HATE: a HArdware Trojan Emulation Environment for Microprocessor-based Systems

Cristiana Bolchini, Luca Cassano,
Ivan Montalbano, Giampiero Repole, Andrea Zanetti
Politecnico di Milano
Milano, Italy
{first_name.last_name}@polimi.it

Giorgio Di Natale
Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA*
38000 Grenoble, France
giorgio.di-natale@univ-grenoble-alpes.fr

***Abstract*—The constant quest of low production cost and short time-to-market, together with the growing complexity of integrated circuits led to the globalization of the supply chain of silicon devices. One of the threats related to such a supply chain are Hardware Trojan Horses (HWTs), that, in the last years, became a serious issue not only for academy but also for industry. Although a large number of methodologies for HWTs prevention, detection and tolerance have been proposed, there is a lack of well-recognized methods and metrics to evaluate their effectiveness. In this paper we present HATE[1], a HArdware Trojan Emulation Environment. The goal of HATE is twofold: (i) the tool can be used to analyse whether a given HWT (or a given set of HWTs) is activated by a software running on a microprocessor, and (ii) it can be used to assess HWTs detection techniques in microprocessors against a set of generated HWTs (either randomly or not). HATE represents, in our vision, a step towards the definition of a reference benchmarking scenario, to provide a comparative ground for evaluating different proposals focusing on HWT detection/tolerance. A subset of MiBench programs have been used to analyse the efficiency of HATE.**

***Index Terms*—Emulation, Hardware Security, Hardware Trojan Detection, Hardware Trojans, Microprocessors**

I. Introduction and related Work

Given the increasing complexity of modern integrated circuits (ICs) and the continuous seek for low production cost and short time-to-market, the current trend in ICs design and fabrication is to globalize many production activities [1]. After system specification, the designer often outsources the design of some of the hardware modules, buys third-party intellectual property cores (3PIPs), sometimes also outsourcing the masks and the final chip fabrication [2].

This globalized supply chain allows for a significant reduction of design cost and time, but comes with a significant loss of trust in the final delivered ICs [3]. Indeed, provided that it is very hard to ensure the trustworthiness of all the parties involved in the supply chain, such a distributed design and fabrication process exposes the product to a huge number of threats: ICs may be overproduced by the foundry and sold in the black market [4]; defective or dismissed ICs may be delivered as good ones [5]; IP core licenses may be violated and IP cores may be overused [6]; designs may be maliciously modified to insert stealthy unwanted functionalities in the final product, known as Hardware Trojan Horses (HWTs) [7].

Generally speaking, a HWT is a very-hard-to-detect modification of a design that remains silent most of the time, although in a specific (usually rare) working condition it alters the nominal behavior of the system. HWTs may be inserted at several stages and levels of abstraction: untrusted IP vendors may sell IP cores infected both at the hardware description language-level and at netlist-level [8]; a malicious employee may alter a few lines of code; a CAD tool provided by an untrusted software house may maliciously modify the design [9]; finally, untrusted mask providers and silicon foundries may alter the layout [10].

In the past, HWTs were considered an issue more by academy than by industry because of the difficulty of insertion in real-world large circuits and because of their reduced complexity and limited dangerousness. In the very last years, complex HWTs have been found in real microprocessors: existing HWTs may allow the attackers to execute their own malicious software, to modify the running software, or to acquire root privileges [11], [12]. Finally, in 2018, security researchers demonstrated the presence of a hardware backdoor, called the *Rosenbridge* backdoor, on a commercial Via Technologies C3 processor [13]. This hardware backdoor is activated and exploited via software to enter in supervisor mode. The reality of this attack makes HWTs a serious threat not only for industries also.

In the last two decades, a very large number of techniques for detection and prevention of HWTs have been proposed considering several attack scenarios, threat models and design stages where a HWT can be inserted [14]. Recently, software-based runtime solutions for HWTs prevention and detection have been proposed ([15], [16]). In [15] tasks are scheduled in such a way that tasks that directly exchange data are not executed by cores belonging to the same vendor, making collusion between 3PIPs impossible, and thus preventing the activation of HWTs. In [16], a genetic algorithm is employed to obfuscate the software before its execution, thus making HWT activation and information stealing extremely unlikely.

However, the same effort has not been devoted to the definition of standardized and commonly recognized methods and metrics to measure the effectiveness of such detection

*Institute of Engineering Univ. Grenoble Alpes

[1]HATE is freely available at http://cassano.faculty.polimi.it/

and prevention methodologies [17]. Recently, Trust-Hub contributed to the hardware security community with a set of 92 Trojan-infected circuits [18], [19]. This benchmark suite contains HWTs of varying size, inserted at various stages of the design process, thus representing a valuable effort to allow researchers to test their detection methodologies. Nevertheless, such a small set of benchmarks presents several limitations: (i) a small, static set of benchmarks may bias researchers to tune their methodologies to detect those HWTs only; (ii) Trust-Hub benchmarks contain only very few HWT-infested microprocessors; and (iii) the HWT-infested microprocessors available on Trust-Hub only belong to two microprocessor models. Very recently, the proposal in [20] tried to overcome these limitations by proposing a tool for the automated insertion of HWTs into a netlist. Although this is a valuable step forward, this tool only works at netlist level and it injects HWTs in the circuit without taking into account the actual controllability of the HWT.

In this paper we present HATE, a HArdware Trojan Emulation environment, addressing the analysis of the security vulnerabilities due to Hardware Trojans into microprocessor-based systems. Starting from the HWT-infested microprocessors available on Trust-Hub [21] we generalized them to build the HWT models considered by HATE. Given a software to be executed on a microprocessor, HATE aims at identifying the ability of the software to activate HWTs on the microprocessor. Like for well-known and well-established fault-injection tools, the rationale behind HATE is to provide designers of microprocessor-based systems with a flexible tool for *HWT-injection*. The designer can specify the target HW platform as well as the the SW to be executed during the analysis and the HWT models to be *injected*. The outcome of such a HWT-injection analysis is twofold: on the one hand the tool can be used to understand when/how a given HWT (or a given set of HWTs) is activated by a software running on a microprocessor. On the other hand, HATE can be used to assess the effectiveness of (innovative) detection techniques against a set of generated HWTs (either randomly or not) in microprocessors. Additionally, HATE allows the designer to inspect several parameters of the microprocessor's behavior and status during execution that could be of interest for vulnerability analyses, e.g., the number of times a register changes its value, the number of times each instruction is executed, number of I/O operations.

The proposed environment can actually be an alternative to having a large set of HWT-infested benchmark circuits, because it actually allows for an in-depth analyses, thus partially mitigating the limitations of the reduced number of HWT-infested benchmark circuits currently available, also aiming at contributing to the definition of metrics for standardized and recognized evaluation campaigns for HWT detection methodologies Moreover, we deem this proposal to be timely as the number of new proposals for software-based runtime solutions for HWTs prevention and detection is rapidly increasing, and HATE can be used to investigate their effectiveness.

The remainder of this paper is organized as follows: Section II introduces the considered threat model; Sections III presents the HATE environment; Section IV discusses a set of experiments that have been carried out to show the potentialities and the performance of HATE on a subset of the MiBench test programs [22]; finally Section V concludes the paper.

II. Threat Model

A rough classification may divide HWTs into *change-functionality*, *denial-of-service* and *information-stealing* ones. For the first two classes we can reasonably assume that the effect of the HWT will always be visible once the triggering condition has been activated, otherwise the attack would not be effective. On the other hand, for the information-stealing HWTs we can assume that the effect of the HWT will never be visible (from the point of view of the original functionality) after the activation of the HWT itself, otherwise the stealing would not be as stealthy as such kind of attacks require.

Thus, unlike in classical fault testing, where the generated test patterns need to activate the fault and then to propagate its effect to either a primary output or a flip-flop, when looking at change-functionality or denial-of-service HWTs, it is safe to assume that no logical or electric masking hides the effect of the Trojan once it has been activated. Similarly, we may assume that, when looking at information-stealing HWTs, no input values will expose the stealthy undesired behavior of the HWT. Given the above considerations, we argue that to evaluate if a software suffers from the presence of a HWT it suffices to evaluate its capability of activating the HWT without taking into account the payload.

According to the classification proposed in [23], HATE considers HWTs (both combinational and sequential) triggered by values (or combinations of values) in the microprocessor logic, memory or I/O, and inserted during any design phase/at any abstraction level. Do note that, always-on HWTs, time bomb HWTs, HWTs triggered externally through covert channels, and HWTs triggered by internal physical conditions of the microprocessor (e.g., temperature and voltage) fall outside of the scope of our work.

III. HATE: the HArdware Trojan Emulation Environment

HATE emulates the presence of HWTs into a microprocessor and support the analysis to determine whether a given software running on the microprocessor can trigger the HWTs. Our goal is to propose a software tool that can be used by microprocessors and software designers when dealing with hardware security and trust in the same way they use classical fault injection tools when dealing with reliability and test. The idea is to allow the designers of HWTs detection techniques to perform *HWTs injection campaigns* to automatically and thoroughly assess the effectiveness of their technique without having to rely on the few publicly available HWT examples. In this way it would be possible to measure the amount of "injected" HWTs the given technique can detect.

A high-level representation of the HATE environment is depicted in Figure 1. It is composed of an Execution Dump

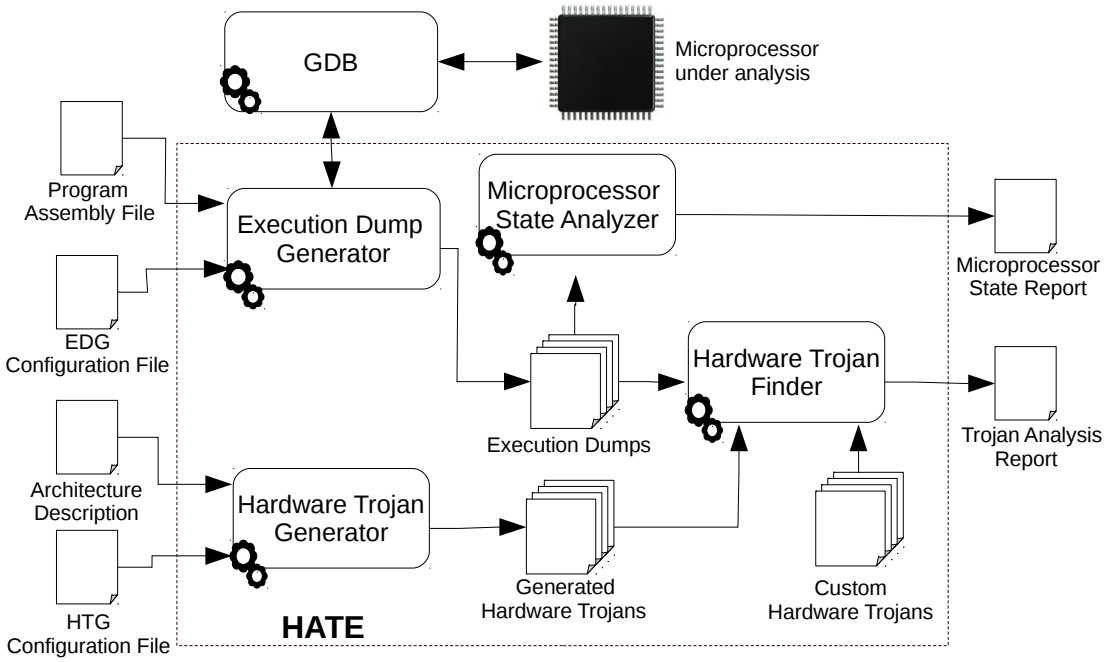

Fig. 1. The HATE Environment

Generator (EDG), a Hardware Trojan Generator (HTG), a Hardware Trojan Finder (HTF) and a Microprocessor State Analyzer (MSA). EDG extracts a set of execution traces from the executions of the software under analysis while running on the specific hardware platform the designer wants to address. HTG generates the list of HWTs to be *injected*. It is worth mentioning that apart from randomly-generated HWTs, the HWTs to be injected can also be manually specified when the designer wants to analyse specific HWTs. HTF analyses the previously generated executions traces, looking for the triggering conditions of the HWTs of interest and generates a detailed report on the activated HWTs and the microprocessor conditions when occurred. Finally, MSA monitors the internal state of the microprocessor and traces the runtime values of a number of parameters of interest.

A. The Implemented Hardware Trojan Models

The complexity of modern microprocessors provides a huge number of possibilities of HWTs insertion. Nevertheless, for the attack to succeed it is mandatory that the inserted HWT is controllable by the attacker, which means that there must exist stealthy but still easy-to-access ways to activate the HWT. By taking as a reference the 11 HWT-infested microprocessor examples available on Trust-Hub [21] (seven 8051 processors and four Microchip PIC16F84 microcontrollers) we can identify HWTs activated by specific data read from I/O interfaces (e.g., the UART interface) and HWTs activated by specific sequences of executed instructions.

Starting from the *real-world* HWT-infested microprocessors available on Trust-Hub we derived a set of HWT models to be implemented in the HATE environment and to be customized by the designer before the HWT-injection campaign. In particular, when modeling the needed HWT controllability for activating it, we mimicked the Trust-Hub circuits, and the set of considered triggering mechanisms, at present, includes: (i) specific combinations of values (or sequences of combinations of values) of the microprocessor's registers, (ii) specific values (or sequences of values) stored/loaded in specific I/O interfaces or memory addresses and (iii) specific sequences of executed instructions.

We do not consider HWTs triggered by specific internal signals or internal states of the microprocessor that are not visible between two instruction executions. Such triggering mechanisms would certainly be very hard-to-detect for detection techniques but also very hard-to-control by the attacker himself/herself.

Given the flexibility of the proposed framework, it is possible to add new triggering mechanisms that the user might be interested in.

B. The Execution Dump Generator

The preliminary step for the HATE analysis is the collection of one or more execution dumps of the execution of the considered software on the microprocessor under analysis. This execution is necessary to collect the microprocessor's resource status and registers' values. The *Execution Dump Generator* takes in input the assembly code of the software to be analyzed and a configuration file that specifies the number of executions to be traced and the input data for each execution. The EDG interacts with the GNU Debugger (GDB): the software is executed in debug mode and at each instruction execution the content of all CPU registers is collected and stored, creating the dump for all requested runs (an excerpt of

```
28      add                 fp , sp , #4
r0      0x1                 1
r1      0xbefff0a4          3204444324
r2      0xbefff0ac          3204444332
r3      0x1043c             66620
r4      0x1045c             66652
r5      0x0                 0
r6      0x10314             66324
r7      0x0                 0
r8      0x0                 0
r9      0x0                 0
r10     0xb6fff000          3070226432
r11     0x0                 0
r12     0xbeffefd0          3204444112
sp      0xbeffef48          0xbeffef48
lr      0xb6e7b678          -1226328456
pc      0x10440 0x10440     <main+4>
cpsr    0x60000010          1610612752
```

Fig. 2. An excerpt of an execution dump on an ARM v6 core

execution dump on an ARM v6 core is shown in Figure 2). These dumps are then used by the remaining modules to perform their analyses.

As mentioned, EDG requires the availability of the GNU Debugger. EDG can either be executed on the specific microprocessor under analysis, or, if not available, it can also be run on top of a simulation environment emulating the target microprocessor, e.g., the gem5 environment [24]. The remaining components of HATE take in input the description of the target architecture but can be executed on any processing platform.

C. The Hardware Trojan Generator

The *Hardware Trojan Generator* is in charge of randomly generating a set of HWTs to be used in the subsequent detection analysis. As previously discussed, the HATE environment models HWTs in terms of their triggering mechanism: we considered all the HWT-infested microprocessors available on Trust-Hub and we generalized them to build the following set of models of HWT triggers[2]:

- *HWT-Type 1*: a specific set of values in a specific set of registers; this models combinational HWTs infesting the CPU registers.
- *HWT-Type 2*: a specific sequence of values in a specific set of registers; this models sequential HWTs infesting the CPU registers.
- *HWT-Type 3*: a specific sequence of instructions and (possibly) associated operands; this models sequential HWTs infesting the fetch, decode and ALU/FPU modules.
- *HWT-Type 4*: a specific value read/written from one I/O interface or memory location; this models combinational HWTs infesting the data/address bus.
- *HWT-Type 5*: a specific sequence of values read/written from specific I/O interfaces or memory locations; this models sequential HWTs infesting the data/address bus.

[2]HATE can be extended to integrate additional HWT models.

HTG takes in input a description of the microprocessor architecture in terms of CPU registers and instruction set, and a configuration file specifying how many HWTs have to be generated, the type and, for each one of the HWTs types, the following information:

- *HWT-Type 1*: the maximum number of triggering registers N_R.
- *HWT-Type 2*: the maximum number of states of the sequential HWT N_S and the maximum number of triggering registers N_R.
- *HWT-Type 3*: the maximum number of states of the sequential HWT N_S.
- *HWT-Type 4*: no configuration is needed.
- *HWT-Type 5*: the maximum number of states of the sequential HWT N_S.

According to the specified configuration information, HTG generates the desired number of HWTs by first randomly determining the type of HWT and then, according to the selected type, by performing the following operations:

- *HWT-Type 1*: a random number $n_R \in [1, N_R]$ is chosen; n_R random registers are then selected and for each of them a random value is set.
- *HWT-Type 2*: a random number $n_S \in [1, N_S]$ is chosen, representing the number of states of the sequential HWT; for each of the n_S states the same random information needed for *HWT-Type 1* is created.
- *HWT-Type 3*: a random number $n_S \in [1, N_S]$ is chosen representing the number of states of the sequential HWT; for each of the n_S states a random instruction is chosen and its operands are randomly set.
- *HWT-Type 4*: a random value v is set as an operand to the `load/store` instruction.
- *HWT-Type 5*: a random number $n_S \in [1, N_S]$ is chosen, representing the number of states of the sequential HWT; for each of the n_S states the same random information needed for *HWT-Type 4* is created.

It is worth mentioning that such a random selection of the triggering mechanisms for the generated HWTs does not mean that we model HWTs that are randomly triggered at runtime. All the *random* parameters that are chosen by the HTG to characterize the behavior of the generated HWTs are randomly chosen at design time, before the HWT-injection. Indeed, during the actual HWT-simulation, each considered HWT will have its own specific triggering mechanism.

D. The Hardware Trojan Finder and the Microprocessor State Analyzer

The last two components of the HATE environment are the *Hardware Trojan Finder* and the *Microprocessor State Analyzer*. Both components take in input the execution dumps previously extracted by the EDG component. Additionally, HTF takes in input two lists of HWTs: the ones randomly

generated by the HTG and, possibly, those manually specified by the designer.

HTF is devoted to the identification of whether the triggering conditions for both previously randomly generated and manually specified injected HWT(s) are found in the execution dump(s). At the end of this process an output file reporting all activated HWTs and the microprocessor's working conditions that activated them is generated (similarly to what happens after fault injection experiments).

MSA monitors a number of parameters of the microprocessor functioning during the simulation. By looking at the execution dumps, MSA reports parameters such as the number of times the content of each register changes, the number of I/O instructions and memory load/store executed instructions, and how many times each instruction is executed. These parameters can then be used by the designer to analyse possible vulnerability locations. As for other aspects, the set of HWT models and monitored parameters can be extended.

IV. Experimental Results

We evaluated the performance of the various components of HATE when used to analyse different microprocessors and benchmark programs, while the analysis of the effectiveness of specific HWT detection techniques falls outside the scope of the current paper.

We implemented HATE as a set of Bash scripts and C and Java programs. As a set of demonstrative case study programs we considered C implementations of `BubbleSort` (`BS`), `MatrixMultiplication` (`MM`) and `Fibonacci` (`Fib`) and the MiBench programs `QuickSort` (`QS`), `Dijkstra` (`Dij`), `SHA`, `AES`, and `FFT` [22].

To show the portability of HATE we selected two microprocessors as the target architectures for the analysis: the embedded 1 GHz 32-bit single-core ARM1176JZF-S processor of a Raspberry Pi Zero board and a Desktop 3.40GHz 64-bit Intel Core i7-3770. The Execution Dump Generator has been executed there to collect the traces, while, as previously discussed, the remaining modules of HATE have been run on a desktop PC equipped with a 3.40GHz 64-bit Intel Core i7-3770 processor and 8GB RAM.

A. EDG performance

As a first experiment we analysed the performance of the Execution Dump Generator. Table I reports the average execution time (in minutes) for the generation of an execution dump for the considered benchmarks, compiled with four different optimization levels for the two considered target processors. As expected, the performance of the Execution Dump Generator depends on the adopted compiler optimization level as well as on the processing power of the target microprocessor on which the EDG is run. Nevertheless, the execution time is always in the order of magnitude of some minutes.

B. HTG performance

The execution time of the HWT Generator does not depend on the hardware platform under analysis, on the executed program, nor on the HWT models to be generated but only on the number of HWTs the designer wants to randomly generate and on the number of triggers. Table II reports the execution time (in milliseconds) when generating 1,000 random HWTs having a number of triggers N_R in various ranges. As expected, the execution time increases with the number of triggers. Nevertheless, even when this number is possibly large, the generation time is still in the order of magnitude of some milliseconds.

TABLE I
EDG: Average Execution Time (in minutes)

	ARM1176JZF-S				Intel Core i7-3770			
	-O0	-O1	-O2	-O3	-O0	-O1	-O2	-O3
`BS`	2.59	0.76	0.43	0.28	0.48	0.52	0.15	0.13
`MM`	0.87	0.37	0.34	0.33	0.09	0.04	0.03	0.01
`Fib`	1.13	0.41	0.33	0.28	0.07	0.05	0.05	0.04
`QS`	1.12	0.49	0.48	0.48	0.86	0.67	0.65	0.63
`Dij`	0.57	0.45	0.39	0.21	0.12	0.05	0.03	0.01
`SHA`	4.06	1.37	1.36	1.27	0.67	0.27	0.45	0.39
`AES`	0.89	0.33	0.11	0.04	0.17	0.09	0.03	0.01
`FFT`	1.93	1.32	0.93	0.81	0.51	0.42	0.38	0.33

TABLE II
HTG: Average Execution Time (in milliseconds)

# of HWTs	$N_R \in [1,3]$	$N_R \in [1,5]$	$N_R \in [1,7]$	$N_R \in [1,10]$
1,000	5.02ms	6.83ms	7.40ms	8.76ms

C. HTF and MSA performance

We analyzed the efficiency of the two output report generators (HTF and MSA) by measuring the time required to parse one of the previously extracted executions dumps.

Table IV reports the average execution time (in seconds) of HTF with respect to different optimization levels, when considering the 1,000 previously generated HWTs. The time required to scan the program execution dump depends on the length of the dump, and thus, on the optimization level. HTF completes its task in some tenth milliseconds in all cases.

A similar experiment has been carried out to analyse the performance of the MSA. Table IV reports the execution time (in seconds) w.r.t. different optimization levels for the considered programs. As for the analysis carried out by HTF, also for MSA the time required to scan the program execution dump depends on the length of the dump, and thus again, on the optimization level. MSA completed its task in some tenth milliseconds in all cases.

V. Conclusions and Discussion

We presented HATE, a framework for Hardware Trojan simulation in microprocessors to investigate the vulnerabilities of a given microprocessor when executing a specific software. The framework can also be used to evaluate the effectiveness of software-based techniques for Hardware Trojans detection. HATE implements a set of HWTs models derived from the

TABLE III
HTF: AVERAGE EXECUTION TIME (IN SECONDS)

	Optimization level			
	-O0	-O1	-O2	-O3
BS	0.052s	0.049s	0.036s	0.036s
MM	0.048s	0.042s	0.039s	0.037s
Fib	0.048s	0.041s	0.040s	0.032s
QS	0.047s	0.032s	0.028s	0.026s
Dij	0.213s	0.1022	0.0872	0.075s
SHA	0.868s	0.376s	0.252s	0.149s
AES	0.354s	0.218s	0.133s	0.096s
FFT	0.751s	0.448s	0.175s	0.102s

TABLE IV
MSA: AVERAGE EXECUTION TIME (IN SECONDS)

	Optimization level			
	-O0	-O1	-O2	-O3
BS	0.123s	0.095s	0.093s	0.027s
MM	0.891s	0.118s	0.079s	0.073s
Fib	0.095s	0.082s	0.079s	0.076s
QS	0.121s	0.099s	0.098s	0.066s
Dij	0.476s	0.143s	0.107s	0.097s
SHA	0.859s	0.611s	0.483s	0.402s
AES	0.752s	0.561s	0.458s	0.348s
FFT	0.821s	0.587s	0.519s	0.392s

real-world HWT-infested microprocessors that are available on Trust-Hub. With respect to the limitations exposed by the available benchmark circuits and the existing automated HWT insertion tools, HATE offers the following advantages:

- model and inject an unlimited number of HWTs;
- extend the available HWTs models by combining the existing ones;
- inject multiple HWTs in a single microprocessor;
- inject HWTs that are actually controllable by the attacker, thus preventing the analysing low-level HWTs that cannot be actually triggered;
- analyse any microprocessor architecture provided the availability of the GDB debugger;
- monitor and collect additional information concerning the state of the microprocessor's resources during the HWT simulation.

Given the huge number of Hardware Trojan detection techniques and the interest in both attacking and protecting complex HW systems, and considering the lack of infested benchmark circuits, automated HWT injection tools and evaluation metrics on the other hand, we believe that HATE represents a step towards a commonly-adopted tool to help the Hardware Security community in the evaluation of the vulnerabilities of given microprocessor architectures and in the assessment of the effectiveness of strengthening techniques and in their tuning/improvement.

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

[14] S. Bhasin and F. Regazzoni, "A survey on hardware trojan detection techniques," in *Proc. Int. Symp. on Circuits and Systems*, pp. 2021–2024, 2015.

[15] C. Liu, J. Rajendran, C. Yang, and R. Karri, "Shielding heterogeneous mpsocs from untrustworthy 3pips through security- driven task scheduling," *IEEE Trans. on Emerging Topics in Computing*, vol. 2, no. 4, pp. 461–472, 2014.

[16] A. Marcelli, E. Sanchez, G. Squiller, M. U. Jamal, A. Imtiaz, S. Machetti, F. Mangani, P. Monti, D. Pola, A. Salvato, and M. Simili, "Defeating hardware trojan in microprocessor cores through software obfuscation," in *Proc. Latin-American Test Symp.*, pp. 1–6, 2018.

[17] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Trans. Design Automation of Electronic Systems*, 2016.

[18] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, 2017.

[19] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *Proc. Int. Conf. Computer Design*, pp. 471–474, 2013.

[20] J. Cruz, Y. Huang, P. Mishra, and S. Bhunia, "An automated configurable trojan insertion framework for dynamic trust benchmarks," in *Proc. Design, Automation & Test in Europe Conf.)*, pp. 1610–1615, 2018.

[21] Trust Hub. www.trust-hub.org.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. Int. Workshop on Workload Characterization)*, pp. 3–14, 2001.

[23] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[24] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.