2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) | 979-8-3503-1500-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/DFT59622.2023.10313563

Improving the Detection of Hardware Trojan Horses in Microprocessors via Hamming Codes

Alessandro Palumbo[a], Luca Cassano[a], Pedro Reviriego[b], Marco Ottavi[c]
[a]Politecnico di Milano, Italy, [b]Universidad Politécnica de Madrid, Spain,
[c]University of Rome Tor Vergata, Italy and University of Twente, The Netherlands
[a]{name.surname}@polimi.it, [b]pedro.reviriego@upm.es, [c]ottavi@ing.uniroma2.it

***Abstract*—Software-exploitable Hardware Trojan Horses (HTHs) can be inserted into commercial microprocessors allowing the attackers to run their own software or to gain unauthorized privileges. As a consequence, HTHs should nowadays be considered a serious threat not only by the academy but also by the industry. In this paper we present a hardware security checker for the detection of the runtime activation of HTHs. In particular, we aim at detecting HTHs that alter the expected execution flow by launching a malicious program. To achieve this goal the proposed checker is connected between the microprocessor and the main memory and observes the fetching activity. We integrated the proposed checker within a case study based on a RISC-V microprocessor running a set of software benchmarks. The experiment demonstrated that our checker is able to detect 100% of possible HTHs activations with no false alarms. We measured an area overhead of less than 1% w.r.t. LUTs and FFs with 8.5 up to 9.5 BRAM blocks required, a 2.51% power consumption increase, and no working frequency reduction.**

***Index Terms*—Hamming Codes, Hardware Security, Hardware Trojan Horses, Microprocessor-based System, RISC-V**

I. Introduction and related work

The continuous seek for low production cost and short time-to-market has shifted the design and fabrication of modern integrated circuits (ICs) to a globalized process [1]. The design of sub-components/systems is outsourced, third-party intellectual property cores (3PIPs) are purchased and the final chip is fabricated by external foundries [2]. This brought a huge design time and cost reduction but also a significant loss of trust in the delivered ICs [3]. Ensuring the trustworthiness of all the entities involved in such a globalized supply chain has become unfeasible; as a consequence, several threats may affect the system. ICs may be overproduced and counterfeited, licenses may be violated and abused and Hardware Trojan Horses (HTHs) may be inserted.

HTHs are very hard-to-detect modifications of a system that are meant to stay hidden most of the time and to alter the nominal behavior of the system or to steal sensitive information in specific (usually rare) conditions [4]. HTHs may be inserted by malicious 3PIPs providers, employees or CAD tools, and mask providers, and silicon foundries. HTHs have always been considered more an academic issue because of the difficulty of insertion in real-world systems that led to reduced advantages for the attacker. Recently, it has been demonstrated that complex *software-exploitable HTHs* can be inserted in real-world commercial microprocessors. Such HTHs allow the attacker to execute his/her own malicious software, to modify the running software, or to acquire root privileges [5]–[7]. In 2018, the *Rosenbridge* backdoor, has been found in a commercial Via Technologies C3 processor [8]. A specific sequence of instructions allowed the attacker to activate the Rosenbridge backdoor and enter the supervisor mode[1].

Several techniques for the design-time detection of HTHs have been proposed. The idea is to analyse the system, generally at the circuit-level, before its deployment by exploiting logic testing [9], formal property verification [10], side-channel analysis [11], structural and behavioral analysis [12], [13] and machine learning [14]. Nevertheless, it has to be considered that HTHs are stealthy by nature, therefore it is extremely hard to detect them before deployment. On the other hand the *Design-for-Trust* paradigm raised: there is a growing interest in *system-level* techniques that allow to obtain a trusted system built from untrusted components [15], [16]. Similarly, HTHs may be defeated by enabling a trusted software execution on an untrusted microprocessor-based system [17], [18].

In this paper, we propose a system-level solution for the runtime detection of HTHs in microprocessor-based systems and forcing the microprocessor to run a malicious program. The proposed solution relies on the integration of a hardware security checker between the microprocessor under protection and the main memory in order to monitor the fetching activity. The security checker is *programmed* during the installation of the program in the main memory: information about the instructions that compose the program and the memory locations in which the program is installed are used by the checker during this phase. At runtime, the checker is in charge of monitoring the fetching activity to check whether the right instructions are loaded from the right memory addresses. By doing so, the security checker detects the runtime activation of HTHs infesting the microprocessor logic, the main memory, or the bus. The proposed solution is completely transparent w.r.t. the nominal functioning of the microprocessor. Indeed, the security checker works without neither interrupting or interfering with the execution of the program under protection.

We integrated the security checker in a case study system based on a RISC-V microprocessor implemented on an FPGA

[1]Via Technologies officially commented that this behavior was due to an undocumented feature meant for debugging.

and running a set of software benchmarks. The experiment demonstrated that our checker is able to detect 100% of possible HTHs activations with no false alarms. We measured an area overhead of less than 1% w.r.t. LUTs and FFs with 8.5 up to 9.5 BRAM blocks required, a 2.51% power consumption increase, and no working frequency reduction.

The works more closely related to ours are the system-level design-for-trust methodologies proposed in [17]–[20]. Unlike in our proposal, in [17], [18] the microprocessor is assumed to be untrusted and the memory to be trusted. In [17] the presented checker observes whether the opcodes and associated control signals are legal or not and whether the number of clock cycles required to complete the execution of an instruction is compliant with the expected one. In [18] the security checker keeps track of the liveness of the microprocessor and of the privilege mode in which it is running. On the other hand, none of the previous solutions target those HTHs that make the CPU run normal instructions without changing the privilege mode. More in details, none of these works is able to detect those cases where the microprocessor is forced to execute an unexpected program or where the microprocessor is forced to access illegal memory locations (as we do in the current paper). The works that we consider the most similar to the current proposal are the ones in [19], [20]: in these papers, a checker to detect the activation of HTHs infesting the microprocessor, the bus and the main memory have been proposed. The solution in [19] introduces a large area overhead still suffering from a not-zero false negative rate. The solution in [20] overperforms the previous one from the overhead point of view but it leaves room for attacks where the accessed address is not altered but the fetched instruction (either opcode or operands) is maliciously modified. The solution proposed in the current paper borrows the principles from the previous two works and adds Hamming codes to overcome the previously mentioned security limitations while requiring limited area and power overhead (and no working frequency reduction).

This paper is organized as follows: Section II presents the HTHs models targeted by our proposal; Section III presents the proposed HTH detection methodology and the details of the security checker on which it relies; Section IV highlights results from a case study application, while Section V discusses the security analysis; Section VI concludes the paper.

II. HTH threat model

We consider change-functionality HTHs that force the CPU to execute an unwanted program. An HTH may reside in the microprocessor and may change the content of the program counter to force the fetch unit to access instruction memory locations where a malicious program has been loaded. Similarly, HTHs may infest the instruction memory or the system bus altering the pointed instruction memory locations, thus again allowing to launch a malicious program.

Referring to the classical classification of HTHs [4], we do not make any assumption on the triggering mechanism of the infesting HTH, i.e., our proposal takes into account both triggered and always-on HTHs. The considered HTHs

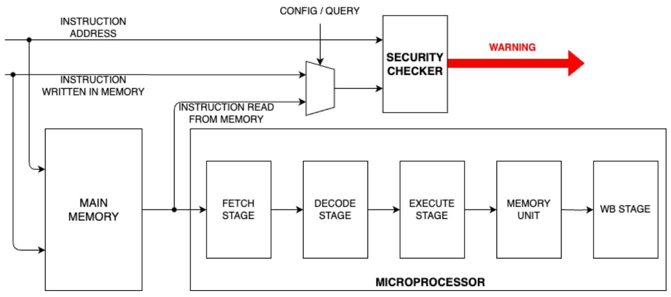


Figure 1: The proposed protection architecture

are supposed to be inserted by a malicious IP provider; on the other hand, we assume that the design team of the company and the employed foundry are trusted, therefore, we the introduced security checker is assumed to be trusted. Since the attacker is the IP provider, we assume that, when injecting the HTH, the attacker knows all the details of the hardware platform he/she is attacking. To summarize, the possible attack scenarios considered by our proposal are:

- A HTH in the microprocessor that alters the content of the program counter;
- A HTH in the bus or in the main memory that modifies the memory address required for instruction fetch;
- A HTH in the bus or in the main memory that modifies the instruction fetched from the right memory address;

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

III. The Proposed Security Solution

Our proposal relies on adding a *Security Checker (SC)* between the microprocessor and the instruction memory (as depicted in Figure 1) to detect HTHs that try to force the execution of malicious programs. The SC is first configured during the installation of the program(s) that the system will execute; then, during program(s) execution at runtime, the SC monitors the fetching activity to detect and signal the activation of an HTH[2].

While each program is loaded in the instruction memory, the SC works in *configure* mode: the SC is *configured* with the instructions that compose the program and with the memory addresses where the instructions are stored. Then, during program(s) execution at runtime, the SC switches into *query* mode: after every instruction has been fetched the SC checks whether the accessed address and the fetched instruction are compliant based on the previously configured information. More in details, the SC checks whether the accessed address belongs to the memory space of the program under execution and whether the fetched instruction is the one that was stored in that memory address during program installation.

A. The Architecture of the Security Checker

The architecture of the *Security Checker (SC)* is drawn in Figure 2: it takes as input a memory address, an instruction,

[2]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.

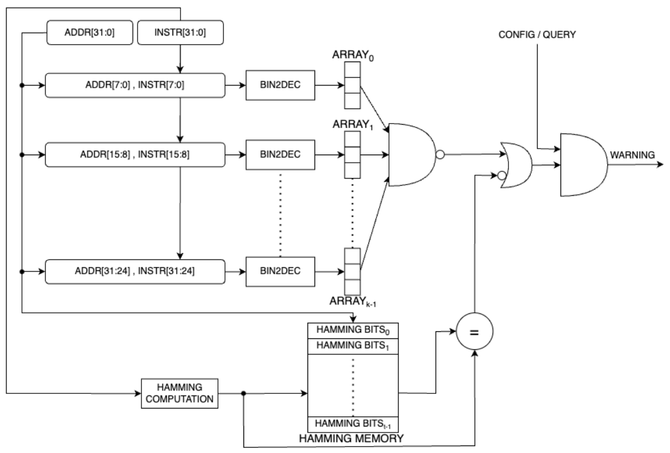


Figure 2: The structure of the proposed Security Checker

and the CONFIGURE/QUERY signal (that specifies whether the SC is working in configure or in query mode) and produces as output a warning signal. When the SC works in configure mode, both the address and the instruction come from the user space, i.e., a user is installing a program in the memory of the system. When the SC works in query mode, the address comes from the microprocessor that is fetching while the instruction comes from the main memory.

The input address and instruction are combined together (in a way that will be presented in the next subsection) and then used, both when configuring and when querying the SC, to address a number of bit arrays within the SC. We call k (the *fragmentation factor*) the number of bit arrays in the SC. The content of specific entries of these k bit arrays is set to 1 at configuration time to keep track of all the address-instruction pairs that are legal for the program under installation. At query time, the content of these bit arrays is read to check whether the current address-instruction pair is legal or not.

Moreover, at configuration time a single error correction Hamming sequence is calculated for every installed instruction. These Hamming bits are then stored in small an ad-hoc *Hamming memory* (every instruction of the program has a dedicated address in the Hamming memory). At query time, the Hamming bits of the fetched instruction are calculated and compared with the Hamming bits that have been stored at configuration time in the Hamming memory address associated with the address from which the microprocessor fetched the current instruction. By adding these Hamming bits we solve the problem that the solution in [20] had left unsolved, i.e., the current solution is also able to detect those HTHs that do not modify the accessed memory address (which will therefore be a legal address) but that directly tamper the fetched instruction.

When the SC is in query mode a warning is raised if at least one of the accessed bit array locations is set to 0 or the calculated Hamming code mismatches with the stored one.

B. Configuration and Usage of the Security Checker

The SC takes in input an address and an instruction; such two inputs are combined within the SC and then fragmented into a number of data chunks ($\texttt{DATA}_0$ up to $\texttt{DATA}_{k-1}$ in Figure 2). More in details, if we call n the size in bits of addresses and instructions in the considered architecture, $\texttt{DATA}_0$ is a data chunk composed of the first n/k bits of the address paired with the first n/k bits of the instruction. Similarly, $\texttt{DATA}_1$ is a data chunk composed of the second n/k bits of the address paired with the second n/k bits of the instruction and so on. These groups of bits are then decoded and used as addresses to access a number of bit arrays, one for each data chunk. At the same time, the SC contains a memory, dubbed the *Hamming memory*, meant to store the Hamming code of each of the t instructions of the previously installed program.

When the SC is in configure mode, both the address a and the instruction i come from the user space that is installing the program. The values of a and i are combined and the k data chunks are produced: such chunks are then used as memory addresses to access the corresponding bit arrays. More in details, a 1 is written in each bit array location addressed by the corresponding chunk to *teach* to the SC that the tuple $< a, i >$ is legal for the program. Moreover, the Hamming code associated with i is calculated and it is stored at address a in the Hamming memory within the SC.

Figure 3 shows an example configuration procedure for the first two instructions of the program. It is worth mentioning that for different address-instruction pairs one or more data chunks may point to the same location of the corresponding bit array (the so called *collision*). This is the case of the last data chunk in the example that points to the last bit of the bit array in both Subfigures 3a and 3b. Moreover, as it can be seen in the bottom of Subfigures 3a and 3b, for every instruction installed in the main memory, the Hamming code is calculated and stored in the corresponding address of the Hamming memory.

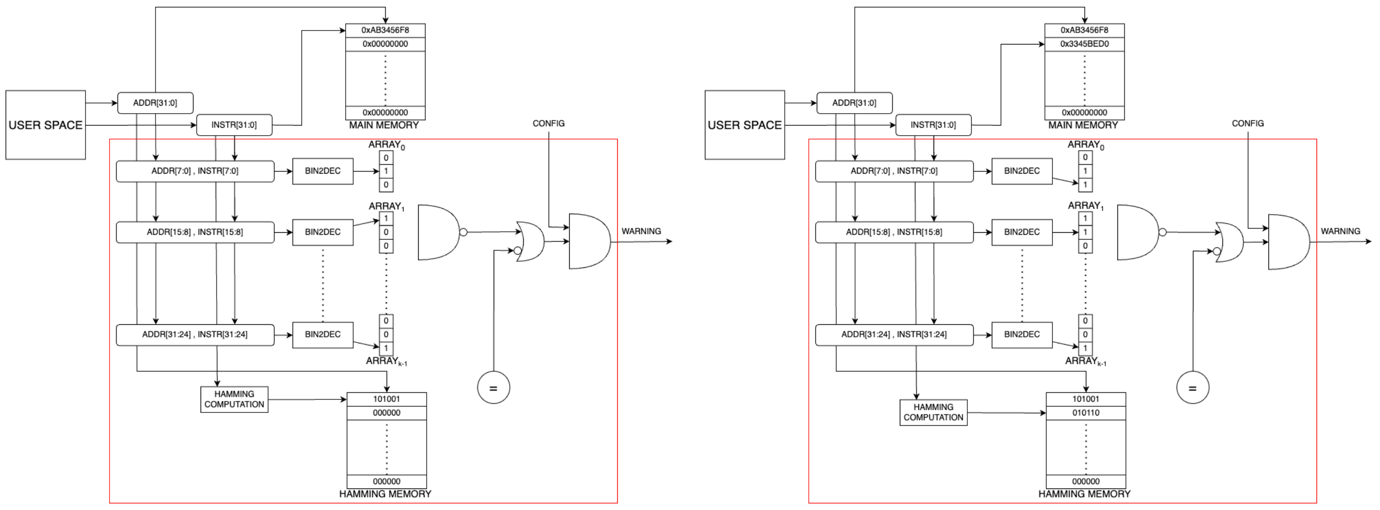
When the SC is in query mode, the address comes from the microprocessor and, after reading the main memory, the instruction comes from the main memory itself. The k data chunks are generated exactly as it has been previously described but in this phase, the content of the bit arrays is read (while it is written in configure mode). As soon as at least one of the values read from the bit arrays is 0, the SC raises an alarm. Moreover, in this phase, the SC calculates the Hamming code of the instruction coming from the main memory and compares it with the Hamming code stored in the Hamming memory at the address coming from the microprocessor. As soon as this comparison produces a mismatch, the SC raises an alarm. Figure 4 depicts an example query procedure for the very first instruction of the program.

IV. Experimental Results

A. Experimental setup

In our experimental campaign, we considered the RI5CY [21] of the PULPINO architecture which is an ultra-low-power 32-bit processing platform mainly targeted to the Internet of Things applications [22].

When synthesized on a Xilinx Artix XC7A35T, RI5CY requires 15314 LUTs and 9881 FFs and it works at about



(a) Writing the first program instruction (b) Writing the second program instruction

Figure 3: Configuring the Security Checker

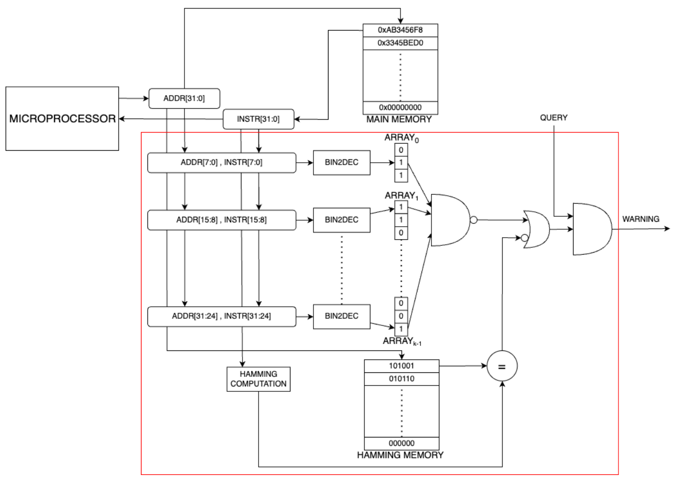


Figure 4: Querying the Security Checker

50MHz with a total power consumption of 159mW (17mW dynamic power consumption on average), as reported in [23]. Finally, we considered the benchmark programs reported in Table I together with the number of assembly instructions.

We first analysed how the choice of the value of the fragmentation factor k affected accuracy and overhead. We started with $k = 1$, thus requiring a 2^{64} bits memory, which is of course totally unfeasible. For the same reason, also a checker with a $k = 2$ (with two 2^{32} bits memories) has been discarded, being this choice unfeasible for an embedded system. On the other hand, SC configurations having $k = 8$ or greater (eight 2^8 bits, sixteen 2^4 bits memories, and so on) achieved extremely poor accuracy and, for this reason, have also been discarded. Therefore, the only feasible checker configuration that provides acceptable accuracy is the one having $k = 4$ with four 2^{16} bits memories: the results reported in the following of this section always refer to this SC configuration. After having identified the target configuration, we integrated the SC in the previously described case study processing architecture.

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
`Coremark (CM)`	1288
`Median (MED)`	1026
`Towers (TW)`	350
`RSort (RS)`	4466

As a first experiment, we assessed the effectiveness of the proposed checker in detecting the activation of HTHs and, at the same time, not in raising false alarms when no HTH is activated. We refer to *false negatives (FNs)* as those cases where the HTH activated but the checker did not detect it and to *false positives (FPs)* as those cases where no HTH activated but the checker raised a false alarm. More in details, we wanted to analyse the capability of our checker in detecting both those HTHs that modify the accessed memory address but also those HTHs that directly tamper the fetched instruction after fetching from the expected memory address. In particular, this last HTH category was the one that limited the accuracy of the proposal in [20]

We first emulated the activation of HTHs by modifying at a random time the instruction memory address from which the microprocessor fetches. Therefore, we emulated HTH able to hijack the execution flow towards a new program. In particular, we forced the selected memory address to be outside the memory space of the program under execution. We simulated 10,000 randomly generated HTH activation cases and we measured the FN rate as the number of runs in which the

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

Bench.	Proposed solution		Solution in [20]		Solution in [19]	
	FP	FN	FP	FN	FP	FN
`BinS`	0%	0%	0%	0%	0%	0.523%
`MM`	0%	0%	0%	0%	0%	0.520%
`BubS`	0%	0%	0%	0%	0%	0.572%
`QS`	0%	0%	0%	0%	0%	0.607%
`SS`	0%	0%	0%	0%	0%	0.249%
`MD`	0%	0%	0%	0%	0%	0.912%
`CM`	0%	0%	0%	0%	-	-
`MED`	0%	0%	0%	0%	-	-
`TW`	0%	0%	0%	0%	-	-
`RS`	0%	0%	0%	0%	-	-
AVG	0%	0%	0%	0%	0%	0.663%

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

Bench.	Proposed solution		Solution in [20]	
	FP	FN	FP	FN
`BinS`	0.00%	0.00%	0.00%	2.25%
`MM`	0.00%	0.34%	0.00%	0.40%
`BubS`	0.00%	0.50%	0.00%	3.01%
`QS`	0.00%	0.08%	0.00%	3.91%
`SS`	0.00%	0.00%	0.00%	0.72%
`MD`	0.00%	0.00%	0.00%	2.83%
`CM`	0.00%	0.11%	0.00%	5.67%
`MED`	0.00%	0.18%	0.00%	2.60%
`TW`	0.00%	0.21%	0.00%	7.34%
`RS`	0.00%	0.05%	0.00%	3.34%
AVG	0.00%	0.15%	0.00%	2.29%

checker did not raise an alarm over the total number of runs. Similarly, we simulated 10,000 runs in which no HTH was activated and we measured the FP rate as the number of runs in which the checker raised an alarm over the total number of runs. Results from this experiment are reported in Table II, where we compare the current solution with the ones in [20] and in [19] (although the solution in [19] could not be applied to all benchmarks). As a first note, our proposal (as for the one in [20]) always achieves both 0% FP and FN rates; on the other hand, the proposal in [19] (which is based on Bloom filters) guarantees 0% FP rate but a not null FN rate.

We then emulated the activation of HTHs by modifying at a random time the fetched instruction itself after the microprocessor fetched from the expected memory location. In this way, we emulated HTHs that do not hijack the execution flow but that directly push malicious instructions in the microprocessor. 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 III where we also compare against the solution proposed in [20]. The proposed checker never raises false alarms (as for the checker proposed in [20]); on the other hand, when considering not detected HTH activations, the current proposal achieves only a 0.15% average FN rate (with several benchmarks showing 0% FN rate) while in the same cases, the proposal in [20] achieves a much higher 2.29% average FN rate, e.g., about 7% FN rate for `Towers`, more than 5% for `Coremark`.

Finally, we evaluated the overhead introduced by the proposed checker in terms of used resources, working frequency reduction, and power consumption increase when targeting an FPGA implementation. Table IV reports the number of LUTs and FFs and the amount of BRAM blocks required by our proposal and by the proposals in [20] and in [19] as well as the maximum working frequency that would be imposed by the presence of the checkers in the system (again, the solution in [19] could not be applied to all benchmarks). First of all, it may be noticed that the overhead introduced by the proposed checker is independent of the executed program, while when considering the checker proposed in [19], the larger the program, the larger the checker itself. It is also 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 9% in [19]. On the other hand, our solution requires much more BRAMs than the one in [19]. If we then compare the current solution with the one in [20], we can see that the introduced area overhead is practically the same. On the other hand, the current proposal brings the advantage highlighted by Table III. If we look at the working frequency overhead introduced by the considered security solutions we can see that none of them has an impact since the considered microprocessor works at 50 MHz. Finally, concerning the power consumption overhead, the considered microprocessor protected with the proposed checker has a total power consumption of 163mW, thus we introduce a 2.51% overhead w.r.t. the 159mW power consumption of the unprotected microprocessor, which we believe is totally acceptable. The power consumption of the same processor protected with the checker in [20] is about 164mW while no power consumption was reported in [19].

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 as well as more than 99% HTHs that try to force the CPU to execute malicious programs by directly tampering the fetched instructions. Furthermore, 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.

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

Table IV: Resource occupation and working frequency of our proposal and of the one in [19]

Bench.	Proposed solution				Solution in [20]				Solution in [19]			
	#LUTs	#FFs	#BRAM	F. (MHz)	#LUTs	#FFs	#BRAM	F. (MHz)	#LUTs	#FFs	BRAM	F. (MHz)
BinS	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	880 (5.43%)	84 (0.84%)	1	112 MHz
MM	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	880 (5.43%)	84 (0.84%)	1	112 MHz
BubS	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	880 (5.43%)	84 (0.84%)	1	112 MHz
QS	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	880 (5.43%)	84 (0.84%)	1	112 MHz
SS	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	1539 (9.13%)	89 (0.89%)	1	106 MHz
MD	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	1539 (9.13%)	89 (0.89%)	1	106 MHz
CM	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	-	-	-	-
MED	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	-	-	-	-
TW	82 (0.53%)	31 (0.31%)	8.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	-	-	-	-
RS	82 (0.53%)	31 (0.31%)	9.5	275 MHz	75 (0.49%)	31 (0.31%)	8	275 MHz	-	-	-	-

the same legal instruction (or sequence of legal instructions) from legal memory locations; 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. Conclusions

We presented a security architecture to protect microprocessor-based systems against hardware Trojan horses that try to force the execution of a malicious program. We integrated our proposal within a system featuring a RISC-V processor implemented on an FPGA device and running a set of software benchmarks. Our proposal was able to detect about 100% of possible HTHs activations with no false alarms. We measured a LUT and FF overhead of about 1% (with 8.5 up to 9.5 BRAMs required), a 2.51% power consumption increase and no working frequency reduction.

VII. Acknowledgment

This work has been supported by the ITACA project (grant no. PDC2022-133888-I00) funded by the Spanish Agencia Estatal de Investigacion (AEI) 10.13039/501100011033.

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

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

[6] 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.

[7] 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.

[8] C. Domas, "Hardware backdoors in x86 cpus." https://i.blackhat.com/us-18/Thu-August-9/us-18-Domas-God-Mode-Unlocked-Hardware-Backdoors-In-x86-CPUs-wp.pdf, 2018.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] A. Palumbo, L. Cassano, B. Luzzi, J. A. Hernández, P. Reviriego, G. Bianchi, and M. Ottavi, "Is your fpga bitstream hardware trojan-free? machine learning can provide an answer," *Journal of Systems Architecture*, vol. 128, p. 102543, 2022.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] A. Palumbo, L. Cassano, P. Reviriego, G. Bianchi, and M. Ottavi, "A lightweight security checking module to protect microprocessors against hardware trojan horses," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2021.

[21] 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.

[22] 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.

[23] 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.