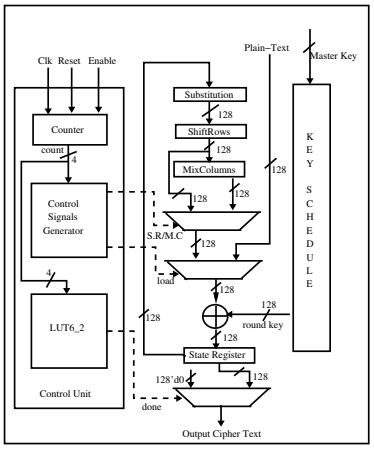

Fig. 5. AES architecture without Trojans

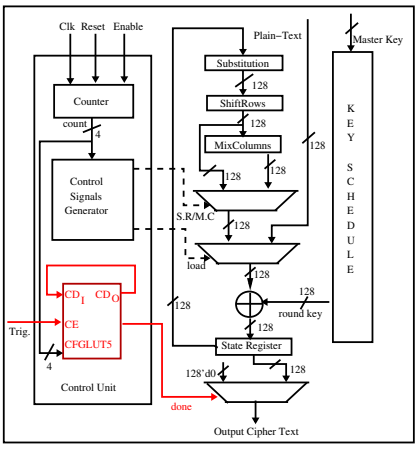

Fig. 6. AES architecture with Trojan 1

The detailed description of the developed Trojans are as follows:

Trojan 1 As we have stated earlier, the control unit of *AES* is based on a counter which also generates a *done* flag to indicate completion of the encryption

cycle and is set to high only if counter value reaches 11. Signal *done* as shown in Fig. 5, is driven by a LUT6_2, which takes 4 bit counter value as input, and under normal operation it should contain *INIT* value `0x00000800` (it means only the 11^{th} bit is set to one i.e. condition required for *done* signal). To insert a Trojan we replace this LUT with *CFGLUT5* with `INIT=0x80000800`. It must be noted that though the *INIT* value of LUT6_2 and *INIT* value of CFGLUT5 are not same, both will essentially produce the same output upon receiving the 4 bit counter value. This is because truth-table of a function of 4 variables consists of 16 bits only, hence any change in the upper 16 bits of the *INIT* value will not change the functionalities of the LUT.

The CD_O output is feedback into CD_I input as in the example above. A trigger of 2 clock cycles at the CE input activates the Trojan (`INIT=0x00002002`) and produces the round 0 output (at round 0, counter value is 1) as the ciphertext. The round 0 output is actually same as *plaintext XOR key* and by knowing the plaintext, one can easily extract the full key with one wrong encryption. Again, we can see that malicious value of the *INIT* is generated by cyclic shift of the original *INIT* value of the *CFGLUT5*, hence we do not need any extra logic to generate the new *INIT* value. After extracting the key, a trigger of 10 clock cycles will restore the normal operations of the AES (`INIT=0x00800800`). This *INIT* value need not to be the same value, with which we started the computation (`INIT=0x80000800`), as long as the LUT generates correct output. The transition of *INIT* to activate the Trojan and restore back is shown in Fig 7(a) and the modifications in the AES architecture is shown in Fig. 6.

In the above Trojan description, we need 2 clock cycles to modify the *CFGLUT5* to malicious Trojan configuration and 10 clock cycles to restore it to the original correct value. So in total, we require 12 clock cycles.

Keeping this in mind, we have implemented three different versions of the same Trojan, depending on the precision of the trigger.

1. **Trojan 1a** needs a 1 cycle trigger synchronized with the start of the encryption. This trigger is used to enable a FSM which generates 12 clock cycles for CE of the $CFGLUT$, in order to activate the Trojan and restore it back after exploitation. Because of this, the overhead of the developed Trojan is 6 **LUTs and** 4 **flip-flops**.
2. **Trojan 1b** is a **zero** overhead Trojan. It assumes an adversary to be slightly stronger than Trojan 1a who can generate a trigger signal active for precisely 12 cycles and synchronized with the start of encryption.This overhead is absent in Trojan 1b as the trigger itself act as the CE signal of RLUT.
3. **Trojan 1c** relaxes the restriction on the adversary seen at previous case. It assumes that there are some delays of $n \gg 10$ clock cycles between two consecutive encryption. The choice of $n \gg 10$ is due to the fact that we need 2 clock cycles to reconfigure the RLUT into malicious Trojan payload, and 10 clock cycles to restore it back to good value. Hence the gap between two consecutive AES encryption should be greater than 10. The adversary provides a trigger of two clock cycles (not necessarily consecutive) before the start of current encryption. After the faulty encryption is complete, the

adversary generates 10 trigger cycles (again not necessarily consecutive) to restore back the cipher operations. The overhead for this Trojan is 2 **LUTs** , due to routing of RLUT.

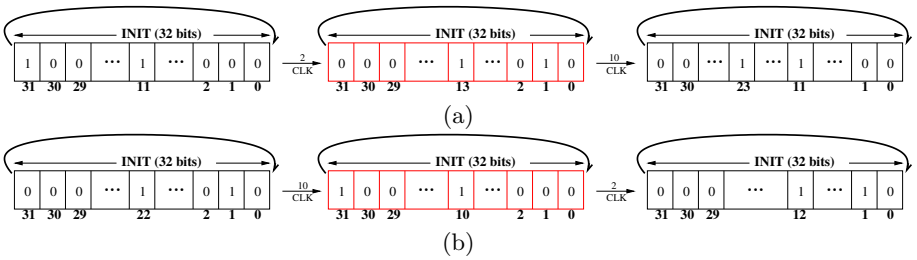

Fig. 7. Operations of CFGLUT5 to activate the Trojan and restore to normal operations for (a) Trojan 1; (b) Trojan 2. Bit positions not shown contain '0'

Trojan 2 This Trojan targets a different signal in the control unit of the *AES* design. As shown in Fig. 5, the design contains a multiplexer which switches between MixColumns output and input plaintext depending on the round/count value. The output of the multiplexer is produced at input of AddRoundKey operation. Under normal operation, multiplexer passes the input plaintext in round 0 (*load* signal of multiplexer is set to 1) and MixColumns output (ShiftRows output in the last round) in other rounds (*select* signal of multiplexer is set to 0). To design the Trojan, we have replaced the LUT6_2 (with `INIT=0x00000002`) which generates *load* signal of the multiplexer with *CFGLUT5*, containing `INIT=0x00400002`. As we have observed for Trojan 1, the difference in the *INIT* value in LUT6_2 and *INIT* value of *CFGLUT5* will essentially produce the same output.

In this case also CD_O port of *CFGLUT5* is connected to CD_I port, enabling cyclic shift of the *INIT* value. Upon a trigger of 10 clock cycles, the *INIT* value gets modified to `INIT=0x80000400` (it means load will set to one during the last round). This actually changes the multiplexer operation, modifying it to select the plaintext in the last round computation. From the resulting ciphertext of this faulted encryption, we can easily obtain the last round key, given the plaintext. Further a trigger of 2 clock cycles restores the normal operation (`INIT=0x00001002`) as shown in Fig 7(b). Again the value over bit position 12 is not a problem as the *select* signal is controlled by a mod-12 counter and the value is never reached. The counter value 0 indicates idle state, $1-10$ encryption and 11 indicates end of encryption. This Trojan also has a **zero** overhead as reconfiguration of the CFGLUT5 is obtained by cyclic right shifting of *INIT*. But the trigger signal need to be precise and should be available for consecutive 12 clock cycles. Hence, triggering cost is same as Trojan 1b.

Hence, the triggering cost is same as Trojan 1b.

Tab. 1 summarizes the nature, trigger condition and cost of the four Trojans.

Table 1. Area overhead of the Trojans on Virtex-5 FPGA. Trigger is given in clock cycles and s subscript indicates trigger must be consecutive synchronized with the start of encryption.

Trojan	Trigger	LUT	Registers	Payload Overhead	Frequency (MHz)
AES (No Trojan)		1594	260	X	212.85
Trojan 1a	1_s	1600	264	6 LUTs & 4 flip-flops	212.85
Trojan 1b	12_s	1594	260	0	212.85
Trojan 1c	12	1596	260	2 LUTs	212.85
Trojan 2	12_s	1594	260	0	212.85

The above described Trojans can also be designed using normal LUTs. The zero overhead Trojans described above can be designed using 2 LUT overhead (One LUT for Trojan operation and other for selecting between Trojan and normal operations). But such Trojan designs can be easy to detect as the Trojan operated LUT is always present on the design unlike CFGLUT5, where the Trojan operated LUT is created by run time reconfiguration.

In this section, we have presented different scenarios where CFGLUT5 can be employed as hardware Trojans and can leak secret information from crypto-IPs like AES. We specifically have targeted multiplexers and FSMs of the circuit. It is also possible to design sophisticated Trojans using CFGLUT5 where the developed Trojan will work in conjunction with side channel attacks or fault injections to increase the vulnerability of the underlying crypto-system.

4 Constructive Applications for RLUT

In the previous section, we discussed some application of RLUT for hardware Trojans into third party IPs. However, RLUT do have a brighter side to their portfolio. The easy and internal reconfigurability of RLUT can surely be well exploited by the designers to solve certain design issues. In the following, we detail two distinct cases with several applications, where RLUT can be put to good use.

4.1 Customizable Sboxes

A common requirement in several industrial application is dynamic or cutomizable substitution boxes (Sboxes) of a cipher. One such scenario which is often encountered by IP designers who design **secret ciphers** for industrial application. A majority of secret ciphers use a standard algorithm like AES with modified specification like custom Sboxes or linear operations. Sometimes the client is not comfortable to disclose these custom specifications to the IP designer. Common solutions either have a time-space overhead or resort to dynamic reconfiguration, to allow the client to program secret parameters at their facilities. A RLUT can come handy in this case.

There are several algorithms where the Sboxes can be secret. The former Soviet encryption algorithm GOST 28147-89 which was standardized by the Russian standardization agency in 1989 is a prominent example [24]. The A3/A8 GSM algorithm for European mobile telecommunications is another example. In the field of digital rights management, Cryptomeria cipher (C2) has a secret set of Sboxes which are generated and distributed to licencees only.

There are certain encryption schemes like DRECON [25], which offers DPA resistance by construction, exploiting tweakable ciphers. In this scheme, users exchange a set of tweak during the key exchange. The tweak is used to choose the set of Sboxes from a bigger pool of precomputed Sboxes. In the proposed implementation [25], the entire pool of Sboxes must be stored on-chip. Using RLUT, the Sboxes can be easily computed as a function of the tweak and stored on the fly. Similarly, a low-cost masking scheme RSM [3] can also benefit from RLUT to achieve desired rotation albeit at the cost of latency. Thus there exist several applications where customizable Sboxes are needed.

Architecture of Sbox Generator: As a proof of concept, we implement the Sbox generation scheme of [25]. The original implementation generates a pool of 32 4×4 Sboxes and stores it into BRAMs, while only 16 are used for a given encryption. It uses a set of Sboxes which are affine transformations of each other. For a given cryptographically strong Sbox $S(\cdot)$, one can generate 2^n strong Sboxes by following: $\mathsf{F_i(x)} = \alpha\mathsf{S(x)} \oplus \mathsf{i}$ for all $i = 0, \cdots, 2^n - 1$, where α is an invertible matrix of dimension $n \times n$. α can also be considered a function of the tweak value t i.e. $\alpha = f(t)$. Since affine transformation does not change most of the cryptographic properties of Sboxes, all the generated Sboxes are of equal cryptographic strength [25].

The Sbox computation scheme of [25] can be very well implemented using RLUT as follows. The **main objective** of this Sbox generator is to compute a new affine Sbox from a given reference Sbox, and store it in the same location. The architecture is shown in Fig 8. As we have stated earlier, each *CFGLUT5* can be modeled as 2 output 4 input function generator, we can implement a 4×4 Sbox using two *CFGLUT5* as shown in Fig 8. We consider that the reference 4×4 Sbox is implemented using 2 *CFGLUT5*. We compute the new Sbox and program it in the same 2 *CFGLUT5*. The reconfiguration of the Sbox is carried through following steps:

1. Read the value of the Sbox for input 15.
2. Compute the new value (4-bits {3,2,1,0}) of the Sbox using affine transformer for the Sbox input 15.
3. Now *CFGLUT5* is updated by the computed value, 2 bits for each *CFGLUT5* ({3,2},{1,0}). However, only one bit can be shifted in *CFGLUT5* in one clock cycle. Hence we shift in two bits, 1-bit in each CFGLUT5 ({0,2}) and store the other 2-bit ({1,3}) in two 16 bit registers.
4. After the 2-bits ({0,2}) of new value of Sbox is shifted in to position 0 of each *CFGLUT5*, old value for the position 15 is flushed out. The old value at position 14 is moved up to position 15. Thus the address is hard-coded to `4'd15`.
5. Repeat steps $1 - 4$ until whole old Sbox is read out i.e. 16 clock cycles.
6. After 16 clock cycles, we start to shift in the data which we stored in the shift register bits ({1,3}) for 16 Sbox entries, which takes another 16 clock cycles. This completes Sbox reconfiguration.

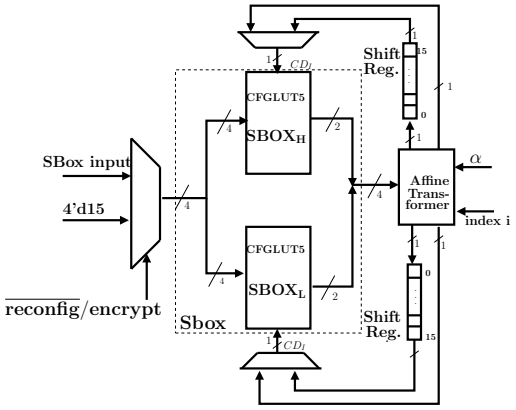

Fig. 8. Architecture of Sbox Computation using affine transformation and storing in RLUT

The architecture requires **56 LUTs, 38 flip-flops with a maximum operating frequency of 271 MHz.** To reconfigure one Sbox, we need 32 clock cycles. Now depending on the application and desired security the sbox recomputation can be done after several encryption or every encryption or every round. It is a purely security-performance trade-off.

4.2 Sbox Scrambling for DPA Resistance

RLUT also have the potential to provide side-channel resistance. The reconfiguration provided by RLUT can be very well used to confuse the attackers. A beneficial target would be the much studied masking countermeasures [2] which suffer from high overhead due to the requirement of *regular mask refresh*. One of the masking countermeasures which was fine-tuned for FPGA implementation is Block Memory content Scrambling (BMS [2]). This scheme claims first-order security and, to our knowledge, no practical attack has been published against it. However, Sbox Scrambling using BRAM is inefficient on lightweight ciphers with 4x4 sboxes due to underutilization of resources. Hence we propose a novel architecture using RLUT to address this. Nevertheless, this mechanism can easily be translated to AES also.

The side channel countermeasure using RLUT, shown in [9] is different from the proposed design architecture. The design of [9] implements standard Boolean masking scheme, where each round uses a different mask. Here, we propose a lightweight architecture of SBox scrambling scheme presented in [2]. These two countermeasures have similar objectives but quite different designs.

The BMS scheme works as follows: let $Y(X) = P(SL(X))$ be a round of block cipher, where X is the data, $P(\cdot)$ is the linear and $SL(\cdot)$ is the non-linear layer of the block cipher. For example in PRESENT cipher [26], the non-linear layer is composed of 16 4×4 Sboxes and the linear layer is bit-permutation. According to the BMS scheme, the masked round can be written as $Y_M(X) = P(SL_M(X_M))$, where X_M is masked data $X \oplus M$ and $SL_M(\cdot)$ is the Sbox layer

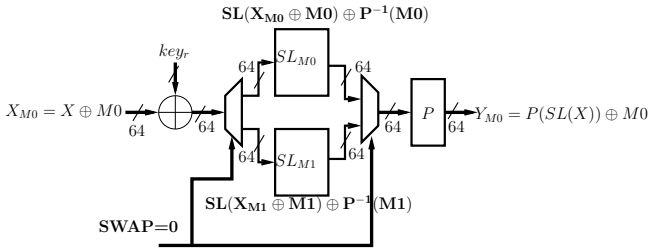

Fig. 9. Architecture of Modified PRESENT Round. SL_{M0} is the (precomputed) active SLayer while SL_{M1} is being computed as in Fig. 10.

of 16 scrambled Sbox. Now each Sbox $S_m(\cdot)$ in SL_M is scrambled with one nibble m of the 64-bit mask M. The scrambled Sbox $S_m(\cdot)$ can be simplified as $S_m(x_m)) = S(x_m \oplus m) \oplus P^{-1}(m)$, where x is one nibble of round input X. Next in a dual-port BRAM which is divided into an active and inactive segment, where the active segment contains $SL_{M0}(\cdot)$ i.e. Sbox scrambled with mask $M0$ is used for encryptions. Parallely, another Sbox layer $SL_{M1}(\cdot)$ scrambled with mask $M1$ is computed in an encryption-independent process and stored in the inactive segment. Every few encryption, the active and inactive contents are swapped and a new Sbox scrambled with a fresh mask is computed and stored in the current inactive segment. This functioning is illustrated in Fig. 9.

BMS is an efficient countermeasure and shown to have reasonable overhead of 44% for LUTs, 2× BRAMs and roughly 3× extra flip-flops in FPGA. Another advantage of BMS is that it is generic i.e., it can be applied to any cryptographic algorithm. BMS can be viewed as a *leakage resilient* implementation, where the cipher is not called enough with a fixed mask for an attack to succeed. The memory contexts are swapped again with a fresh mask. However, for certain algorithms BMS could become unattractive. For example in a lightweight algorithm like PRESENT, a 4×4 Sbox can be easily implemented in 4 LUTs. In newer FPGA families which support 2-output LUT, 2 LUTs are enough to implement a Sbox. Using a BRAM in such a scenario would lead to huge wastage of resources.

Sbox Scrambling using RLUT: In the following, we use RLUT to implement BMS like countermeasure. Precisely we design a PRESENT cryptoprocessor protected with a BMS like scrambling scheme but using RLUTs to store scrambled Sboxes. The rest of the scheme is left same as [2]. The architecture of Sbox scrambler using RLUT is shown in Fig 10. $SBOX_P$ is the PRESENT Sbox. A mod16 counter generates the Sbox address $ADDR$ which is masked with Mask m of 4-bits. The output of Sbox is scrambled with the inverse permutation of the mask to scramble the Sbox value. Please note that the permutation must be applied on the whole 64-bits of the mask to get 4-bits of the scrambling constant for each Sbox. Each output of the scrambler is 4-bits. As stated before, each 4×4 Sbox can be implemented in 2 CFGLUT5 each producing 2-bits of the Sbox com-

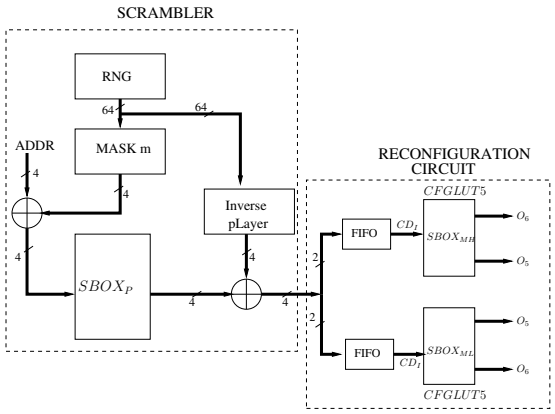

Fig. 10. Architecture of Sbox Scrambler

Table 2. Area and Performance Overhead of Scrambling Scheme on Virtex-5 FPGA

Architecture	LUTs	Flip-flops	Frequency (MHz)
Original	208	150	196
Scrambled	557	552	189
Overhead	2.67×	3.68×	1.03×

putation. Let us call the CFGLUT5 producing bits $0, 1$ as $SBOX_{ML}$ and bits $2, 3$ as $SBOX_{MH}$. The 4-bit output of the scrambler is split into two buses of 2-bits ({3,2},{1,0}). Bits {3,2} and {1,0} are then fed to the CD_I of $SBOX_{ML}$ and $SBOX_{MH}$ respectively, through a FIFO. The same scrambler is used to generate all the 16 Sboxes one after the other and program CFGLUT5. In total it requires 16×32 clock cycles to refresh all 16 inactive Sboxes. We implement two parallel layers of SBoxes. When the active layer is computing the cipher, the inactive one is being refreshed. Thus cipher operation is not stalled. 16×32 clocks (16 encryptions) are needed to refresh the inactive layer and this means that we can swap active and inactive SBoxes after every 16 encryptions. Swap means that active SBox become inactive and vice versa. The cipher design uses active SBox only. The area overhead comes from the scrambler circuit and multiplexers used to swap active/inactive Sboxes. We implemented a PRESENT crypto-processor and protected it with Sbox scrambling countermeasure. The area and performance figures of the original design and its protected version are summarized in Tab. 2. It should be noted that proposed design has more overhead compared to original BMS scheme in terms of LUT and flip-flops, but does not require any block RAMs which are essential part of original BMS scheme.

5 Conclusions

This paper addresses methods to exploit reconfigurable LUTs (RLUTs) in FPGAs for secure applications, with both views: destructive and constructive. First it has been shown that the RLUT can be used by an attacker to create Hardware Trojans. Indeed the payload of stealthy Trojans can be inserted easily in

IP by untrusted vendors. The Trojans can be used to inject faults or modify the control signals in order to facilitate the key extraction. This is illustrated by a few examples of Trojans in AES. Second the protective property of RLUT has been illustrated by increasing the resiliency of the Sboxes of cryptographic algorithms. This is accomplished either by changing dynamically the Sboxes of customized algorithms or scrambling the Sboxes of standard algorithms. These type of design techniques are extremely useful for lightweight block ciphers with SBox of smaller dimension. Moreover, generating Sboxes in runtime is an attractive design choice for the designer employing ciphers with secret Sboxes.

To sum up, this paper clearly shows that RLUT is a double-edged sword for security applications on FPGAs. Due to the obvious positive application of RLUTs in security, one cannot simply restrict the use of RLUT in secure applications. This motivates further research in two principal directions. Firstly, there is need for Trojan detection techniques at IP level. This detection techniques should be capable of distinguishing a RLUT based optimizations from potential Trojans. Finally certain new countermeasures totally based on RLUTs can be studied.

References

1. S.M. Trimberger and J.J. Moore. FPGA Security: Motivations, Features, and Applications. *Proceedings of the IEEE*, 102(8):1248–1265, Aug 2014.
2. Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES*, volume 6917 of *LNCS*, pages 33–48. Springer, 2011.
3. Shivam Bhasin, Wei He, Sylvain Guilley, and Jean-Luc Danger. Exploiting FPGA block memories for protected cryptographic implementations. In *ReCoSoC*, pages 1–8. IEEE, 2013.
4. Tim Güneysu and Christof Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In *CHES*, pages 62–78, 2008.
5. Debapriya Basu Roy, Debdeep Mukhopadhyay, Masami Izumi, and Junko Takahashi. Tile before multiplication: An efficient strategy to optimize DSP multiplier for accelerating prime field ECC for NIST curves. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 1–6. ACM, 2014.
6. Tim Güneysu. Getting Post-Quantum Crypto Algorithms Ready for Deployment.
7. Wei He, Andrés Otero, Eduardo de la Torre, and Teresa Riesgo. Automatic generation of identical routing pairs for FPGA implemented DPL logic. In *ReConFig*, pages 1–6. IEEE, 2012.
8. Martin Kumm, Konrad Möller, and Peter Zipf. Reconfigurable FIR filter using distributed arithmetic on FPGAs. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, China, May 19-23, 2013*, pages 2058–2061. IEEE, 2013.
9. Pascal Sasdrich, Amir Moradi, Oliver Mischke, and Tim Güneysu. Achieving Side-Channel Protection with Dynamic Logic Reconfiguration on Modern FPGAs. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2015, Washington, DC, USA, 5-7 May, 2015*, pages 130–136, 2015.
10. F. Madlener, M. Sotttinger, and S.A. Huss. Novel hardening techniques against differential power analysis for multiplication in gf(2n). In *Field-Programmable*

Technology, 2009. FPT 2009. International Conference on, pages 328–334, Dec 2009.
11. Xilinx. Xilinx Partial Reconfiguration User Guide (UG702). `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/ug702.pdf`.
12. Éric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, August 11–13 2004. Cambridge, MA, USA.
13. Subidh Ali, Rajat Subhra Chakraborty, Debdeep Mukhopadhyay, and Swarup Bhunia. Multi-level attacks: An emerging security concern for cryptographic hardware. In *Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011*, pages 1176–1179, 2011.
14. Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware Trojan: Threats and Emerging solutions. In *IEEE International High Level Design Validation and Test Workshop, HLDVT 2009, San Francisco, CA, USA, 4-6 November 2009*, pages 166–171, 2009.
15. Mohammad Tehranipoor and Domenic Forte. Tutorial T4: All You Need to Know about Hardware Trojans and Counterfeit ICs. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, India, January 5-9, 2014*, pages 9–10, 2014.
16. Zhimin Chen, Xu Guo, Raghunandan Nagesh, Anand Reddy, Michael Gora, and Abhranil Maiti. Hardware trojan designs on basys fpga board.
17. Anju P. Johnson, Sayandeep Saha, Rajat Subhra Chakraborty, Debdeep Mukhopadhyay, and Sezer Gören. Fault Attack on AES via Hardware Trojan Insertion by Dynamic Partial Reconfiguration of FPGA over Ethernet. In *Proceedings of the 9th Workshop on Embedded Systems Security*, WESS '14, pages 1:1–1:8, New York, NY, USA, 2014. ACM.
18. Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, Xuan Thuy Ngo, and Laurent Sauvage. Hardware Trojan Horses in Cryptographic IP Cores. In Wieland Fischer and Jörn-Marc Schmidt, editors, *FDTC*, pages 15–29. IEEE, 2013.
19. Jean-Baptiste Note and Éric Rannaud. From the Bitstream to the Netlist. In *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, FPGA '08, pages 264–264, New York, NY, USA, 2008. ACM.
20. Benchmarks. `https://www.trust-hub.org/resources/benchmarks`. Accessed: 2015-01-30.
21. Naofumi Homma, Yu-ichi Hayashi, Noriyuki Miura, Daisuke Fujimoto, Daichi Tanaka, Makoto Nagata, and Takafumi Aoki. EM Attack Is Non-invasive? - Design Methodology and Validity Verification of EM Attack Sensor. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pages 1–16, 2014.
22. Gilles Piret and Jean-Jacques Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and Khazad. In *CHES*, volume 2779 of *LNCS*, pages 77–88. Springer, September 2003. Cologne, Germany.
23. Subidh Ali, Debdeep Mukhopadhyay, and Michael Tunstall. Differential fault analysis of AES: towards reaching its limits. *J. Cryptographic Engineering*, 3(2):73–97, 2013.
24. Axel Poschmann, San Ling, and Huaxiong Wang. 256 Bit Standardized Crypto for 650 GE - GOST Revisited. In Stefan Mangard and François-Xavier Standaert Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 219–233. Springer Berlin Heidelberg, 2010.

25. Suvadeep Hajra, Chester Rebeiro, Shivam Bhasin, Gaurav Bajaj, Sahil Sharma, Sylvain Guilley, and Debdeep Mukhopadhyay. DRECON: DPA Resistant Encryption by Construction. In David Pointcheval and Damien Vergnaud, editors, *AFRICACRYPT*, volume 8469 of *Lecture Notes in Computer Science*, pages 420–439. Springer, 2014.
26. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*, volume 4727 of *LNCS*, pages 450–466. Springer, September 10-13 2007. Vienna, Austria.
27. Xilinx. Virtex-5 fpga system monitor. `http://www-inst.eecs.berkeley.edu/~cs150/fa13/resources/ug192.pdf`.

A Trigger generation for Hardware Trojans

For the hardware Trojan trigger signal, we exploit directly the temperature sensor measurement to generate the ***trigger*** signal. The device, used for this experiment, is Xilinx Virtex 5 FPGA mounted on SASEBO-GII boards. As described in the documentation [27], the temperature measurement is read directly on 10 bits signal output of system monitor. This output allows a value which varies from 0 to 1023. System monitor measurement allows to sense a temperature in range of $[-273°\text{C}, +230°\text{C}]$ hence the LSB of the 10 bits output is equal to $1/2°\text{C}$. At the normal operating temperature (25°C), system monitor output is around $605 = b'1001011101$. Thanks to this observation, we decided to use directly the 7^{th} bit of system monitor output as hardware Trojan trigger signal. The hardware Trojan will be activated when 7^{th} bit of monitor output is high, i.e., when the monitor output is superior to $640 = b'1010000000$. This value corresponds to 42°C. Therefore the trigger signal will be active when FPGA temperature is higher than 42°C. The trigger temperature can be easily changed according to the design under test. In our case study, a simple hair dryer of cost $5 is enough to heat the FPGA and reach this temperature. We assume that a system monitor is already instantiated in the design, to monitor device working conditions and the alarm is raised at a temperature higher than 42°C. In such a scenario, the hardware Trojan trigger part does not consume much extra logic and would result in a very low-cost hardware Trojan example.

Whenever we need to trigger the Trojan, we bring the heater circuit close to the FPGA. The FPGA heats up slowly to the temperature of 42°C and raises the output bit to '1'. At this point, we switch-off the heater. Now this output bit stays '1' till the FPGA cools down below 42°C, therefore we cannot precisely control the duration of trigger in terms of cycle count. We further process this output bit of the system monitor to generate a precise duration trigger. This can be done with some extra logic. In other words, we need a small circuit which can generate a precise trigger signal when the output bit of system monitor goes to '1'. For the Trojans in Tab 1, we either need a trigger of 1 clock cycle or 12 clock cycles. Both these triggers can be generated by deploying one LUT and one flip-flop to process output bit of system monitor. Thus, we can generate a very small trigger circuit to trigger a zero-overhead hardware Trojan.