Protecting against Cryptographic Trojans in FPGAs

Pawel Swierczynski*, Marc Fyrbiak*, Christof Paar*, Christophe Huriaux†, and Russell Tessier‡
*Ruhr-Universität, Bochum, Germany
†IRISA, University of Rennes I, Lannion, France
‡University of Massachusetts, Amherst, MA, USA

***Abstract*—In contrast to ASICs, hardware Trojans can potentially be injected into FPGA designs post-manufacturing by bitstream alteration. Hardware Trojans which target cryptographic primitives are particularly interesting for an adversary because a weakened primitive can lead to a complete loss of system security. One problem an attacker has to overcome is the identification of cryptographic primitives in a large bitstream with unknown semantics. As the first contribution, we demonstrate that AES can be algorithmically identified in a look-up table-level design for a variety of implementation styles. Our graph-based approach considers AES implementations which are created using several synthesis and technology mapping options. As the second contribution, we present and discuss the drawbacks of a dynamic obfuscation countermeasure which allows for the configuration of certain crucial parts of a cryptographic primitive *after* the algorithm has been loaded into the FPGA. As a result, reverse-engineering and modifying a primitive in the bitstream is more challenging.**

I. Introduction

A major security issue for SRAM-based FPGAs is the external storage of the FPGA bitstream. Recent work has shown that bitstream encryption is not impervious to attack [1], and with sufficient effort, the logical function of any FPGA design can be reverse-engineered from a decrypted bitstream, even though the bitstream configuration mapping is proprietary. With the goal of weakening system security, the next hurdle for an adversary is the identification of the cryptographic algorithm within the target third-party design. Instantiations of cryptographic primitives such as the Advanced Encryption Standard (AES) or Data Encryption Standard (DES) implement specific non-linear and key-based operations. As previously shown [2], this regularity provides a mechanism for locating the logic that corresponds to the primitives even in very large logic-level netlists.

In this contribution, we show that there are more general methods available beyond those shown in previous work, which was based on identifying the specific output values of S-Box tables [2]. We introduce a graph-matching approach which can be used to identify look-up table (LUT) and flip-flop-based implementations of cryptographic primitives. Our method is effective for AES cores synthesized using a wide variety of mapping options (area-optimized, delay-optimized, etc.) and it is FPGA architecture independent.

As the second main contribution of this paper, we present a dynamic obfuscation countermeasure which reduces the risk of detection of the crucial, non-linear S-Box layer through automatic and static reverse-engineering. Our approach relies upon a partial self-configuration of the S-Box layer *after* the FPGA bitstream is loaded, rendering specific modification of S-Boxes in the bitstream ineffective.

II. AES Identification in LUT-Level Netlists

To motivate our approach of a post-bitstream-load self-configuring AES core, we demonstrate that it is possible to identify static AES cores in large LUT-based FPGA designs in an algorithmic manner, regardless of the synthesis approach used to create the LUT-based design or the size of the design. This type of static analysis mimics the possible actions of an attacker looking to locate and afterwards manipulate the AES functionality in a LUT-based netlist. In our approach, we make the following two assumptions regarding the target netlist: first, the third-party design may contain other elements and not solely the AES. Second, the AES is implemented using LUTs and flip-flops as well as precomputed S-Box tables. Further, we assume that the attacker has access to the netlist (through bitstream reverse-engineering), but has no information regarding the pre-synthesis RTL or high-level implementation of the AES core. Also, no information is available to the attacker regarding the synthesis tool or synthesis parameters used to create the LUT-based design.

Our graph-based algorithm uses a LUT-level, third-party netlist as input and attempts to locate subgraphs which represent portions of AES combinational logic functionality. Combinational logic in the input design between primary inputs/flip-flop outputs (CI) and primary outputs/flip-flop inputs (CO) is represented as a Directed Acyclic Graph (DAG). The algorithm first determines the *support* (DAG input CIs) for each flip-flop input (CO). Flip-flops which have a matching support are grouped into registers. A backward DAG traversal is then initiated from each register CO to locate logic cones. The combinational logic within each cone is matched against known combinational subfunctions identified in an AES core (e.g., XOR, S-Boxes) in an attempt to isolate the core.

To illustrate the capabilities of our graph search for AES primitives, the algorithm was implemented in C and used to assess netlists generated using the open-source VTR flow[1] including ODIN II register-transfer synthesis, ABC logic synthesis, and VPR place and route.

Library and design setup: In an initial step, a library of synthesized common internal AES transformations was created. ODIN II and ABC were used to synthesize these

[1] https://code.google.com/p/vtr-verilog-to-routing/

functions (variants of XORs and 8-input, 1-output functions representing partial S-Box functionality) into And-Inverter Graphs (AIGs). The library functions were subsequently written as a series of independent files in BLIF format. Subsequently, an open source RTL version of AES-128 from OpenCores[2] was embedded within RTL versions of four VTR benchmarks (*ch_intrinsics*, *mkDelayWorker32B*, *mkSMAdapter4B*, and *sterovision0*). Each design was synthesized with ODIN II and ABC to interconnected collections of 6-LUTs and flip-flops using three different synthesis ABC scripts: *resyn/resyn2* (default, timing-optimized), *compress/compress2* (area-driven), and *rwsat* (SAT-based). These synthesized designs represent the type of LUT-based netlists an attacker would be able to obtain via reverse-engineering.

TABLE I: AES subgraph search results. The S-Box column detection indicates the number of 8-input, 1-output S-Box components identified during the search (datapath components / key schedule components) out of (128/32) possible. LUT counts synthesized with *resyn* are shown in parentheses.

Design	Synth. script	S-Box Column Detection	Sub-graphs	Overall time (s)	BM time (s)
aes	resyn	124/32	252,810	249.57	223.56
	compr	111/28	207,030	208.19	186.17
	rwsat	128/32	284,170	265.65	239.93
ch_	resyn	124/32	487,530	957.48	916.33
intrinsics	compr	111/28	363,760	462.95	428.04
(7,560)	rwsat	128/32	459,830	567.43	526.43
mkSM-	resyn	124/32	644,760	528.13	405.26
Adapter4B	compr	111/28	604,000	499.11	376.83
(9,480)	rwsat	128/32	695,890	537.98	418.87
mkDelay-	resyn	124/32	1,082,370	977.62	580.95
Worker32b	compr	111/28	1,018,050	1020.21	618.07
(12,060)	rwsat	127/32	1,070,830	963.89	576.46
stereo-	resyn	124/32	1,150,870	3485.93	793.83
vision0	compr	111/28	1,104,550	3670.51	802.80
(18,510)	rwsat	128/32	1,253,420	3488.30	812.37

Search algorithm: Our search algorithm, which was integrated into ABC, operates in a series of search steps. First, the LUT-level netlist is read and combinational logic is flattened into a series of AIGs. The *support* for each CO is determined via a backward netlist traversal from the CO to CIs. Flip-flops with a greater than 50% overlap in support for more than eight inputs are grouped into registers. A series of searches are then initiated from each register flip-flop to identify logic cones with k-inputs. The combinational function of each cone is compared against stored library functions of k-inputs to identify matches (e.g., XOR, S-Box functionality). If a match is identified, each cone input is used as the starting point for the generation of additional cones and additional match operations. The graph search for a cone is terminated when logic depth of the cone (in terms of 2-input AND gates) exceeds depth $d = 25$ (the upper limit logic depth of an AES S-Box) or all CIs are reached.

Boolean matching: A key aspect of our function search approach is the use of Boolean matching. Since the input order of candidate cones may differ from the ordering in the stored library version, it is necessary to assess the combinational equivalence of the graphs under a collection of input permutations (*p-equivalence*). Our approach uses a Boolean matching algorithm integrated into ABC [3].

Our graph identification approach was applied to a synthesized and LUT-mapped AES core and the four VTR with embedded AES benchmarks mentioned in this section. Three synthesized versions of the five benchmarks using the scripts mentioned earlier were analyzed. The same library of common AES transformations was used for the five analyses. In all cases, all XOR gates used for the AddRoundKey transformations were identified along with components of all twenty 8-input, 8-output S-Boxes present in the AES core[3]. The results of our approach are summarized in Table I. The S-Box detection results are reported in terms of the number of 8-input, 1-output S-Box components that were identified in the design for the AES datapath (out of $128 = 8 \cdot 16$ possible) and the key schedule (out of $32 = 8 \cdot 4$ possible). Regardless of the synthesis script used to create the initial design, it was possible to identify almost all individual S-Box components. Boolean matching (BM) time is included in the search overall time in the table.

III. Partial self-configuration as protection

Once (parts of) the AES have been identified in a third-party design, an algorithm substitution attack [4] can be launched in order to weaken the cipher. Altering the S-Box layer is attractive, because it is identifiable in an algorithmic manner, cf. Section II and, crucially, manipulating it does not require any re-routing efforts. The practical feasibility of an S-Box substitution attack, with a particular focus on third-party FPGA designs, has been shown recently for AES and DES by Swierczynski *et al.* [2]. If the S-Boxes are maliciously altered in the bitstream, the cryptographic key can be leaked or the resulting ciphertext can directly be decrypted using trivial cryptanalysis. Such a modification of the FPGA configuration exhibits a serious problem for real-world applications where encryption and decryption are performed by the same device, e.g., by a USB flash drive.

Attacker Model: We only consider static attacks which attempt to identify and alter precomputed AES S-Boxes. To be more precise, no potential round counter, state machine, or data transfer modifications, etc. are considered in this work.

The strategy behind our countermeasure is to significantly increase the reverse-engineering efforts for an attacker (using static analysis) who intends to mount such an S-Box substitution attack, as described above. Our *raise the bar* countermeasure hides the precomputed S-Box values in a cryptographic manner, thus an attacker is not able to easily detect and modify any S-Box value in a predictable fashion.

Key Idea of the S-Box Protection: The main idea of our approach is to first instantiate a modified AES core with random and bijective S-Boxes. After the FPGA has been initially configured, the random S-Boxes are dynamically updated

[2]http://opencores.org/project,aes_core

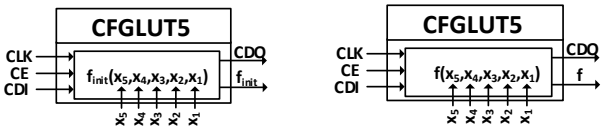
[3]Round-based AES design with 16 S-Boxes for the state transformation (datapath) and 4 S-Boxes for the key schedule operation.

(through encryption[4] with the altered AES) to the correct AES S-Boxes to achieve proper encryption and decryption behavior. Thus, the correct S-Box values are only dynamically available and cryptographically hidden for an attacker who only statically analyzes the design and searches for the correct AES S-Boxes.

During the system design phase, the correct AES S-Boxes are *decrypted* with the initial modified AES. The resulting ciphertexts are stored in the distributed LUT memory of the FPGA design. Note that the ciphertexts may be more easily identified if they are stored in the block memory and not in the FPGA's distributed memory. The ciphertext bits are distributed over multiple LUTs and hence identification and reverse-engineering seem more challenging (compared to block memory) for an attacker using static analysis. After FPGA power-up, the design starts to recover and replace the random S-Boxes with the correct ones by *encryption* of the ciphertexts (decrypted correct S-Boxes). Once the reconfiguration phase is finished, the valid AES can be used to encrypt and decrypt data. To implement our proposed countermeasure, we make use of dynamic look-up tables that can be changed during runtime of the FPGA, i.e., only by the FPGA configuration itself. Additionally, we implemented an AES design using block memory to support smaller FPGA devices.

Dynamic LUT reconfiguration is used as a building block for our countermeasure. This feature is supported by several Xilinx FPGA families, in particular we implemented the protection scheme utilizing a Spartan 6 Xilinx FPGA device.

Building Block, Xilinx CFGLUT5: Xilinx provides a dynamic LUT primitive called *CFGLUT5* (5-input, 1-output LUT).



(a) Initial LUT with Boolean function f_{init}

(b) Reconfigured LUT with Boolean function f

Fig. 1: Dynamic reconfiguration of a CFGLUT5 primitive

Fig. 1a illustrates the initial state of a CFGLUT5 (directly after FPGA power-up and initial configuration), while Fig. 1b depicts the reconfigured state. The f_{init} Boolean function output values are encoded in the FPGA bitstream, whereas the Boolean function output values of f are reconfigured during runtime.

Reconfiguration of CFGLUT5 Elements: Three input reconfiguration pins are available for this hardware element. The *CDI* pin handles the new LUT content data or functionality. The *CE* pin needs to be pulled high to activate or keep the reconfiguration process running. The clock is used to configure the current data of the *CDI* pin. The 5-input Boolean function f is coded by $2^5 = 32$ output values that are being shifted bit-wise for updating the functionality. Two output pins are provided by the CFGLUT5: through the *CDO* pin, the old configuration data can be read out and the other output pin provides the image of the Boolean function f.

Memory Organization: In the case of AES, an S-Box instance can be implemented using $2048 = 2^8 \cdot 8$ bits of memory. Thus, $64 = \frac{2048}{32}$ CFGLUT5 primitives are used and connected together in a consecutive chain, cf. the lower part of Fig. 2. As all 64 units are consecutively connected together over an 1-bit data bus, 2048 clock cycles are required to reconfigure all S-Box instances. To be more precise, one AES 8-input, 1-output S-Box column is divided into 8 individual 5-input, 1-output subfunctions. We denote by $S_i^j(x)$, the i^{th} S-Box column and the j^{th} subfunction (implemented in the CFGLUT5 elements). For each S-Box column i, all subfunctions $(S_i^1(x), \ldots, S_i^8(x))$ have to be multiplexed using the remaining input bits, i.e., x_8, x_7, and x_6 in Fig. 2.

As previously discussed in this section, we require a random and bijective AES S-Box denoted by $\widetilde{S}(x)$ in the following. This S-Box $\widetilde{S}(x)$ is the initial Boolean function of all CFGLUT5 units (and thus available in the proprietary bitstream configuration mapping). This slightly modified AES core (using the random S-Box $\widetilde{S}(x)$) is denoted by $\widetilde{AES}$. In the following, each step from system design to FPGA power-up is described to present the concept in its entirety.

Phase I – System Design: During the system design phase, the following steps have to be carried out: first, a random and bijective 8-input, 8-output S-Box $\widetilde{S}(x)$ and its inverse $\widetilde{S}^{-1}(x)$ have to be generated. Then, a so-called *reconfiguration key* k_1 has to be generated and the correct AES S-Box values (256 bytes) have to be *decrypted* with $\widetilde{AES}_{k_1}^{-1}$ (whose S-Box is replaced by the random and bijective S-Box $\widetilde{S}(x)$). To be more precise, the S-Box plaintext is computed as follows:

$$\widetilde{c}_i := \widetilde{AES}_{k_1}^{-1}(\widetilde{p}_i), \quad i = 1, 2, \ldots, 16$$

A 128-bit input $\widetilde{p}_i$ (the original AES S-Box $S(x)$ divided into its columns and subfunctions) is defined as follows (the $||$ denotes the concatenation of a 32-bit value):

$$\widetilde{p}_i := \begin{cases} S^1_{\lceil \frac{i}{2} \rceil}(x) \;||\; S^2_{\lceil \frac{i}{2} \rceil}(x) \;||\; S^3_{\lceil \frac{i}{2} \rceil}(x) \;||\; S^4_{\lceil \frac{i}{2} \rceil}(x) & \text{i odd} \\ S^5_{\lceil \frac{i}{2} \rceil}(x) \;||\; S^6_{\lceil \frac{i}{2} \rceil}(x) \;||\; S^7_{\lceil \frac{i}{2} \rceil}(x) \;||\; S^8_{\lceil \frac{i}{2} \rceil}(x) & \text{i even} \end{cases}$$

The S-Box $S(x)$ is represented in the above manner, so that it can be directly loaded into the CFGLUT5 FPGA elements. Fig. 2 provides a detailed overview of the inner workings of our dynamic AES core. Note that the encryption function $\widetilde{AES}_{k_1}$ is initially implemented in the FPGA design.

Phase II – FPGA Power-Up: The following steps are performed to dynamically recover the valid AES core, directly after the FPGA power up. First, the 16 ciphertext blocks $\widetilde{c}_i$ are *encrypted* by $\widetilde{p}_i := \widetilde{AES}_{k_1}(\widetilde{c}_i)$ to recover the valid AES S-Box values. Second, all $\widetilde{S}(x)$ instances are dynamically reconfigured to $S(x)$ (utilizing the values $\widetilde{p}_i$) from the previous step. After this process, the $\widetilde{AES}$ core turns into the correct AES core. Finally, the reconfiguration key k_1 is set to the data encryption key k_2.

[4]Note that the decryption function may/must not be available in the FPGA design, e.g., due to the implemented mode of operation or the system setting.

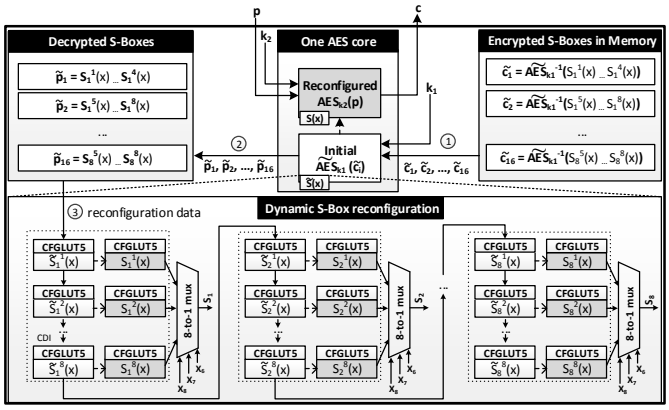


Fig. 2: Dynamic reconfiguration of AES S-Boxes using CFGLUT5 elements

We now examine the performance and utilized hardware area of our dynamically reconfigurable AES core. We implemented a dynamic DES core based on the reconfiguration scheme (using the same method as outlined for AES). Corresponding implementation results are also provided.

Backwards Compatibility: Since not all Xilinx FPGA families support dynamic CFGLUT5 elements, we also implemented a similar approach for AES and DES that instantiates S-Boxes using block RAM instead of dynamic LUTs. Due to similarity with the LUT-based scheme, the block RAM implementation is not further explained, although its resource usage is provided.

For performance and hardware area evaluation we used the same synthesis options (using Xilinx ISE 13.2) for each FPGA implementation and verified the functionality on a Spartan 6 FPGA board (SP601 evaluation kit with a XC6SLX16). We compared three different implementation strategies: (unprotected) static S-Boxes in LUTs, (protected) S-Boxes instantiated in CFGLUT5 memory, and (protected) S-Boxes instantiated in block RAM. The results are depicted in Table II. The amount of utilized slices, LUTs and flip-flops, etc. refer to a single instantiation of one AES / DES core. Each design contains a UART core due to the communication interface that is required for verification.

TABLE II: Static LUT-based / dynamic DES and AES designs

Design	#Slices	#FFs	#LUTs			#RAMBs		f_{max}
			6-to-1	5-to-1	Sh-Reg.	8	16	(MHz)
DES								
Static LUT	142	341	374	52	1	–	–	165
Dyn. CFGLUT5	1005	4570	2480	1125	138	–	–	183
Dyn. BRAM	218	495	313	186	–	8	2	156
AES								
Static LUT	1256	1619	3410	332	1	–	–	203
Dyn. CFGLUT5	1371	4065	1848	1376	1130	–	–	211
Dyn. BRAM	451	1444	1234	346	1	16	4	238

The performance of the AES and DES implementations varies moderately. The large amount of flip-flops and shift registers in the dynamic CFGLUT5 designs (compared to the static LUT-based and block RAM designs) is based on the interconnections of all CFGLUT5 units for AES and DES.

IV. Cautionary Notes on Cryptographic Implementations in FPGA Bitstreams

As stated earlier, if the bitstream encryption scheme (if available at all) can be overcome with moderate effort, a design's integrity and confidentiality is no longer certain, even if the bitstream configuration mapping is proprietary. The reverse-engineering of an FPGA netlist (that was obtained through bitstream reverse-engineering) is less explored in the research community and thus real-world attacker capabilities are unclear. As shown in this work, algorithmic detection of cryptographic S-Boxes and XOR operations is feasible, hence a reverse-engineer can use this information as an entry point of his FPGA design analysis. To further impede algorithm substitution attacks, a memory access control unit could be implemented that analyzes the written memory content to raise the bar for an S-Box replacement attempt. By extension of the attacker model, several other vulnerabilities might occur in our scheme. A static analysis might allow for the identification and modification of the random S-Box layer. Also, a stronger attacker might modify the reconfiguration circuitry. Both strategies would lead to an unwanted key extraction. Thus, we recommend the implementation of selftest circuitry that adds further protection to the system. Future research should explore how a cryptographic design can be obfuscated such that automatic detection of primitives is challenging and hence hardware Trojan insertion, e.g., through algorithm substitution, becomes a time consuming task for an adversary.

V. Conclusions and Future Work

In this paper we briefly described a protection approach for cryptographic primitives, such as AES and DES, to defend against simple and static bitstream S-Box layer manipulations. The S-Boxes in an AES core are configured *after* bitstream loading, as a partial defense against Trojan insertion in the bitstream. To motivate our approach, we show that it is algorithmically possible to identify standard AES cores in LUT-based designs by means of static analysis, even if the attacker has no information about the original RTL design or the synthesis parameters used to create the LUT-based design. The security limitations of our *raise the bar* approach are also briefly analyzed.

References

[1] A. Moradi, D. Oswald, C. Paar, and P. Swierczynski, "Side-Channel Attacks on the Bitstream Encryption Mechanism of Altera Stratix II - Facilitating Black-Box Analysis using Software Reverse-Engineering," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2013, pp. 91–100.

[2] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "FPGA Trojans through Detecting and Weakening of Cryptographic Primitives," *IEEE Transactions on CAD*, 2015, http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7029686.

[3] H. Katebi and I. L. Markov, "Large-scale Boolean Matching," in *IEEE/ACM Design Automation and Test in Europe Conference*, Mar. 2010, pp. 771–776.

[4] M. Bellare, K. G. Paterson, and P. Rogaway, "Security of Symmetric Encryption against Mass Surveillance," in *Advances in Cryptology - CRYPTO - 34th Annual Cryptology Conference*, Aug. 2014, pp. 1–19.