Reconfigurable LUT: Boon or Bane for Secure Applications

Debapriya Basu Roy
Indian Institute of Technology Kharagpur
deb.basu.roy@cse.iitkgp.ernet.in

Shivam Bhasin
Telecom Paristech
shivam.bhasin@telecom-paristech.fr

Sylvain Guilley
Telecom Paristech
sylvain.guilley@telecom-paristech.fr

Jean-Luc Danger
Telecom Paristech
jean-luc.danger@telecom-paristech.fr

Debdeep Mukhopadhyay
Indian Institute of Technology Kharagpur
debdeep@cse.iitkgp.ernet.in

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 performance of the system. In this paper, we study RLUT in the context of secure applications. Next, we design several case-studies to apply this feature on security critical scenarios. These case-studies vary from destructive scenarios like stealthy hardware Trojans to constructive scenarios like implementation of secret ciphers with custom Sboxes and masking countermeasure.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have had a significant impact on the semiconductor market in recent years. FPGAs have evolved a long way from a mere array of few thousand programmable look-up tables (LUTs) to multi-million LUTs. Moreover, modern FPGAs come with several on-chip features like 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. FPGAs turn out to be a very economical solution in 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, given the critical nature of key markets like defense, space, etc., FPGA manufacturers have started considering security as the fourth parameter. Most recent FPGAs support bit-stream protection by authentication and encryption schemes. Other security features like tamper resistance, blocking bit-stream read-back, temperature/voltage sensing, etc. are also available.

Apart from the built-in security features, designers can use FPGA primitives and constraints to implement their own designs in a secure manner. In [1], authors show several side-channel countermeasures which could be realized in FPGA to protect one design. Another work [2] 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 [3, 4] and other post-quantum algorithms. Moreover, papers like [5] have used FPGA constraints like *KEEP, Lock_PINS* or language like *XDL* to design efficient physical countermeasures.

Another feature which has received very little attention is the reconfigurable LUT (RLUT). As the name suggests, this LUT can be reconfigured during the operation phase to change the input-output mapping of the LUT. Unlike dynamic reconfiguration, a RLUT does not need any external link and can be totally reconfigured on chip. To the best of our knowledge, RLUTs have found relevant use in pattern matching and filter applications [6]. In this paper, we study RLUT and deploy it in 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 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 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).

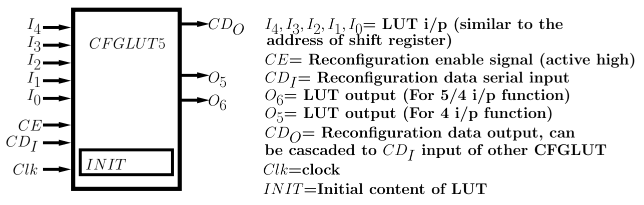
I_4
I_3
I_2
I_1
I_0
CE
CD_I
Clk
$CFGLUT5$
$INIT$
CD_O
O_5
O_6
I_4, I_3, I_2, I_1, I_0= LUT i/p (similar to the address of shift register)
CE= Reconfiguration enable signal (active high)
CD_I= Reconfiguration data serial input
O_6= LUT output (For 5/4 i/p function)
O_5= LUT output (For 4 i/p function)
CD_O= Reconfiguration data output, can be cascaded to CD_I input of other CFGLUT
Clk=clock
$INIT$=Initial content of LUT

Figure 1: Block diagram of CFGLUT5

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

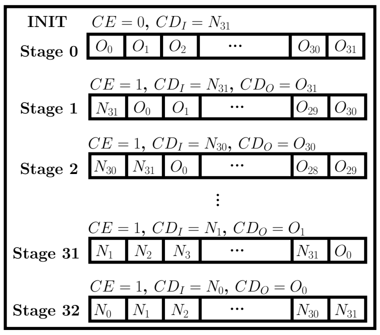
INIT
$CE = 0$, $CD_I = N_{31}$
Stage 0
O_0 O_1 O_2 ··· O_{30} O_{31}
$CE = 1$, $CD_I = N_{31}$, $CD_O = O_{31}$
Stage 1
N_{31} O_0 O_1 ··· O_{29} O_{30}
$CE = 1$, $CD_I = N_{30}$, $CD_O = O_{30}$
Stage 2
N_{30} N_{31} O_0 ··· O_{28} O_{29}
$CE = 1$, $CD_I = N_1$, $CD_O = O_1$
Stage 31
N_1 N_2 N_3 ··· N_{31} O_0
$CE = 1$, $CD_I = N_0$, $CD_O = O_0$
Stage 32
N_0 N_1 N_2 ··· N_{30} N_{31}

Figure 2: $INIT$ value reconfiguration in CFGLUT5

It is a common impression that any LUT can be implemented as CFGLUT5. However, this is not true. The number of CFGLUT5 are far less than the number of LUTs in an FPGA. Basically, 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. CFGLUT5, when instantiated, is essentially mapped into a *SLICE_M*, configured as shift register ($SRL32$) as shown in Fig 3.

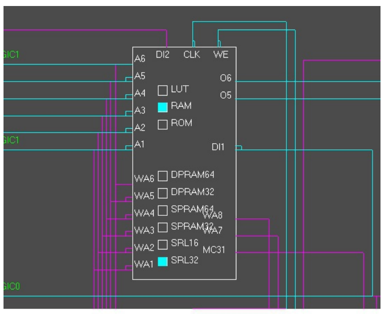

Figure 3: CFGLUT5 mapped in LUT as SRL32 as shown from Xilinx FPGA Editor

2.1 Comparison With Dynamic Configuration

Another way to reconfigure FPGA in run-time is to use partial or dynamic reconfiguration. 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*. Reconfiguration latency is in order of millisecond. 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. Moreover RLUT is configured internally and no external access to either JTAG or Ethernet ports are required for reconfiguring RLUTs.

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). 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 reconfigurabilty 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. In the following sections, we would show some relevant applications of constructive or deadly nature.

3. DESTRUCTIVE APPLICATIONS

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. This section mainly discusses about effective design of hardware Trojan payload using RLUT.

3.1 Adversary Model

We consider an adversary model where a user buys specific IPs from a third party IP vendor. It is a common trend in the semiconductor industry to acquire proven IPs to reduce time to market. 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 backdoors 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*). 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.

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. We can also restore the original value of RLUT to remove any trace of Trojan, of course, at some overhead. An IP designer can easily replace a normal LUT with RLUT. Instantiation of *CFGLUT5* does not report any special element in the design summary report, **but a LUT of *SLICE_M* (*SRL32*).** The only requirement is efficient triggering and a reconfiguration logic which will generate the malicious value upon receiving trigger signal. The basic methodology is same for all the Trojans, which can be tabulated as follows:

- Choose a sensitive sub-module of the crypto-algorithm.
- Replace the LUTs of the chosen sub-module with *CFGLUT5* keeping the same *INIT* value.
- Modify the *INIT* value upon trigger.
- Upon nominal operation, restore original *INIT* value.

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 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, 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. In the following to not deviate from the topic, we focus mainly on the payload design of the Trojan. We do not propose any special triggering circuit and let the designer choose any of the published techniques or innovate one. We precisely propose the design of the Trojan and the required triggering conditions. Notice that, RLUTs can be exploited even to trigger a Trojan.

3.2 Trojan Description

Before designing Trojan 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.

To insert a buffer, a designer can use *LUT1* primitive with `INIT=0x2`. However, this buffer is implemented in a *LUT6* with `INIT=0xAAAAAAAAAAAAAAAA` and can be easily replaced by *CFGLUT5.* A simple Trojan would be to change 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.** 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 or controlling data multiplexers or misreading status flags, etc.

Next, we target a basic *AES* IP as a Trojan target. The AES takes 128 bits of plaintext and key as input and produce 128 bit ciphertext in 11 clock cycles along with a 1-bit *done* signal. Original AES circuit requires **1594 LUTs, 260 flip-flops** and runs at a maximum frequency of **212.85 MHz** on SASEBO-GII boards (Virtex-5). We developed four different Trojans with the objective of obtaining the secret key of the underlying *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. 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. The detailed description of the developed Trojans are as follows:

3.2.1 Trojan 1

This Trojan targets the control unit of *AES* implementation to get access to the secret key. The control unit of *AES* is a counter which also produces a *done* flag to indicate completion of the encryption cycle. *done* is set to high only if counter value reaches 11. Signal *done* is driven by a

LUT, which takes 4 bit counter value as input, and under normal operation it should contain value $0x00000800$. To insert a Trojan we replace this LUT with *CFGLUT5* with `INIT=0x80000800`. The CD_O output is fedback 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. By knowing the plaintext, one can easily extract the full key with one wrong encryption. After extracting the key, a trigger of 10 clock cycles will restore the normal operations of the AES (`INIT=0x00800800`). The transition of *INIT* to activate the Trojan and restore back is shown in Fig 4(a). The value above the 11^{th} bit are not important for normal operation, as a mod12 counter cannot attain those value. Keeping the same Trojan we implemented three different versions, 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. The overhead for this Trojan is 6 **LUTs and** 4 **flip-flops**.
2. **Trojan 1b** is a **zero** overhead Trojan. It assumes a 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.
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 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**.

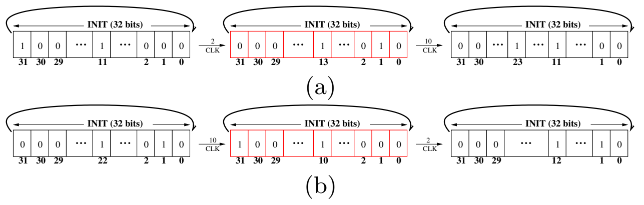

Figure 4: Operations of CFGLUT5 to activate the Trojan and restore to normal operations for (a) Trojan 1; (b) Trojan 2. Bit positions not shown contain '0'

3.2.2 Trojan 2

This Trojan targets the datapath of the *AES* design. The underlying 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 (*select* signal of multiplexer is set to 1) and MixColumns output in other rounds (*select* signal of multiplexer is set to 0). To design the Trojan, we have replaced the LUT (with `INIT=0x00000002`) which generates *select* signal of the multiplexer with *CFGLUT5*, containing `INIT=0x00400002`. Upon a trigger of 10 clock cycles (`INIT=0x80000400`), the multiplexer is programmed to select input plaintext during 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 4(b). Again the value over bit position 12 is not a problem as the *select* signal is controlled by a mod12 counter and the value is never reached. This Trojan also has a **zero** overhead.

Tab. 1 summarizes the nature, trigger condition and cost of the four Trojans. The frequency overhead of all the designs were negligible and hence not mentioned below.

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	Payload Overhead
Trojan 1a	1_s	6 LUTs & 4 flip-flops
Trojan 1b	12_s	0
Trojan 1c	12	2 LUTs
Trojan 2	12_s	0

4. CONSTRUCTIVE APPLICATIONS

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 three 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 Sboxes at its side. 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 [7]. 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 [8], which offers DPA resistance natively by 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 [8], 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 [2] 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.

As a proof of concept, we implement the Sbox generation scheme of [8]. 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. It is based on the theory that if $S(\cdot)$ is a cryptographically strong Sbox, 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. The scheme can be very well implemented using RLUT as follows. We read the initial Sbox and store the newly computed Sbox in the same location. The architecture is shown in Fig 5. 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 5. 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 register.
4. After the 2-bits ({0,2}) of new value of Sbox is shifted in, old value for the input 15 is flushed out. The old value at position 14 is moved up to position 15. Thus the address is hard-coded to `0xf`. We keep on repeating the above steps until we have read the whole old Sbox for all possible inputs, which requires 16 clock cycles.
5. 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.

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.

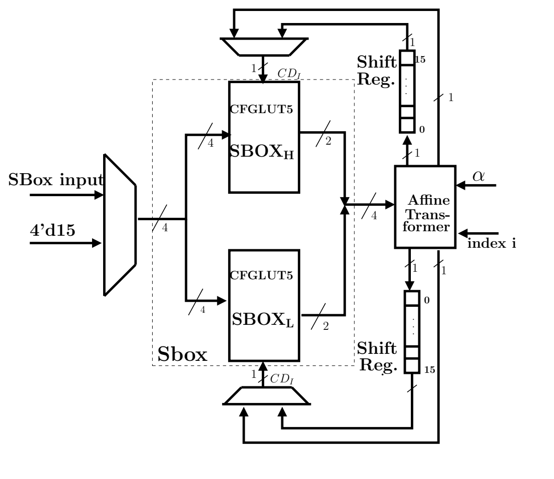

Figure 5: Architecture of Sbox Computation using affine transformation and storing in RLUT

4.2 Sbox Scrambling

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 [1] 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 [1]). This scheme has limited overhead, claims first-order security and, to our knowledge, no practical attack has been published against it.

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 [9], 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 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. 6.

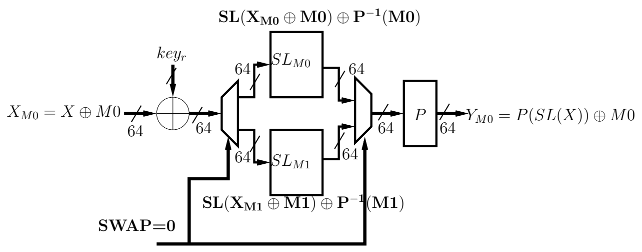
$\mathbf{SL(X_{M0} \oplus M0) \oplus P^{-1}(M0)}$
key_r
$X_{M0} = X \oplus M0$
64
64
64
64
64
64
64
64
SL_{M0}
SL_{M1}
P
$Y_{M0} = P(SL(X)) \oplus M0$
$\mathbf{SL(X_{M1} \oplus M1) \oplus P^{-1}(M1)}$
SWAP=0

Figure 6: Architecture of Modified PRESENT Round. SL_{M0} is the (precomputed) active SLayer while SL_{M1} is being computed as in Fig. 7

BMS is nice 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.

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. Rest of the scheme is left is same as [1]. The architecture of Sbox scrambler using RLUT is shown in Fig 7. $SBOX_P$ 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 inverse permutation of the mask to scramble the Sbox value. Please note the 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 computation. 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. This means we can swap active and inactive Sboxes every 16 encryptions. The area overhead comes from the scrambler circuit and multiplexers used to swap active/inactive Sboxes. It is summarized in Tab. 2.

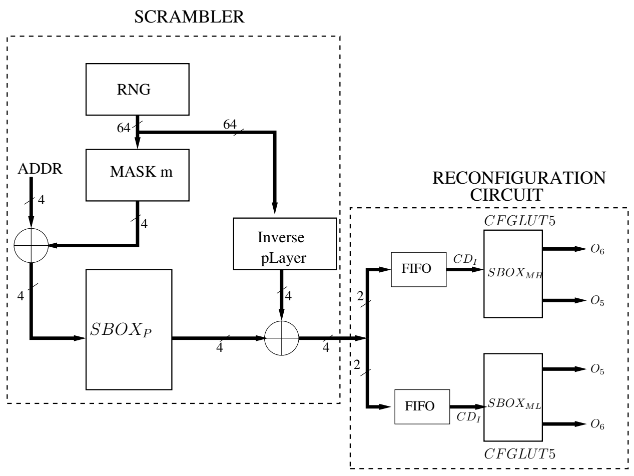

Figure 7: Architecture of Sbox Scrambler

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×

4.3 Design obfuscation

A usual prerequisite for an attack to work is to know the system under attack. Of course, some attacks can work in completely blackbox conditions (consider the cube attack [10]), but the secrecy of the system seriously undermines the potential of attackers. Basically, most attack paths will require an initial stage of reverse-engineering. Therefore, obscuring a netlist can be a very effective preventing protection. The designer can focus on critical parts of the system, such as state machines.

Those are initially programmed by random values, which leads to absurd results of reverse-engineering, be them static or dynamic. Then, in a second step, the circuit requests an unlocking key, which enables a reconfiguration of critical part. Depending on the expected reactivity of the system, the rewriting of the RLUT contents can be achieved in series (low complexity) or parallel (low programmation time). The second step requires an interaction with an external component, or a connection to some network. The missing correct bitstream part is then either reprogrammed, or fed externally.

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.

To sum up, this paper clearly shows the positive and negative impact of RLUT on security of 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 principle 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.

6. REFERENCES

[1] 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.

[2] 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.

[3] Tim Güneysu and Christof Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In *CHES*, pages 62–78, 2008.

[4] 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.

[5] 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.

[6] 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.

[7] Axel Poschmann, San Ling, and Huaxiong Wang. 256 bit standardized crypto for 650 ge £ gost revisited. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 219–233. Springer Berlin Heidelberg, 2010.

[8] 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.

[9] 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.

[10] Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009, Cologne, Germany, April 26-30, 2009. Proceedings*, volume 5479 of *Lecture Notes in Computer Science*, pages 278–299. Springer, 2009.