A Microprocessor Protection Architecture against Hardware Trojans in Memories

Conference Paper · April 2020

DOI: 10.1109/DTIS48698.2020.9080961

5 authors, including:

Alperen Bolat
TOBB University of Economics and Technology
6 PUBLICATIONS **21** CITATIONS
SEE PROFILE

Oguz Ergin
TOBB University of Economics and Technology
106 PUBLICATIONS **1,762** CITATIONS
SEE PROFILE

Marco Ottavi
University of Twente
175 PUBLICATIONS **2,319** CITATIONS
SEE PROFILE

A Microprocessor Protection Architecture against Hardware Trojans in Memories

Alperen Bolat[a], Luca Cassano[b], Pedro Reviriego[c], Oguz Ergin[a], Marco Ottavi[d]
[a]TOBB University of Economics and Technology, Turkey, [b]Politecnico di Milano, Italy
[c]Universidad Carlos III de Madrid, Spain, [d]University of Rome Tor Vergata, Italy
[a]{alperenbbolat12, oergin}@gmail.com, [b]luca.cassano@polimi.it, [c]revirieg@it.uc3m.es, [d]ottavi@ing.uniroma2.it

***Abstract*—Software exploitable Hardware Trojan Horses (HWTs) have been currently inserted in commercial CPUs and, very recently, in memories. Such attacks may allow malicious users to run their own software or to gain unauthorized privileges over the system. Therefore, HWTs are nowadays considered a serious threat both from academy and industry. This paper presents a protection architecture meant to shield the communication between the CPU and the memory in a microprocessor-based system. The architecture aims at detecting the activation on HWTs infesting the instruction and data memories of the system. Our proposal relies on the use of Bloom Filters (BFs) that are included in ad-hoc designed checkers and integrated in the protection architecture. BFs guarantee zero false alarms and a small (and configurable) percentage of undetected alarms. We applied the protection architecture to a case study system based on a RISC-V microprocessor implemented on an FPGA and running a set of software benchmarks. Our proposal demonstrated to be able to detect more than 99% of possible HWTs activations with zero false alarms. We measured a lookup table overhead ranging from 0.68% up to 10.52% and a flip-flop overhead between 0.68% and 0.99%, and with no working frequency reduction.**

***Index Terms*—Bloom Filter, Hardware Security, Hardware Trojan Horses, Microprocessor-based System, RISC-V**

I. Introduction and related work

The increasing complexity of integrated circuits (ICs) and the seek for low production cost and short time-to-market, led to the globalization of the design and fabrication process of silicon devices [1]. More and more often the design of some of the hardware modules is outsourced, third-party intellectual property cores (3PIPs) are bought, and sometimes also masks and the final chip fabrication are outsourced [2]. On the one hand, this globalization allows for a significant reduction of design cost and time, but it comes with a significant loss of trust in the delivered ICs [3].

Indeed, it is very hard to ensure the trustworthiness of all the parties involved in such a globalized supply chain. As a consequence, the product is exposed to a huge number of threats, among which overproduction [4], counterfeiting [5], 3PIPs licenses violation and abuse [6] and Hardware Trojan Horses (HWTs) [7]. A HWT is a very hard-to-detect modification of a design that is meant to stay silent most of the time, while in specific (usually rare) conditions it alters the nominal behavior of the system or it steals information. HWTs may be inserted by vendors in 3PIPs [8], by employees in the developed HDL code, by CAD tools [9] and by mask providers and silicon foundries in the final layout [10].

Given the difficulty of insertion in real-world circuits and their limited dangerousness in the past HWTs were considered an issue more by academy than by industry. Nevertheless, in the last years, complex *software-exploitable HWTs* have been found and inserted in real-world commercial microprocessors. It has been demonstrated that a HWT may allow the attacker to execute his/her own malicious software, to modify the running software or to acquire root privileges [11]–[13]. Moreover, in 2018, a HWT, called the *Rosenbridge* backdoor, has been found in a commercial Via Technologies C3 processor [14]. This Trojan could be activated and exploited via software to enter in supervisor mode.

Given the complexity of modern integrated circuits and the extreme stealthy nature of HWTs it is more and more difficult to detect HWTs before the system has been deployed. Indeed, together with the "classical" *circuit-level* techniques that aim at detecting HWTs at design time (logic testing [15], formal property verification [16], side-channel analysis [17], structural and behavioral analysis [18], [19]) there is a growing interest in *system-level* techniques that allow to obtain a trusted system built with untrusted components [20]–[22]. A similar idea has been proposed in [23], [24] where the focus is on microprocessor-based systems and the goal is to achieve a trusted software execution with an untrusted CPU. On the other hand, very recently also HWTs in memory chips have been studied [25]. At the same time, as discussed in [25], there has not yet been enough work in protecting microprocessor-based systems from HWTs inserted in memories.

In this paper we propose a system-level architecture for protecting microprocessor-based systems against HWTs. More in details, the proposed architecture aims at detecting the runtime activation of HWTs infesting both the instruction and data memories of a Harvard architecture. We aim at detecting those software-exploitable HWTs that force the microprocessor to run a malicious code and/or to read/write data in unauthorized memory locations. Moreover, a subset of the possible denial-of-service and information stealing HWTs are also addressed by our methodology. The proposed protection architecture relies on two checkers based on *Bloom filters (BFs)* that monitor the instructions fetched from the instruction memory and the accessed addresses in both the instruction and data memory. It is worth mentioning that the proposed solution is completely transparent w.r.t. the normal functioning of the system. Indeed, the runtime monitoring is performed without

interrupting code execution.

We applied the proposed microprocessor protection architecture to a case study system based on a RISC-V processor implemented on an FPGA device and running a set of software benchmarks. Our proposal was always able to detect more than 99% of possible HWTs activations with zero false alarms. We measured a lookup table overhead ranging from 0.68% up to 10.52% and a flip-flop overhead between 0.68% and 0.99%, with no working frequency reduction.

To the best of our knowledge no system-level protection methodologies against HWTs in the memories of microprocessor-based systems have been yet proposed. The works that we consider as the more similar to our proposal are the ones in [23], [24] where, like in our proposal, the problem is tackled at the system-level (and not at the circuit-level) and where a protection unit is inserted between the CPU and the memory. On the other hand, unlike in our proposal, in these works the microprocessor is assumed to be untrusted and the memory to be trusted. In [23] the protection unit checks whether the opcode of the executed instructions and the associated control signals are legal or not and whether the number of clock cycles employed to execute an instruction is correct or not. In [24] the protection unit checks whether the microprocessor is still alive and whether it is running in the right privilege mode. Both solutions do not take into account those HWTs that change the functionality of the system by making the CPU run normal instructions. In other words, none of these works checks whether the microprocessor is executing an unwanted software and whether it is accessing illegal memory locations.

The remainder of this paper is organized as follows: Section II presents the models of HWTs that are targeted by our proposal while Section III briefly presents background information about Bloom filters; Section IV presents the proposed protection architecture, the checkers on which it relies and the design flow that we used to customize such filters; Section V discusses results from a case study application of the proposed architecture on a RISC-V based system; Finally, Section VI concludes the paper.

II. The considered threat model

As we previously mentioned, a HWT is a very hard-to-detect modification of a design that stays silent most of the time, while in specific rare conditions it alters the nominal behavior of the system or it steals information. According to the taxonomy presented in [7], HWTs may be classified based on their *triggering mechanism*, *payload* and *insertion phase*.

A HWT may be triggered: i) *internally* by logical signals (or sequences of logical signals) or by physical quantities, e.g., temperature or voltage, or by a counter (the so-called time bombs); ii) *externally* by either received messages or physical interactions, e.g., again temperature or voltage; and iii) *always-on*, i.e., HWTs that become active as soon as the system is turned on.

When looking at the payload, HWTs may be classified in:

- *Change functionality HWTs* that modify the functionality carried out by the infected system;
- *Information stealing HWTs* that leak unauthorized information through either the available communication interfaces or covert side-channels, e.g., temperature or magnetic field; and
- *Denial-of-service HWTs* that halt the functioning of the system, e.g., by introducing *nop* instructions, by draining the system's batteries, or by jamming the communication interfaces.

Finally, HWTs may be inserted by IP providers in the purchased 3PIPs, by malicious designers and by the employed CAD tools possibly in every stage of the design flow and by the foundry during chip fabrication.

In this work we consider HWTs infesting the instruction and data memory of a microprocessor-based system. On the other hand, the microprocessor is here considered to be trusted. The effectiveness of the proposed methodology does not depend on the trigger. Indeed, HWTs having any of the previously discussed triggering mechanisms are addressed. When looking at the payload, we are able to detect those HWTs that change the functionality of the system by forcing the CPU to execute an unwanted code. Moreover, we are able to detect information stealing HWTs that read/write secret information in unauthorized memory locations.

From the HWT insertion point of view, it is worth mentioning that, since the proposed detection methodology works at runtime, it is able to detect HWTs that have been inserted by any of the actors taking part in the design process and supply chain of the memory chip.

Denial-of-service HWTs that halt the system by maliciously making the CPU fetch always the same instruction (or sequence of instructions) could be detected by providing the proposed architecture with an ad-hoc dimensioned watchdog (this falls outside the scope of the paper). Further, by exploiting watchdogs that monitor the fetching activity of the processor, the proposed methodology could detect denial-of-service HWTs that freeze the CPU.

Information-stealing HWTs that send the stolen information through a covert channel and denial-of-service HWTs that act outside the microarchitectural level, e.g., drain the batteries or jam the communication interfaces, fall outside the scope of this proposed methodology.

III. Background: the Bloom filters

Bloom filters (BFs) are widely used probabilistic data structures that store the membership information of a set of elements. Queries can be run on BFs to check whether an element belongs to the set or not [26], [27]. A key feature of BFs is that, although querying may result in false positive, it is always true that there could not be false alarms. In other words, if a query of an element returns a positive there is a chance for it not to be in the set (*undetected alarm* in our case); however if a negative is returned, it is not possible that the element is in the set (*false alarm* in our case). A BF is implemented as a bit array where each element of the set of interest is mapped

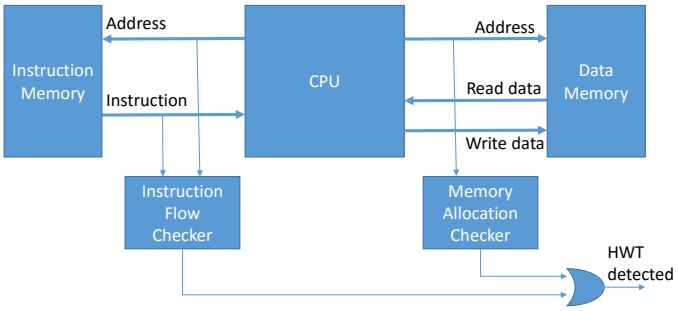

Figure 1. The proposed microprocessor protection architecture

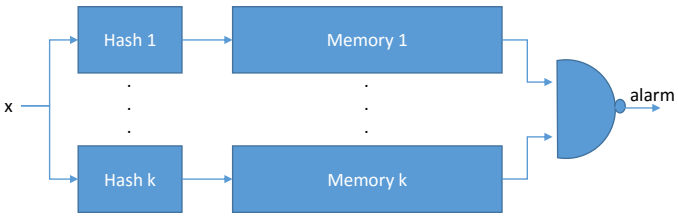

Figure 2. Hardware implementation of a Bloom Filter

to one or more array locations. The address of each location is associated with the output of an hash function calculated on the element itself. Two operations can be performed on a BF: *load* and *query*. Loading a BF means storing in it all the elements of the set of interest: to do so, for each element e_i in the set and for each hash function $hash_j$ the bit array location associated with address $hash_j(e_i)$ is set to 1 (initially all bit arrays are 0). Querying a BF means checking whether an element is in the set of interest, and thus that the associated location(s) of the bit array is (are) set to 1. For a given element e_i that has to be checked and for each hash function $hash_j$ the BF raises an alarm if at least one of the bit array locations associated with address $hash_j(e_i)$ is found to be 0.

BFs guarantee zero false alarms but there can still be a not null percentage of undetected alarms. Nevertheless, the percentage of such undetected alarms can be calculated as a functions of m, the size of the bit array (expressed in bits), of n, the number of elements that have been inserted in the BF and of k, the number of employed hash functions. The formula to calculate the *undetected alarms rate (UAR)* is the following:

$$uar \approx (1 - e^{-\frac{n \cdot k}{m}})^k. \quad (1)$$

therefore, as m increases, UAR decreases. The value of k that minimizes the UAR is given by:

$$k_{opt} = \frac{m}{n} \cdot ln2. \quad (2)$$

IV. The Proposed Microprocessor Protection Architecture

The proposed architecture protects the microprocessor from HWTs inserted in the instruction and data memory. In particular, it aims at detecting the activation of anomalous behaviors in the instruction flow and in the memory address space. This is done by adding two checkers that monitor the fetched instructions and the CPU accesses to the instruction and data memories, as shown in Figure 1 for a generic Harvard architecture, i.e. with instruction memory separated from the data memory.

The two checkers, namely the *Instruction Flow Checker* and the *Memory Allocation Checker*, monitor at runtime whether the accesses to the instruction and data memory, respectively, are those expected by the program that is being executed. Moreover, the Instruction Flow Checker monitors whether the CPU the instructions fetched by the CPU are those expected by the program. In case of unexpected memory accesses or fetched instructions the checkers immediately signal an issue, possibly leading to fast countermeasures such as pipeline flush, interrupt generation or others (countermeasures against HWTs fall outside the scope of this paper). To detect unexpected behaviors the checkers should be able to store all the correct $< address, instruction >$ combinations of the running program, and all the authorized data addresses. To efficiently store these values we implemented the two checkers as Bloom filters (like what has been done in [28] for fault detection).

A. The designed checkers

As we previously discussed, both the Instruction Flow Checker (IFC) and the Memory Allocation Checker (MAC) rely on BFs. A BF can be efficiently implemented in hardware using k memories such that each hash function maps to one of them. Then, the memories can be read in parallel and the results are combined with an NAND gate to obtain the final result. An alarm is raised as soon as at least one of the accessed memory locations contains a 0. A high-level representation of the implemented BF structure is illustrated in Figure 2 where x is the tuple $< memory\ address, fetched\ instruction >$ for the IFC and it is only the $memory\ address$ for the MAC. This structure achieves an undetected alarm rate (UAR) that is similar to the value calculated by Equation 1.

As we previously discussed, the considered HWT models cause the fetch of unexpected sequences of instructions and/or the access to unauthorized instruction and data memory addresses. Therefore, the BF in the IFC is loaded with all the tuples $< memory\ address, instruction >$ of the program under analysis and the MAC is loaded with all the possible data memory addresses accessed by the program. This information are extracted at design time from an execution trace obtained through simulation (see Subsection IV-B).

At runtime the instructions fetched from the memory together with their address are queried in IFC before executing the instruction itself waiting the response of the checker to enable the execution. Similarly, before accessing a data memory address the MAC is queried to verify whether the requested address is legal for the running program or not.

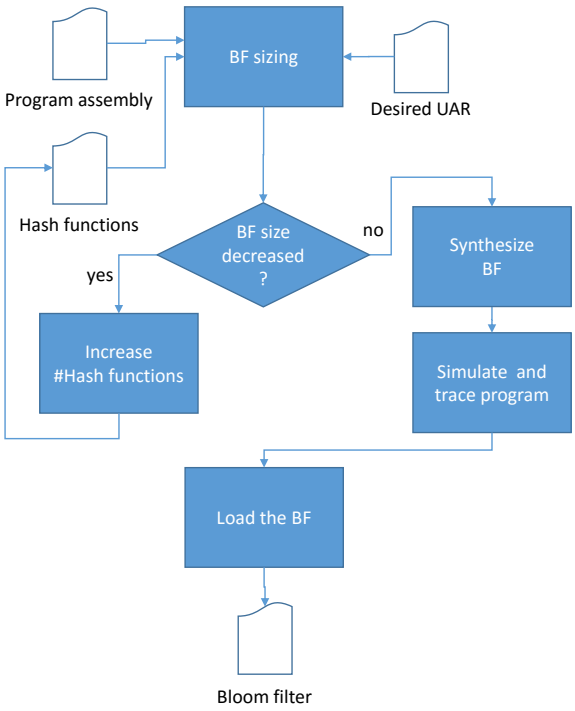

Figure 3. The design flow for the Bloom filter

B. The checker's design flow

We implemented the design flow depicted in Figure 3 to determine the best configuration of the BFs, in terms of size of the bit array and number of employed hash functions, and to synthesize and load them before deployment. The flow takes the assembly code of the program under analysis, a set of hash functions and the desired maximum acceptable undetected alarm ratio (UAR). At the end of the flow the BF synthesized and loaded ready for deployment is generated.

The first step of the flow consists in an iterative process meant to identify test best number of hash functions and size of the bit array of the BF to meet the desired UAR. Given the program's assembly, the desired UAR and an initial number h of hash functions, we calculate the required bit array size by means of Equation 1. By increasing the number of employed hash functions the required bit array size decreases as predicted by Equation 2. Given this, and based on the assumption (experimentally confirmed in Subsection V-C) that the area occupation of hash functions is smaller than the one of memories, we keep increasing the number of employed hash functions while the bit array size decreases significantly. When the reduction is small, we stop such iterative process and we synthesize a BF having the identified bit array size and number of hash functions.

Then, we load the BF for the IFC with all the pairs of address and associated instruction. Similarly, the BF of the MAC is loaded with the trace of all the memory addresses issued by the CPU toward the Data memory (the memory contents are not considered since they are variable in their nature). In order to obtain such accessed data memory addresses we simulate in Modelsim the microprocessor under analysis to produce a trace of all the addresses issued on the data memory address bus.

Once the BFs of the IFC and the MAC have been loaded, the protection architecture can be fully instantiated and the protected microprocessor-based system can be deployed.

Table I
THE CONSIDERED BENCHMARK PROGRAMS

Benchmark	#Instructions
`Binary Sort (BinS)`	1719
`Matrix Multiplication (MM)`	1733
`Bubble Sort (BubS)`	1775
`Quick Sort (QS)`	1927
`Sudoku Solver (SS)`	3227
`Motion Detection (MD)`	4452

V. EXPERIMENTAL RESULTS

We carried out a set of experiments to assess the effectiveness and the efficiency of the proposed protection architecture. We here provide details about the considered hardware platform and benchmark programs; we report results from the simulation-based analysis that was carried out to drive the implementation of the BFs; and we finally discuss the accuracy and the efficiency of hardware-implemented checkers.

A. Experimental setup

The BF sizing step of the design flow has been implemented as a C++ program. For synthesizing the BFs we employed Xilinx Vivado targeting a Xilinx Artix XC7A35T device.

After defining the best parameters for the BFs, we integrated the checkers in an FPGA-based emulation platform running a RISC-V core which provides a Harvard architecture. We considered the PULPINO architecture which is an ultra-low-power processing platform mainly targeted to Internet of Things applications. We considered the RI5CY [29] version of PULPINO, which is a small 4-stage RISC-V core. When synthesized on a Xilinx Artix XC7A35T RI5CY requires 14616 LUTs, 8959 FFs and 16 BRAMs, and it works at 50MHz, as reported in [30].

Finally, we considered a set of benchmark programs (reported in Table I together with the number of assembly instructions) varying from simple sorting algorithms to the more complex `Sudoku Solver` and `Motion Detection`.

B. Bloom filters sizing

After compiling the set of benchmark programs to obtain the corresponding assembly code we dimensioned the BFs by following the flow described in Subsection IV-B. In particular, we fixed the maximum acceptable value of the undetected alarm rate (UAR) at 1% and we calculated the minimum bit array size to be adopted when considering several numbers of employed hash functions. Tables II and III report about this experiment with 3 up to 7 hash functions for the IFC and the MAC, respectively. It can be noticed that in both

Table II
SIZE (IN BITS) OF THE BIT ARRAY OF THE IFC AT 1% UAR FOR DIFFERENT NUMBER OF EMPLOYED HASH FUNCTIONS

Bench.	#Hash Functions				
	3	4	5	6	7
BinS	41200	24720	17512	18320	17064
MM	42816	23232	17848	18120	17848
BubS	42544	27360	19584	19304	18128
QS	46184	30488	20544	20208	19624
SS	65296	51056	33140	31648	32432
MD	68760	53552	44080	43080	42272

Table III
SIZE (IN BITS) OF THE BIT ARRAY OF THE MAC AT 1% UAR FOR DIFFERENT NUMBER OF EMPLOYED HASH FUNCTIONS

Bench.	#Hash Functions				
	3	4	5	6	7
BinS	1080	1016	944	816	752
MM	1192	1072	912	976	880
BubS	1336	1048	984	808	712
QS	1368	1136	1016	824	720
SS	3712	3080	2552	2864	2576
MM	5416	4912	4664	4992	4520

Table IV
SIZE (IN KBIT) AND ACHIEVED UAR OF THE HARDWARE IMPLEMENTATION OF THE BFS FOR THE IFC AND THE MAC

Bench.	IFC		MAC	
	Mem. size	UAR (%)	Mem. size	UAR (%)
BinS	32	0.523	1	0.085
MM	32	0.520	1	0.122
BubS	32	0.572	1	0.525
QS	32	0.607	1	0.073
SD	64	0.249	4	0.134
MD	64	0.912	8	0.232

Table V
RESOURCE OCCUPATION AND WORKING FREQUENCY OF THE IMPLEMENTED IFCS

Bench.	Instruction Flow Checker		
	#LUTs	#FFs	Freq. (MHz)
BinS	880 (6.02%)	84 (0.93%)	112.19
MM	880 (6.02%)	84 (0.93%)	112.19
BubS	880 (6.02%)	84 (0.93%)	112.19
QS	880 (6.02%)	84 (0.93%)	112.19
SS	1539 (10.52%)	89 (0.99%)	106.37
MD	1539 (10.52%)	89 (0.99%)	106.37

experiments the bit array size significantly decreased between 3 and 4 hash functions, and again, between 4 and 5, while further increasing the number of hash functions did not bring any significant reduction of the bit array size. For this reason, we implemented BFs having 5 hash functions to evaluate the real detection accuracy and the introduced overhead.

C. Bloom filters hardware implementation and evaluation

Based on the exploration discussed in the previous subsection, we implemented the BFs on an FPGA for both the IFC and the MAC for the considered benchmark programs with 5 hash functions and choosing a memory size which is the power of 2 value closest to the sizes in bit calculated in Tables II and III, respectively.

First of all we measured the real UAR achieved by the implemented checkers. To do so, we emulated the activation of a HWT belonging to the considered models by modifying the data to/from the instruction and data memories to: i) make the CPU execute an instruction that was not in the original program; ii) make the CPU execute an instruction that was in the original program but that was fetched from an unexpected instruction memory address; and iii) make the CPU read/write data from/to an unexpected data memory address. We simulated 100000 randomly generated HWT activation cases and we measured the numbers of alarms that were not raised. Results from this experiment are reported in Table IV. As expected, the measured UAR is always below 1% (in most cases much below 1%). Then, by running the benchmark programs without any alteration we also checked that the implemented BFs did not raise any false alarm.

Finally, we evaluated the overhead impact of the proposed checkers in terms of used resources and working frequency when targeting an FPGA implementation. Tables V and VI report details for the IFC and the MAC, respectively, for the considered benchmark programs. The resource occupation is reported both in absolute values and in percentage w.r.t. the resource occupation of the RI5CY core. LUT overhead ranges from 0.68% up to 10.52% while, FF overhead ranges from 0.68% up to 0.99%. We believe that such overheads are totally acceptable when considering that the proposed architecture would protect the CPU from a wide range of HWT models and with an extremely small percentage of undetected alarms. Another interesting data related to the resource overhead is that, the broad variation of the LUT usage accounts for the fact that the chosen implementation used LUT also for memory elements of the BF therefore to implement a range from 1 Kbit to 64 Kbit arrays to optimize timing performances. Looking at the working frequency, it can be observed that the proposed checkers do not introduce any slowdown, since the maximum operating frequency is much higher (2x in the worst case) than the one of the considered RI5CY core.

D. Security analysis

The presented experimental results demonstrate that the proposed protection architecture is able to detect much more than the 99% of the runtime activations of HWTs that try to force the CPU to execute malicious code with zero false alarms. In case the designer wants to further reduce the undetected alarm rate, it is sufficient to increase the size of the bit array of the BF, still having zero false alarms.

More in details, any HWTs that try to make the CPU execute instructions that are not in the legal program the CPU is meant to execute or that are in the program but that have been loaded from instruction memory locations out of the memory space of the legal program are detected. Moreover, HWTs that make the CPU read/write data from/to unauthorized data memory addresses are always detected as well. It is worth mentioning that the effectiveness of the proposed solution is

Table VI
RESOURCE OCCUPATION AND WORKING FREQUENCY OF THE IMPLEMENTED MACS

Bench.	Instruction Flow Checker		
	#LUTs	#FFs	Freq. (MHz)
BinS	100 (0.68%)	61 (0.68%)	181.91
MM	100 (0.68%)	61 (0.68%)	181.91
BubS	100 (0.68%)	61 (0.68%)	181.91
QS	100 (0.68%)	61 (0.68%)	181.91
SS	170 (1.16%)	71 (0.79%)	154.13
MD	275 (1.88%)	76 (0.84%)	143.67

independent of the triggering mechanism of the HWT, i.e., combinational/sequential triggered, externally activated, time-bombs and always-on.

The proposed solution could be defeated by denial-of-service HWTs that make the CPU fall into an infinite loop of legal instructions. Providing the protection unit with a watchdog could easily solve such vulnerability. Finally, HWTs that steal information by sending it through covert side-channel are still able to defeat the proposed solution.

VI. CONCLUSION

We presented a protection architecture (and the companion design flow) for the identification of the runtime activation of hardware Trojan horses in the memories of microprocessor-based systems. The proposed architecture can be fine-tuned in order to achieve a desired maximum acceptable undetected alarm rate while having zero false alarms. We applied the proposed microprocessor protection architecture to a case study system based on a RISC-V processor implemented on an FPGA device and running a set of software benchmarks. Our proposal was always able to detect more than 99% of possible HWTs activations with zero false alarms. We measured a lookup table overhead ranging from 0.68% up to 10.52% and a flip-flop overhead between 0.68% and 0.99%, with no working frequency reduction.

REFERENCES

[1] DIGITIMES, "Trends in the global IC design service market." http://www.digitimes.com/news/a20120313RS400.html?chid=2.

[2] M. Rostami, F. Koushanfar, J. Rajendran, and R. Karri, "Hardware security: Threat models and metrics," in *Proc. Int. Conf. Computer-Aided Design*, pp. 819–823, 2013.

[3] Mohammad Tehranipoor and Cliff Wang, *Introduction to Hardware Security and Trust*. Springer-Verlag New York, 2012.

[4] U. Guin, Z. Zhou, and A. Singh, "A novel design-for-security (dfs) architecture to prevent unauthorized ic overproduction," in *2017 IEEE 35th VLSI Test Symposium (VTS)*, pp. 1–6, 2017.

[5] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain," *Proc. IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[6] A. P. Donlin, P. Sundararajan, and B. J. New, "Method and system for secure exchange of ip cores," Aug. 2010. US Patent 7,788,502.

[7] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, 2010.

[8] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware Trojans in third-party digital IP cores," in *Proc. Hardware-Oriented Security and Trust*, pp. 67–70, 2011.

[9] J. A. Roy, F. Koushanfar, and I. L. Markov, "Extended abstract: Circuit cad tools as a security threat," in *2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, 2008.

[10] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *Cryptographic Hardware and Embedded Systems*, 2013.

[11] Y. Jin, M. Maniatakos, and Y. Makris, "Exposing vulnerabilities of untrusted computing platforms," in *Proc. Int. Conf. Computer Design*, pp. 131–134, 2012.

[12] N. G. Tsoutsos and M. Maniatakos, "Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation," *IEEE Trans. Emerging Topics in Computing*, vol. 2, no. 1, pp. 81–93, 2014.

[13] X. Wang, T. Mal-Sarkar, A. Krishna, S. Narasimhan, and S. Bhunia, "Software exploitable hardware trojans in embedded processor," in *2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 55–58, IEEE, 2012.

[14] https://github.com/xoreaxeaxeax/rosenbridge.

[15] X. Chuan, Y. Yan, and Y. Zhang, "An efficient triggering method of hardware Trojan in AES cryptographic circuit," in *Proc. Int. Conf. Integrated Circuits and Microsystems*, pp. 91–95, 2017.

[16] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "Veritrust: Verification for hardware trust," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 7, pp. 1148–1161, 2015.

[17] Y. Liu, Y. Zhao, J. He, A. Liu, and R. Xin, "Scca: Side-channel correlation analysis for detecting hardware trojan," in *Proc. Int. Conf. Anti-counterfeiting, Security, and Identification*, pp. 196–200, 2017.

[18] H. Salmani and M. Tehranipoor, "Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level," in *Proc. Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, pp. 190–195, 2013.

[19] H. Salmani and M. Tehranipoor, "Layout-aware switching activity localization to enhance hardware trojan detection," *IEEE Trans. Information Forensics and Security*, vol. 7, no. 1, pp. 76–87, 2012.

[20] D. Šišejković, F. Merchant, R. Leupers, G. Ascheid, and S. Kegreiss, "Control-lock: Securing processor cores against software-controlled hardware trojans," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, GLSVLSI '19, pp. 27–32, 2019.

[21] D. M. Shila, V. Venugopalan, and C. D. Patterson, "Fides: Enhancing trust in reconfigurable based hardware systems," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, 2015.

[22] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security assurance for system-on-chip designs with untrusted ips," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017.

[23] J. Dubeuf, D. Hly, and R. Karri, "Run-time detection of hardware trojans: The processor protection unit," in *2013 18th IEEE European Test Symposium (ETS)*, pp. 1–6, 2013.

[24] G. Bloom, B. Narahari, and R. Simha, "Os support for detecting trojan circuit attacks," in *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 100–103, 2009.

[25] T. Hoque, X. Wang, A. Basak, R. Karam, and S. Bhunia, "Hardware trojan attacks in embedded memory," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, pp. 1–6, April 2018.

[26] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[27] S. Pontarelli and M. Ottavi, "Error detection and correction in content addressable memories by using bloom filters," *IEEE Transactions on Computers*, vol. 62, pp. 1111–1126, June 2013.

[28] M. Atamaner, O. Ergin, M. Ottavi, and P. Reviriego, "Detecting errors in instructions with bloom filters," in *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–4, Oct 2017.

[29] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Grkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 2700–2713, Oct 2017.

[30] R. Hller, D. Haselberger, D. Ballek, P. Rssler, M. Krapfenbauer, and M. Linauer, "Open-source risc-v processor ip cores for fpgas overview and evaluation," in *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–6, June 2019.