Trojans Modifying Soft-Processor Instruction Sequences Embedded in FPGA Bitstreams

Ismail San*†, Nicole Fern†, Cetin Kaya Koc†, and Kwang-Ting (Tim) Cheng†
*Anadolu University, Eskisehir, Turkey
†University of California, Santa Barbara, CA, USA
isan@anadolu.edu.tr, nicole@ece.ucsb.edu, koc@cs.ucsb.edu, timcheng@ece.ucsb.edu

***Abstract*—Reconfigurable platforms such as FPGAs and CPLDs are used to implement flexible and lightweight embedded systems often using soft-processors and a fixed instruction sequence stored in block memories. The bitstream format is proprietary for most vendors, however, in this work we demonstrate how to identify and extract block memory contents within the bitstream, allowing an adversary to learn and possibly modify the fixed instruction sequence. Manipulating the instruction sequence by inserting a Trojan in the bitstream as opposed to in the RTL code allows an adversary to bypass many verification steps. Moreover, the proposed Trojans only add extra instructions to the sequence to leak secret information, and do not change the original program behavior, making them virtually impossible to detect using functional tests. We present a case study where a Trojan is injected into a MIPS AES encryption program to leak internal state information by adding extra instructions from the available ones without changing the original program behavior.**

***Index Terms*—Security, Trojan, System-on-chip, FPGA**

I. Introduction

Reconfigurable platforms such as Field Programmable Gate Arrays (FPGA)s and Complex Programmable Logic Devices (CPLD)s are often used to implement flexible embedded systems. Due to the sophistication of adversaries [1] and increasing use of soft-IP in applications [2], securing reconfigurable platforms is critical and the task has become increasingly challenging. FPGA security has been explored from various perspectives [3], [4] including bitstream integrity and confidentiality.

Typically programs in embedded systems are *infinite loops*, meaning the instruction sequence will continue to execute on the soft-core processor until the device is turned off. *"Embedded systems now refers to any system with fixed software on the chip"* [5]. Hence, there is **fixed program code**, alive for the lifetime of the system, stored in the configuration bitstream, which may contain sensitive information. This motivates powerful adversaries to reverse-engineer and modify the bitstream for such an embedded system.

Typically the instruction sequence is embedded in portions of the bitstream which will be transferred to block memory when the device is powered. In this work, we present an algorithm to reverse engineer the bitstream format to learn the contents of the block memory. In systems with soft-core processors, this means any attacker able to examine the device bitstream can learn the entire instruction sequence (provided the instruction format is publicly available). This is a serious security flaw for applications running proprietary algorithms assuming program code is safely obfuscated within the bitstream.

Moreover, the attacker can deduce the desired functionality of the program then insert a Trojan in the program instruction sequence by adding extra *valid* instructions which do not interfere with the original program behavior, but perform some malicious functionality such as leaking information or establishing timing or power side channels. Because these Trojans are inserted in the bitstream as oppose to the C or register transfer level (RTL) code for the soft-core processor, there are limited verification mechanisms to detect the modifications which are not cryptographically secure [6]. Most FPGA devices have a cyclic redundancy check (CRC) mechanism to detect and correct errors during transmission of the bitstream to the device, yet they can be disabled, as shown by Chakraborty *et al.* [7] and easily manipulated [8]. To protect against the proposed attack bitstream compression and memory encryption can be employed, however due to space limitations a detailed discussion of these techniques is not included in this work.

II. Literature Review

A. Bitstream Reverse Engineering and Modification

There exist open-source Computer-Aided Design (CAD) tools for the FPGA design process [9], [10] as well as methods to reverse-engineer the bitstream format [4], [11]–[13]. However, these studies only target a limited number of FPGA devices and provide very limited information about the bitstream format. Swierczynski *et al.* [14] provide practical content extraction methods for look-up table (LUT)s and embedded memory in order to detect the S-box used in cryptographic algorithms and manipulate its contents to weaken the corresponding cryptographic primitive. Their S-box modification completely changes the behavior of the AES cipher, meaning even a single test encryption will compute an incorrect ciphertext and reveal the modification. In contrast, our case study focuses on the instruction stream for the AES encryption algorithm stored in block memory, not on the modification of S-boxes stored in LUTs, and illustrates how an attacker can learn the AES key without causing the computation of incorrect ciphertexts.

B. Program Code Integrity

Preventing hijacking of processor program code has been addressed and studied extensively [15]. Chang and Atallah [16] propose a method to protect program integrity by computing the checksum of the program code in a distributed manner. Jacob *et al.* [17] introduce a technique based on oblivious hashing for the integrity of the program code. Instead of checking code integrity prior to execution, their method provides a runtime verification mechanism. These mechanisms for protecting program code are costly, and applicable to code executing on high-performance processors, but in lightweight embedded systems these dynamic integrity checking mechanisms may be infeasible.

III. Threat Model

The attack model targets FPGA-based embedded systems containing peripherals, a bus interconnect, and a soft-processor. The soft-processor has an instruction memory whose contents are loaded from the FPGA bitstream. Our threat model assumes the attacker is capable of introducing a Trojan at the bitstream level to modify the instruction sequence and injects extra instructions to leak information without altering the original functionality of the program.

There are several scenarios in which the attacker is able to access and modify the bitstream. For example, if the bitstream for the system is provided by an untrusted third party, it may contain a Trojan. Because our threat model assumes the Trojan preserves the functionality of the device from the user's perspective, even if thorough functional testing is performed by the user, it is unlikely they will detect such Trojan modifications.

In another scenario, the adversary can obtain access to the bitstream by having physical access to the FPGA device, as in [18], where the bitstream is intercepted while being transferred from non-volatile to volatile memory during start-up.

In order to check the data integrity of the configuration data, bitstreams usually come with a built-in CRC. However, there are ways to disable it. Chakraborty *et al.* [7] describes how to disable CRC verification. Another way of stopping our proposed attack is to encrypt the configuration data with a cryptographic algorithm. Two dominant leaders in the FPGA market provide bitstream encryption mechanisms to stop Intellectual Property (IP) counterfeiting and related issues [3]. However, there exist many practical attacks [19]–[21] to leak the cryptographic keys used by the device for this bitstream encryption.

IV. Retrieving Block Memory Content Embedded in Configuration Data

Block memory units are used to store register files, state information, first-in first-out (FIFO) buffers, and algorithm look-up tables and constants. When the system is running cryptographic algorithms or processing private information, the security of these memory units is essential.

In the configuration bitstream, a fixed mapping is used for the contents of block memories. Algorithm 1 shows the methodology used to understand this mapping. To perform Algorithm 1, the attacker must be able to generate bitstreams for the specific FPGA device targeted. To learn the mapping, the attacker will create several versions of an RTL design instantiating block memory with known contents then compare how these differences are reflected in the bitstream. Once this analysis is performed for a particular device, block memory contents residing in any bitstream generated for the device can be learned by the attacker. Block memories for a given FPGA device support several configurations, each having different depths and data word widths. Memory depth is the total number of words that one can store in a block memory while the bit length of each word is determined by the width. We denote bit positions within each memory word as *columns* while *rows* define the location of each word. Algorithm 1 extracts the column and row decoding information for a given block memory configuration.

Algorithm 1 Revealing the row and column encodings of block memory contents embedded in the FPGA bitstream

Input: An RTL level design consisting of a memory instantiation of block memories for a selected configuration; $\mathsf{B}[N_r][N_c]$ (a block of data in the design), N_r (number of rows) and N_c (number of bits in a row).

Output: $\mathcal{D}_c[N_c]$ (Decoding information for the columns) and $\mathcal{D}_r[N_r]$ (Decoding information for the rows).

```
1.  for $i \leftarrow 0$ to $N_r - 1$ do                    (Block Initialization)
2.      $\mathsf{B}[i] \leftarrow 0$;
3.  end for
4.  Place and Route the design; Write the bitstream ($\mathcal{B}_0$);
5.  $\mathsf{B}[0] \leftarrow 1$;
6.  Place and Route the design; Write the bitstream ($\mathcal{B}_1$);
7.  $\mathcal{B}_{diff} \leftarrow \mathcal{B}_1 \bigoplus \mathcal{B}_0$;
8.  Find the address ($Addr_1$) of 1 in $\mathcal{B}_{diff}$
9.  $\mathcal{D}_c[0] \leftarrow Addr_1$
10. for $j \leftarrow 1$ to $N_c - 1$ do          (Decoding Column Mapping)
11.     $\mathsf{B}[0] \leftarrow (1 << j)$;
12.     Place and Route the design; Write the bitstream ($\mathcal{B}_{j+1}$);
13.     $\mathcal{B}_{diff} \leftarrow \mathcal{B}_{j+1} \bigoplus \mathcal{B}_0$;
14.     Find $Addr_{j+1}$ in $\mathcal{B}_{diff}$
15.     $\mathcal{D}_c[j] \leftarrow Addr_{j+1}$
16. end for
17. $\mathsf{B}[0] \leftarrow 0$;
18. for $i \leftarrow 1$ to $N_r - 1$ do              (Decoding Row Mapping)
19.     $\mathsf{B}[i] \leftarrow 1$;
20.     Place and Route the design; Write the bitstream ($\mathcal{B}_i$);
21.     Find the bit location ($l$) of the 1 in $Addr_1$ of $\mathcal{B}_i$
22.     $\mathcal{D}_r[j] \leftarrow l$
23. end for
24. return $\mathcal{D}_c$ and $\mathcal{D}_r$;
```

First, Algorithm 1 initializes the block memory contents with zeroes in the attacker-generated RTL design consisting of instantiations of block memories for a selected configuration. A bitstream, $\mathcal{B}_0$, is generated and over the course of the Algorithm we compare this initial bitstream with bitstreams generated with different, but known, block memory contents.

We observe that there is a row and column encoding which can be revealed by only examining a few bit locations, meaning our method avoids an exhaustive search of all possible bit decodings (which would entail assigning 1 to every bit to get as many bitstream files as the number of bits in a block memory).

The column decoding is revealed by (1) assigning 1 to each bit only for the first memory word, (2) generating corresponding bitstream files $\mathcal{B}_j$, (3) comparing them with $\mathcal{B}_0$ and (4) storing the address of the difference as a column decoding ($\mathcal{D}_c$) (Algorithm 1, lines 5 through 16). Note that it is sufficient to only examine a single word since all column decodings are identical within a memory block. To reveal the row decoding, (1) assign 1 to the least significant bit of each word, (2) generate corresponding bitstream files $\mathcal{B}_i$ and (3) examine the word at address $Addr_1$ in $\mathcal{B}_i$ and store the bit position of the 1 as a row decoding ($\mathcal{D}_c$) where $Addr_1$ is computed during the column decoding process (lines 18 through 23 of Algorithm 1).

Algorithm 2 Retrieving the contents of the instruction memory embedded in FPGA bitstream

```
Input: $\mathcal{D}_c[N_c]$ (Decoding information for the columns) and
    $\mathcal{D}_r[N_r]$ (Decoding information for the rows), $N_r$ (number
    of rows) and $N_c$ (number of bits in a row); The bitstream
    ($\mathcal{B}$) is to be modified, Configuration frame size ($Nb_{cf}$),
    Number of Frames ($N_f$), Header size ($Nb_H$), HEXDUMP
    utility is to view binary data in hexadecimal view.
Output: Contents of the Instruction Memory (IM)
1.  Remove $Nb_H$ bits in $\mathcal{B}$.
2.  for $i \leftarrow 0$ to $N_f - 1$ do     (Decoded Block Initialization)
3.    for $k \leftarrow 0$ to $N_r - 1$ do
4.      $\mathsf{DB}[i][k] \leftarrow 0$;
5.    end for
6.  end for
7.  for $i \leftarrow 0$ to $N_f - 1$ do     (Retrieving All Frames)
8.    $\mathsf{F}[i] \leftarrow$ HEXDUMP $\mathcal{B}$ in blocks of $Nb_{cf}$ bits;
9.  end for
10. for $i \leftarrow 0$ to $N_f - 1$ do     (Decoding Frames)
11.   for $j \leftarrow 0$ to $N_c - 1$ do     (Decoding Columns)
12.     $c \leftarrow \mathsf{F}[i][\mathcal{D}_c[j]]$
13.     for $k \leftarrow 0$ to $N_r - 1$ do     (Decoding Rows)
14.       $m \leftarrow 1 << \mathcal{D}_r[k]$
15.       $\mathsf{DB}[i][k] \leftarrow (c \oplus m) >> \mathcal{D}_r[k]$
16.     end for
17.   end for
18. end for
19. Search possible instruction sequences in decoded blocks.
20. $\mathsf{IM}[i][k] \leftarrow$ Instruction Sequences in $\mathsf{DB}[i][k]$
21. return
```

For a specific block memory configuration (example depth=8, width=32) this mapping is fixed, therefore the mapping revealed for a single block applies to all blocks with the same configuration on the device. One can use this fixed mapping to read the contents of the block memory sequentially from the bitstream.

Algorithm 2 operates on the bitstream the attacker wishes to maliciously modify, and describes a method to extract the instruction sequences embedded in the block memories. Once one knows the encoding of the block memory configuration by applying Algorithm 1, Algorithm 2 searches for possible instruction sequences in the contents of the block memories.

For a given bitstream $\mathcal{B}$, Algorithm 2 first removes the header, zeros the temporary decoded block (DB) variable, and dumps the bitstream $\mathcal{B}$ in blocks of Nb_{cf} bits as a hex file (F). Algorithm 2 then decodes each frame in the bitstream file using the column and row decoding information ($\mathcal{D}_c$ and $\mathcal{D}_r$, respectively) and stores the decoded contents in DB (Algorithm 2, lines 10 through 18). As a final step, Algorithm 2 searches for possible instruction sequences in DB. Instruction sequences are recognizable due to fixed opcodes for a given instruction set architecture. Additionally, high level knowledge of the algorithm also makes identification of meaningful instruction sequences possible. For example, in many cryptographic applications round constants, initial key values, and S-box mapping information are commonly stored in block memories.

V. Trojans Modifying Instruction Memory Embedded in an FPGA Bitstream

Block memories in an FPGA device are a natural choice for storing instruction sequences for a given processor. Instructions carry complete information about the algorithm being executed, and control the information flow in the data-path. Once the block memory contents are known using the Algorithms presented in Section IV, the functionality of the instruction sequence can be understood by the attacker, and places where extra instructions can be advantageously inserted can be identified.

The following section details this methodology for a MIPS-based instruction sequence performing AES encryption. Trojans created through the addition of extra malicious instructions which do not disturb the original program functionality are not limited to information leakage in cryptographic applications. One can easily extend the ideas in the proposed attack to different applications.

A. Case Study: Trojan Insertion in AES Instruction Sequence

1) MIPS-based Platform Overview: In order to demonstrate the operation of the proposed Trojans, we implement a Trojan infested MIPS based system on an AC701 evaluation board featuring the XC7A200T FPGA device. The system consists of data and instruction memories residing in a dual-port block memory, a MIPS processor, and several peripherals for communication outside the FPGA including an LCD controller, switches, LEDs, and a UART serial interface. The methodology given in the following section is applicable to similar platforms which have instruction memory embedded in block memories of an FPGA device. The MIPS processor runs an AES-128 algorithm and drives the LCD display, LEDs, and UART channel. For our case study, the processor encrypts a 128-bit text with a 128-bit key and then the encrypted text is sent through a UART channel.

2) Trojan Insertion Methodology: First, the block memory contents stored in the bitstream are extracted using Algorithms 1 and 2, then the contents are interpreted as program code and disassembled using the freely available Online Disassembler (ODA) [22] and the MIPS instruction sequence is extracted.

The next step in Trojan insertion is to identify possible attack points in the program code. One should note that the AES key may not be stored directly in the bitstream. For example, a physically unclonable function (PUF) can generate the key at runtime or it may be provided from outside the processor. Therefore, we assume we do not have direct access to the AES key from the bitstream, and the attack point selected is the AddRoundKey transformation of AES which involves an XOR operation with round keys. From the extracted program code it is possible to distinguish AES transformations since they have a fixed pattern. Listing 1 shows the AddRoundKey step implemented in the MIPS processor. The code loads state values of AES and round keys in 8-bit increments and applies the XOR operation to them. After identifying necessary code segments related to the sensitive information processed in the device, we leak this information through a peripheral observable by the attacker. In this case study, the output of the UART communication controller is easily visible to anyone with physical access to the device. Since systems often have memory mapped Input/Output (I/O)s, one can easily identify a UART write operation in the instruction sequence. After identifying the subroutine corresponding to the UART write operation, data can be sent through the UART channel by writing the data to register `a0` (argument to the subroutine) and calling the subroutine with a jump and link instruction (`jal 4b4` in Listing 1). To leak the key information, we inject an instruction performing a UART write operation during the AddRoundKey step (Listing 1).

```
518:  3c020000  lui    v0,0x0
51c:  8c471308  lw     a3,4872(v0)
520:  00042100  sll    a0,a0,0x4
524:  3c020000  lui    v0,0x0
528:  24850004  addiu  a1,a0,4
52c:  24421258  addiu  v0,v0,4696
530:  00452821  addu   a1,v0,a1
534:  24e80010  addiu  t0,a3,16
538:  24a3fffc  addiu  v1,a1,-4
53c:  00e01021  move   v0,a3
540:  90640000  lbu    a0,0(v1)
544:  90460000  lbu    a2,0(v0)
548:  24630001  addiu  v1,v1,1
54c:  00862026  xor    a0,a0,a2
550:  0c00012d  jal    4b4         #UARTWriteByte
554:  a0440000  sb     a0,0(v0)
558:  14a3fffa  bne    a1,v1,540   #AddRoundKey+0x28
55c:  24420001  addiu  v0,v0,1
560:  24e70004  addiu  a3,a3,4
564:  14e8fff5  bne    a3,t0,538   #AddRoundKey+0x20
568:  24a50004  addiu  a1,a1,4
56c:  03e00008  jr     ra
570:  00000000  nop
```

Listing 1. Code segment from MIPS instruction sequence corresponding to the AddRoundKey step in AES (compiled with MIPS cross-compiler toolchain from the C code in https://github.com/kokke/tiny-AES128-C). The red instruction is the injected jump-and-link instruction to the UART channel write subroutine.

In our case study, we leak the intermediate results of the AddRoundKey steps for each round, but the attacker can easily recover the AES key using this information from the first round only. It is also possible to leak the AES key directly by calling the UART write subroutine before the XOR operation if the register carrying the AES key is known. Adding extra instructions into the program code after its been processed by an assembler breaks branch and jump instructions due to the fact that instructions following the injected code are shifted in memory from their original locations. A final detail in our attack is to increment the addresses in branch/jump instructions to restore correct program behavior. The modified program code is reintroduced to the bitstream using the information provided by Algorithm 1. The instructions in the resulting bitstream still correctly perform the functionality of the original instruction sequence, however, the internal state of the AES algorithm is leaked, allowing an attacker to recover the AES key.

VI. Conclusion

We present a methodology for reverse engineering soft-core processor instruction sequences embedded in portions of the bitstream corresponding to block memories. We then show how to modify these contents to create an instruction sequence which leaks sensitive algorithm information by *inserting extra instructions without affecting the functionality of the original program code.* Since our Trojans are introduced in the bitstream after the place-and-route (PAR) process, we avoid most of the verification and Trojan detection mechanisms at RT, synthesis, and PAR levels. We provide an example of our proposed Trojans by modifying the bitstream for a platform containing a soft-core MIPS processor performing AES encryption and several peripherals by injecting additional instructions into the encryption program to leak internal AES state information without disturbing the encryption algorithm.

References

[1] S. M. Trimberger and J. J. Moore, "FPGA security: Motivations, features, and applications," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1248–1265, Aug 2014.

[2] D. Jansen, Ed., *The electronic design automation handbook*. Springer Science & Business Media, 2003.

[3] S. Trimberger, "Trusted design in FPGAs," in *Proceedings of the 44th Annual Design Automation Conference*, ser. DAC'07, 2007, pp. 5–8.

[4] S. Drimer, "Security for volatile FPGAs," Ph.D. dissertation, University of Cambridge, 2009.

[5] M. Birnbaum, *Essential Electronic Design Automation (EDA)*, ser. Prentice Hall Modern Semiconductor Design Series. Prentice Hall PTR/Pearson Education, 2004.

[6] B. Badrignans, F. Devic, L. Torres, G. Sassatelli, and P. Benoit, *Security Trends for FPGAS: From Secured to Secure Reconfigurable Systems*, 2011, ch. Embedded Systems Security for FPGA, pp. 137–187.

[7] R. S. Chakraborty, I. Saha, A. Palchaudhuri, and G. K. Naik, "Hardware trojan insertion by direct modification of FPGA configuration bitstream," *IEEE Design Test*, vol. 30, no. 2, pp. 45–54, April 2013.

[8] T. Güneysu, I. Markov, and A. Weimerskirch, "Securely sealing multi-FPGA systems," in *Proceedings of the 8th Int. Conf. on Reconfigurable Computing: Architectures, Tools and Applications*, 2012, pp. 276–289.

[9] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-it-yourself CAD tools for Xilinx FPGAs," in *Int. Conf. on Field Programmable Logic and Applications*, 2011, pp. 349–355.

[10] R. K. Soni, N. Steiner, and M. French, "Open-source bitstream generation," in *Field-Programmable Custom Computing Machines (FCCM), IEEE 21st Annual International Symposium on*, April 2013, pp. 105–112.

[11] J.-B. Note and E. Rannaud, "From the bitstream to the netlist," in *Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, ser. FPGA'08, 2008, pp. 264–264.

[12] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *International Conference on Field Programmable Logic and Applications*, Aug 2012, pp. 735–738.

[13] D. Ziener, S. Assmus, and J. Teich, "Identifying FPGA IP-cores based on lookup table content analysis," in *International Conference on Field Programmable Logic and Applications*, Aug 2006, pp. 1–6.

[14] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, "FPGA trojans through detecting and weakening of cryptographic primitives," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, pp. 1236–1249, Aug 2015.

[15] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, "A taxonomy of computer program security flaws," *ACM Comput. Surv.*, vol. 26, no. 3, pp. 211–254, Sep. 1994.

[16] H. Chang and M. J. Atallah, "Protecting software code by guards," in *Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, ser. DRM'01, 2002, pp. 160–175.

[17] M. Jacob, M. H. Jakubowski, and R. Venkatesan, "Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings," in *Proceedings of the 9th Workshop on Multimedia & Security*, ser. MM&Sec'07, 2007, pp. 129–140.

[18] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar, "Interdiction in practice – Hardware Trojan against a high-security USB flash drive," Cryptology ePrint Archive, Report 2015/768, 2015.

[19] 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 *Proc. of the ACM Int. Symp. on Field Programmable Gate Arrays*, 2013, pp. 91–100.

[20] A. Moradi, A. Barenghi, T. Kasper, and C. Paar, "On the vulnerability of FPGA bitstream encryption against power analysis attacks: Extracting keys from Xilinx Virtex-II FPGAs," in *Proceedings of the 18th ACM Conf. on Computer and Communications Security*, 2011, pp. 111–124.

[21] A. Moradi, M. Kasper, and C. Paar, "Black-box side-channel attacks highlight the importance of countermeasures: An analysis of the Xilinx Virtex-4 and Virtex-5 bitstream encryption mechanism," in *Proceedings of the 12th Conference on Topics in Cryptology*, 2012, pp. 1–18.

[22] "ODA - the online disassembler." [Online]. Available: https://www.onlinedisassembler.com/odaweb/