A Lightweight Security Checking Module to Protect Microprocessors against Hardware Trojan Horses

Alessandro Palumbo[a], Luca Cassano[b], Pedro Reviriego[c], Giuseppe Bianchi[a], Marco Ottavi[a]
[a]University of Rome Tor Vergata, Italy, [b]Politecnico di Milano, Italy, [c]Universidad Carlos III de Madrid, Spain,
[a]{name.surname}@uniroma2.it, [b]luca.cassano@polimi.it, [c]revirieg@it.uc3m.es

***Abstract*—It has been demonstrated that Software exploitable Hardware Trojan Horses (HTHs) can be inserted in commercial CPUs and memories. Such attacks allow malicious users to run their own software or to gain unauthorized privileges over the system. As a consequence, HTHs must nowadays be considered a serious threat not only from academy but also from industry. In this paper we present a security checking module meant to be connected between the microprocessor and the instruction memory in order to monitor the fetching activity with the aim of detecting the activation of HTHs. In particular, we aim at detecting those HTHs that alter the expected execution flow by launching a malicious program. We integrated the proposed security checking module within a case study system based on a RISC-V microprocessor implemented on an FPGA and running a set of software benchmarks. This experiment demonstrated that our proposal is able to detect 100% of possible HTHs activations with no false alarms. We measured a LUT overhead of 0.5% and a FF overhead of 0.3%, with a 2.36% power consumption increase and no working frequency reduction.**

***Index Terms*—Hardware Security, Hardware Trojan Horses, Malicious Software, Microprocessor-based System, RISC-V**

I. Introduction and related work

The dramatic complexity of modern integrated circuits (ICs) and the continuous seek for low production cost and short time-to-market, has led to a globalized design and fabrication process [1]. More and more often the design of several hardware modules is outsourced, third-party intellectual property cores (3PIPs) are purchased, masks are also outsourced and the final chip is fabricated by third party foundries [2]. Such a globalization allows for a significant reduction of design cost and time, at the cost of a significant loss of trust in the delivered ICs [3].

It is all but impossible to ensure the trustworthiness of all the entities involved in such a globalized supply chain. As a consequence, the produced system is exposed to a number of threats, among which overproduction [4], counterfeiting [5], license violation and abuse [6] and Hardware Trojan Horses (HTHs) insertion [7]. From a very high-level point of view, a HTH is a very hard-to-detect modification of a design that is meant to stay hidden most of the time, while in specific (usually rare) conditions it alters the nominal behavior of the system or it steals sensitive information. A produced system may be infected by HTHs belonging to 3PIPs providers [8], employees or malicious CAD tools [9] and mask providers and silicon foundries [10].

In the past, HTHs have been considered an issue more by academy than by industry because of the difficulty of insertion in real-world circuits and the limited advantages the attacker could count on. Nevertheless, in the last years it has been demonstrated that complex *software-exploitable HTHs* can be inserted in real-world commercial microprocessors. Thanks to this class of more powerful HTHs, the attacker is able to execute his/her own malicious software, to modify the running software or to acquire root privileges [11]–[13]. Finally, in 2018, a HTH, called the *Rosenbridge* backdoor, has been found in a commercial Via Technologies C3 processor [14]. The Rosenbridge backdoor could be activated via software and allowed the attacker to enter in supervisor mode.

A number of techniques to detect HTHs before system deployment have been proposed. They are generally *circuit-level* techniques that aim at detecting HTHs at design time via logic testing [15], formal property verification [16], side-channel analysis [17], structural and behavioral analysis [18], [19]. On the other hand, given the extreme stealthy nature of HTHs and the huge amount of resources available in a modern integrated circuits among which a HTH can be hidden, it is extremely hard to detect HTHs before the system has been deployed. There is therefore a growing interest in *system-level* techniques that allow to obtain a trusted system built with untrusted components [20]–[22]. A similar paradigm has been proposed in [23], [24] where the focus is on microprocessor-based systems and the goal is to enable a trusted software execution on an untrusted CPU. Finally, very recently also HTHs in memories have been studied [25]. At the same time, few work has been devoted to design methodologies to protect a microprocessor from HTHs inserted in memory chip [26].

In this paper we propose a system-level solution for protecting microprocessor-based systems against HTHs. More in details, we integrate a security checker between the microprocessor under protection and the instruction memory. Such security checker is *programmed* while installing a program in the instruction memory of the system. In particular, the checker stores information about the instructions that compose the program and the memory locations in which the program is installed. Then, at runtime, the checker is in charge of monitoring the fetching activity of the microprocessor to check whether the right instructions are being loaded and from the right memory locations. In this way, our checker is able to detect the runtime activation of HTHs infesting the microprocessor itself, the instruction memory or the bus and aiming at forcing the microprocessor to run a malicious program by fetching unauthorized instructions or by reading

unauthorized memory locations. 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 any interruption of the code execution.

We integrated the proposed security checking module within a case study system based on a RISC-V microprocessor implemented on an FPGA and running a set of software benchmarks. This experiment demonstrated that our proposal is able to detect 100% of possible HTHs activations with no false alarms. We measured a LUT overhead of about 0.5% and a FF overhead of about 0.3%, with a 2.36% power consumption increase and no working frequency reduction.

The works related to our proposal are the system-level design-for-trust methodologies proposed in [23], [24], [26]. Unlike in our proposal, in [23], [24] 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 the expected one. 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 HTHs that change the functionality of the system by making the CPU run normal instructions without changing privilege mode. In other words, none of these works checks whether the microprocessor is executing an unwanted software and whether it is accessing illegal memory locations (as we do in the current paper). Finally, the work we consider the most similar to our proposal is the one in [26]: in this paper a checker to detect the activation of HTHs infesting the main memory has been proposed by the same authors of the current paper. The solution relied on a Bloom filter thus exposing a probabilistic behaviour. As it will be demonstrated in the experimental section, the checker proposed in the current paper outperforms the one in [26] both in terms of accuracy (w.r.t. both detection capability and false alarm rate) and overhead.

The remainder of this paper is organized as follows: Section II presents the models of HTHs that are targeted by our proposal; Section III presents the proposed methodology, discussing the details of the checker on which it relies; Section IV highlights results from a case study application of the proposed solution to a RISC-V based system running a set of benchmark programs, while Section V presents the security analysis; Section VI concludes the paper.

II. The Considered Threat Model

In this work we consider HTHs that aim at changing the functionality of the system by forcing the CPU to execute an unwanted program. Therefore, our main target are those HTHs infesting the fetching unit of the core. Indeed, for a HTH infesting the fetching unit it would be enough to force the program counter to point to an instruction memory location where the malicious program has been loaded. For the same reason, target HTHs may be those infesting the instruction memory and the system bus of the system. Indeed, also these HTHs may alter the pointed instruction memory

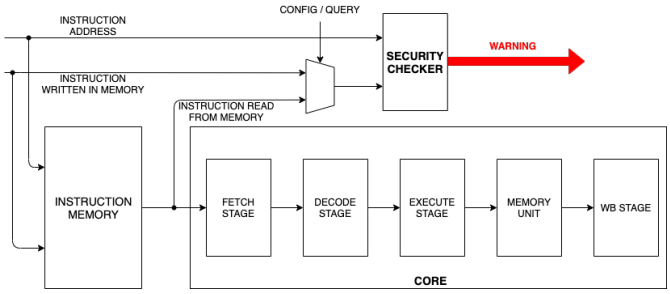


Figure 1: The proposed protection architecture

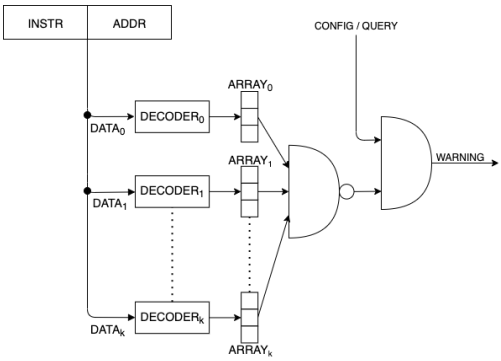


Figure 2: The structure of the proposed Security Checker

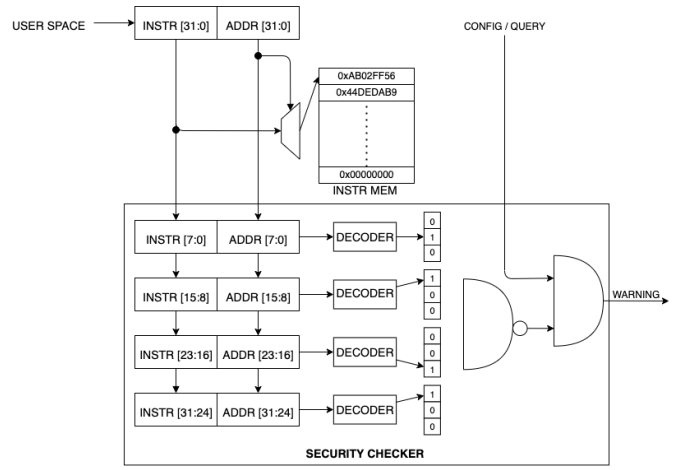
location, thus allowing to launch a malicious program. We do not make any assumption on the triggering mechanism of the infesting HTH. We assume that, when injecting the HTH at design- or fabrication-time, the attacker knows all the details of the hardware platform he/she is attacking. Moreover, we assume that the attacker has an idea about which operating system and programs will be executed but, on the other hand, he/she cannot have all the details about software versions and implementations. From the HTH insertion point of view, since the proposed detection methodology works at runtime, it is able to detect HTHs that have been inserted during any stage of the design process and by any actors taking part in the design and supply chain of the system. By summarizing, possible attack scenarios considered by our solution are:

- A HTH in the microprocessor that alters the content of the program counter;
- A HTH in the instruction bus that modifies the memory address required for instruction fetch;
- A HTH in the instruction memory that forces the memory to access an incorrect location;

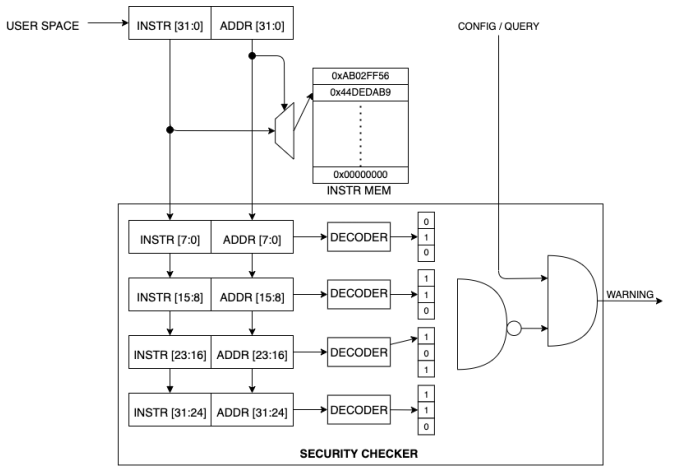
Finally, it is worth mentioning that denial-of-service and information stealing HTHs, on the other hand, fall outside the scope of this paper.

III. The Proposed Security Checker

We propose the architecture depicted in Figure 1 where a *Security Checker (SC)* is inserted between the microprocessor and the instruction memory to protect the system against HTHs that try to force the execution of malicious programs. More



(a) Writing the first program instruction



(b) Writing the second program instruction

Figure 3: Configuring the Security Checker during program installation into the instruction memory

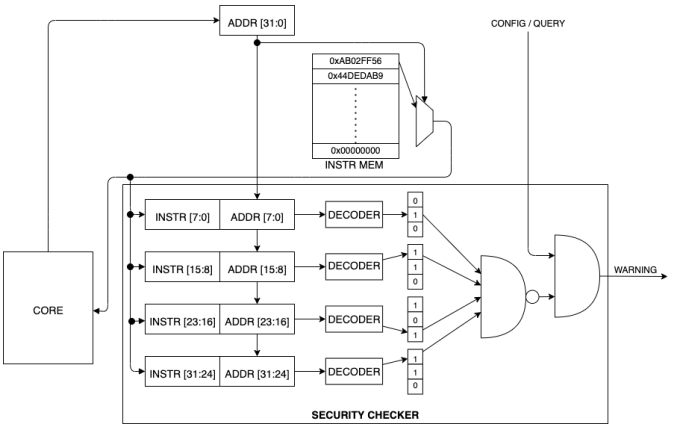


Figure 4: Querying the Security Checker during program execution

in details, the SC is configured during the installation of the program(s) that the system will execute, then, at runtime, the SC monitors the fetching activity of the microprocessor with the aim of detecting and signalling the activation of a HTH. On the other hand, the management of the warning, e.g., a non-maskable interrupt, by the overlying operating system does not fall into the scope of this work.

During each program installation, i.e., while the program is loaded in the instruction memory, the proposed security architecture will work in *configure* mode. In this working mode the SC is configured with the instructions that compose the program and with the instruction memory addresses in which each instruction is loaded. On the other hand, at runtime, while the program(s) is/are running, the security architecture will work in *query* mode. In this working mode, after every instruction read from the memory location required by the microprocessor, and based on the previously configured information, the SC checks the accessed instruction memory address and the fetched instruction. In particular, the SC checks whether the accessed address is legal, i.e., it belongs to the memory space of the program under execution, and whether the fetched instruction is exactly the one that was loaded in that specific memory address during program installation.

As we will experimentally demonstrate, the proposed SC module achieves extremely high detection accuracy with a very limited overhead in terms of area occupation, power consumption and working frequency reduction.

A. The Security Checker architecture

The architecture of the proposed *Security Checker (SC)* is depicted in Figure 2. The SC takes in input a memory address, an instruction and the `CONFIGURE/QUERY` signal (that specifies whether the SC is working in configure or in query mode) and produces as output a warning. When working in configure mode, both the address and the instruction come from the user space that is installing a program in the instruction memory of the system; on the other hand, when working in query mode, the address comes from the core while the instruction comes from the instruction memory. A combination of address and instruction (that will be presented in the next subsection) is then used, both when configuring and when querying the SC, to address a number of bit arrays within the SC. We refer to k as the number of bit arrays in the SC and we call it the *fragmentation factor*. The content of these k bit arrays is set at configuration time to keep track of all the address-instruction pairs legal for the program that is going to be installed. At query time the content of these bit arrays is read to check whether the current address-instruction pair is legal or not. As it can be noticed from Figure 2 a warning is raised if at least one of the accessed bit array locations is set to 0 and if the SC is working in query mode.

B. The Security Checker configuration and usage

As we previously said, the SC takes in input an address and an instruction. Such two input data are combined within the

SC and then fragmented into a number of data chunks ($\texttt{DATA}_0$ up to $\texttt{DATA}_k$ in Figure 2). In particular, being n the size in bit of addresses and instructions in the considered architecture, $\texttt{DATA}_0$ is composed of the first n/k bits of the address paired with the first n/k bits of the instruction, $\texttt{DATA}_1$ is composed of the second n/k bits of the address paired with the second n/k bits of the instruction and so on. The produced bit groups are then decoded and used to access specific locations of a number of bit arrays.

When working in configure mode, both the address and the instruction come from the user space that is installing the program in the instruction memory of the system. After pairing the address and the instruction and producing the k data chunks such chunks are used as memory addresses to access the corresponding bit arrays. In particular, a 1 is written in each bit array location addressed by the corresponding chunk to *teach* to the SC that the specific address-instruction pair is legal for the program.

Figure 3 depicts an example configuration procedure for two consecutive example instructions in a system having 32 bit long addresses and instructions and where the SC has a fragmentation factor of 4. It is worth observing that for different address-instruction pairs one or more data chunks may point to the same location of the corresponding bit array. This is the case of the first data chunk in the example that points to the middle bit in both Subfigure 3a and 3b. This does not represent a problem, i.e., does not lead to false alarms (as we will experimentally demonstrate in the next section), as those are triggered when no address-instruction pair in the program maps to the positions selected.

When working in query mode, the address comes from the core and, after reading the instruction memory, the address come from the instruction memory itself. The k data chunks are generated exactly as previously described but in this phase the content of the bit arrays is read. As soon as at least one of the read values is 0, the SC raises an alarm. Figure 4 depicts an example query procedure for an example address in a system having 32 bit long addresses and instructions and where the SC has a fragmentation factor of 4.

IV. Experimental Results

A. Experimental setup

For our experimental campaign we considered the PULPINO architecture which is an ultra-low-power 32 bit processing platform mainly targeted to Internet of Things applications [27]. We considered the RI5CY [28] version of PULPINO, which is a small 4-stage RISC-V core. When synthesized on a Xilinx Artix XC7A35T, RI5CY requires 15097 LUTs and 9881 FFs and it works at about 50MHz with a total power consumption of 127mW (21mW dynamic power consumption on average), as reported in [29]. 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`.

Table I: The considered benchmark programs

Benchmark	#Instructions
`Binary Search (BinS)`	215
`Matrix Multiplication (MM)`	216
`Bubble Sort (BubS)`	268
`Quick Sort (QS)`	1023
`Sudoku Solver (SS)`	475
`Motion Detection (MD)`	934

Table II: FP and FN rates when the HTH modifies the accessed instruction memory location

Bench.	Our proposal		Proposal in [26]	
	FP	FN	FP	FN
`BinS`	0%	0%	0%	0.523%
`MM`	0%	0%	0%	0.520%
`BubS`	0%	0%	0%	0.572%
`QS`	0%	0%	0%	0.607%
`SD`	0%	0%	0%	0.249%
`MD`	0%	0%	0%	0.912%
AVG	0%	0%	0%	0.663%

When designing the checker we started with a fragmentation factor k of 1, that makes the checker require a 2^{64} bits memory, which is of course totally unfeasible. For the same reason, also a checker with a $k = 2$ (that would require two 2^{32} bits memories) can be considered unfeasible for an embedded system. Solutions having $k = 8$ or greater (eight 2^8, sixteen 2^4 bits memories and so on) achieved extremely poor accuracy and, for this reason, we do not even report the numbers in the paper (we will only draw some considerations at the end of this section). Therefore, the only feasible checker configuration that provides acceptable accuracy (whose results will be presented in the remainder of this section) is the one having $k = 4$, thus requiring four 2^{16} bits memories. After having identified the target checker configuration, we integrated the checker in the considered processing architecture as described in the previous section.

First of all we aimed at assessing the effectiveness of the proposed checker in detecting the activation of HTHs and not in raising false alarms when no HTH activated. In the remainder of this section we will refer as *false negatives (FNs)* to those cases where the HTH activated but our checker did not detect it; similarly, we will refer as *false positives (FPs)* to those cases where no HTH activated but our checker raised a false alarm. We emulated the activation of a HTH belonging to the previously presented models by modifying at a random time the instruction memory address from which the microprocessor fetches an instruction. In particular, in order to emulate a HTH that makes the microprocessor run a malicious program, we force the selected memory address to be outside the memory space of the legal program. We simulated 10,000 randomly generated HTH activation cases and we measured the FN rate as the number of runs in which the checker did not raise an alarm over the total number of runs. Similarly, we simulated 10,000 runs in which no HTH activated and we measured the FP rate as the number of runs in which the checker raised an alarm over the total number

Table III: Resource occupation and working frequency of our proposal and of the one in [26]

Bench.	Our proposal				Proposal in [26]			
	#LUTs	#FFs	BRAM size	Freq. (MHz)	#LUTs	#FFs	BRAM size	Freq. (MHz)
BinS	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
MM	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
BubS	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
QS	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	880 (5.83%)	84 (0.85%)	32 KBit	112 MHz
SS	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	1539 (10.19%)	89 (0.90%)	64 KBit	106 MHz
MD	75 (0.49%)	31 (0.31%)	208 Kbit	275 MHz	1539 (10.19%)	89 (0.90%)	64 KBit	106 MHz

of runs. Results from this experiment are reported in Table II, where we compare our proposal with the one in [26]. First of all, it is worth mentioning that our proposal always exposes both 0% FP and FN rates; on the other hand, the proposal in [26] (which is based on Bloom filters) guarantees 0% FP rate but a not null (but configurable) FN rate. As a final effectiveness experiment, we wanted to analyse the scalability of the proposed solution w.r.t. the size of the program under execution. To do so, we repeated the same experiment with a program counting 10,000 instructions (one order magnitude larger the the previously considered benchmarks). The result of this experiment has been again 0% FP and 0% FN rates, thus demonstrating that the chosen checker configuration is optimal also for larger programs. On the other hand, one could think that the chosen checker configuration could be overdimensioned for small programs. Therefore, we repeated the previous experiment with a program counting only 50 instructions and with a checker having eight 2^8 bits memories ($k = 8$). Under this configuration, the proposed checker achieved 0% FP and 12.97% FN rates, while the checker with four 2^{16} bits memories ($k = 4$) achieved again 0% FP and 0% FN rates. This experiment allowed us to argue that ($k = 4$) is the optimal configuration also for small programs.

We then evaluated the overhead introduced by the proposed checker in terms of used resources, working frequency and power consumption increase when targeting an FPGA implementation. Table III reports the number of LUTs and FFs and the amount of BRAM bits required by our proposal and by the proposal in [26] as well as the maximum working frequency that would be imposed by the presence of the checkers in the system. First of all, it may be noticed that the overhead introduced by our checker is independent of the executed program, while when considering the Bloom filter-based checker in [26], the larger the program, the larger the checker itself. It is worth noting that while the overhead in terms of additional FFs is very similar between the two solutions (still lower in the current one), the overhead in terms of additional LUTs is negligible in our solution while it reaches about 10% in [26]. On the other hand, our solution requires much more BRAMs than the one in [26]. Looking at the working frequency overhead, since the considered microprocessor works at 50 MHz, neither our solution nor the one in [26] have an impact. Finally, concerning the power consumption overhead, the considered microprocessor protected with the proposed checker has a total power consumption of 130 mW (25 mW dynamic power consumption) thus, we introduce a 2.36% power consumption increase, which we believe is totally acceptable. No power consumption data was reported in [26].

Table IV: FP and FN rates when the HTH modifies the fetched instruction

Bench.	Accuracy	
	FP	FN
BinS	0%	2.25%
MM	0%	0.40%
BubS	0%	3.01%
QS	0%	3.91%
SD	0%	0.72%
MD	0%	2.83%
AVG	0%	2.18%

V. Security analysis

The presented experimental results demonstrate that the proposed security checker allows to detect 100% of the runtime activations of HTHs that try to force the CPU to execute malicious programs installed in instruction memory locations outside the memory space of the running program. Furthermore, as demonstrated by the reported experiments, the proposed checker never incurs in false alarms. It is worth mentioning that, as it has already been discussed, the effectiveness of the proposed solution is independent of the triggering mechanism of the HTH, i.e., combinationally/sequentially triggered, externally activated, time-bombs and always-on, and of the design stage during which the HTH has been inserted.

A hypothetical threat for our protection system would be a HTH in the microprocessor, instruction bus or instruction memory that does not modify the requested instruction memory location but that alters the fetched instruction after reading the correct memory location. In this scenario, in order to be able to force the microprocessor to run a malicious program, the HTH should be able to modify the fetched instruction of a number of consecutive fetch operations (as many as the number of instructions composing the malicious program). Of course, such malicious instructions should be either hard-coded in the HTH itself or accessible from specific memory locations by the HTH. We believe that such attack is much harder to be implemented than a HTH that modifies the content of the program counter, and thus, it represents a minor threat. Nevertheless, we analysed the effectiveness of our proposal in detecting this kind of attack. In particular, we ran 10,000 times each benchmark program and in each run we emulated the activation of this kind of HTH by leaving unaltered the requested instruction memory address and by substituting the

fetched instruction with a random instruction after having accessed the instruction memory itself. Results from this analysis are reported in Table IV. As it can be observed, our checker never raises false alarms while, on average, 2.18% of the HTH activations are not detected. Given the previously discussed difficulty in deploying such attack, we believe that the achieved results can be considered reasonable.

On the other hand, the proposed solution could be defeated by denial-of-service HTHs that modify the execution flow of the legal program. We identified two possible scenarios: i) HTHs that halt the system by maliciously making the CPU fetch always the same legal instruction (or sequence of legal instructions) from memory locations belonging to the authorized program; and ii) HTHs that halt the system by making it crash by fetching a legal instruction from a memory locations belonging to the authorized program but at the wrong time or in the wrong order, e.g., fetching a jump instruction too early during the execution flow. These attack conditions (that, as discussed in the threat model fall outside the scope of our solution) cannot be detected by the proposed security checker but they can be managed by providing the system with ad-hoc dimensioned watchdogs. Further, by exploiting watchdogs that monitor the fetching activity of the processor, the proposed methodology could detect denial-of-service HTHs that freeze the CPU. Finally, HTHs that steal information by sending it through covert side-channel are still able to defeat the proposed solution.

VI. Conclusion

We presented a security checking module to protect microprocessor-based systems against those hardware Trojan horses that try to force the system to run a malicious program. We integrated the proposed solution within 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 able to detect 100% of possible HTHs activations with no false alarms. We measured a LUT overhead of about 0.5% and a FF overhead of about 0.3%, with a 2.36% power consumption increase and 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. Hély, 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] A. Bolat, L. Cassano, P. Reviriego, O. Ergin, and M. Ottavi, "A microprocessor protection architecture against hardware trojans in memories," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–6, 2020.

[27] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "Pulpino: A small single-core risc-v soc," in *3rd RISCV Workshop*, 2016.

[28] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, 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.

[29] R. Höller, D. Haselberger, D. Ballek, P. Rössler, 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.