DETON: DEfeating hardware Trojan horses in microprocessors through software ObfuscatioN

Luca Cassano[1], Mattia Iamundo[1], Tomas Antonio Lopez[1], Alessandro Nazzari[1], Giorgio Di Natale[2]

Abstract

Hardware Trojan Horses (HTHs) represent today a serious issue not only for academy but also for industry because of the dramatic complexity and dangerousness attackers can count on. It has been shown that HTHs can be inserted in modern and complex microprocessors allowing the attacker to run malicious software, to acquire root privileges and to steal secret user information. In this paper we propose DETON, an automatic methodology for software manipulation aimed at introducing obfuscation in programs' execution to protect microprocessor-based systems against information stealing HTHs. The high-level goal of DETON is to produce an obfuscated version of the program under protection in order to allow a trusted execution over a (possibly) untrusted CPU-based system. The obfuscated program will then be the one actually executed on the target hardware platform. DETON aims at i) reducing the amount of sensitive information exposed to the attacker by spreading it through microprocessor's registers and by submerging it among garbage information, and ii) reducing the time for which sensitive information is exposed to the attacker by scrambling data among microprocessor's registers. We first present a set of guidelines, requirements and metrics aimed at driving and assessing software obfuscation against always-on information-stealing HTHs and we then present the DETON framework. We prove the effectiveness and efficiency of DETON on the Ariane version of the 64bit RISC V microprocessor running a set of MiBench programs.

Keywords: Hardware Security, Hardware Trojan Detection, Hardware Trojan Horses, Microprocessors, Software Obfuscation

1. Introduction

The increasing complexity of modern integrated circuits (ICs) and the continuous seek for low production cost and short time-to-market, pushed the ICs design and fabrication towards a globalized supply chain [1]. Indeed, after system requirements have been specified, the design house often outsources the design of some of the hardware modules, or it resorts to third-party intellectual property cores (3PIPs) and even it outsources the masks definition and the final chip fabrication [2].

The benefit of such a globalized supply chain is a significant reduction of design cost and time. On the other hand, this comes at the cost of a significant loss of trust in the final delivered ICs [3]. The consequence of this globalization is that it is very hard to ensure the trustworthiness of all the parties involved in the supply chain. Therefore, the product is exposed 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, also known as Hardware Trojan Horses (HTHs) [7].

A HTH can be defined as a very-hard-to-detect malicious modification of a design meant i) to stay silent most of the time and to activate in a specific (usually rare) working condition to alter the nominal behavior of the system, or ii) to stay constantly active and to secretly steal sensitive information processed by the system [7]. HTHs may be inserted in any stage stage of the design process and at any level of abstraction: untrusted IP vendors may sell IP cores infected both at the hardware description language-level and at netlist-level [8]; rogue employees and untrusted CAD tools may maliciously modify the design [9]; finally, untrusted mask providers and silicon foundries may alter the layout [10].

In the past, HTHs have been considered a purely academic issue because they generally exposed reduced complexity and, as a consequence, limited dangerousness. In the very last years, a new menace raised: the so called *software exploitable HTHs* [11]. Indeed, it has been demonstrated that complex and highly dangerous HTHs may be implanted in real-world microprocessors. A software-exploitable HTH may allow the attackers to execute their own malicious software, to modify the running software or to steal secret information [12], or even to acquire root privileges on the system [13]. Finally, in 2018, security researchers demonstrated the presence of a hardware backdoor, called the *Rosenbridge* backdoor, on a commercial Via Technologies C3 processor [14, 15]. This hardware backdoor can be activated and exploited via software (by making the CPU fetch a pre-define sequence of instructions) to enter in supervisor mode. The feasibility of implanting and activating software exploitable HTHs in real-world microprocessors makes such attacks not only a concern for academy but also a serious threat for industry.

There is a vast literature about HTHs detection techniques [16]. Most of these methodologies attempt to detect the HTH before the circuit has been deployed, by exploiting a plethora of techniques: logic testing [17], formal property verification [18], side-channel analysis [19], optical inspection [20], proof-carrying

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy, email: {name.surname}@polimi.it

[2]Institut Polytechnique de Grenoble, Universite Grenoble Alpes, TIMA, Grenoble, France, email: giorgio.di-natale@univ-grenoble-alpes.fr

hardware [21]. All these techniques suffer from a number of limitations, among which the difficulty of triggering the HTHs at design time for detection, the need for a golden reference of the circuit under analysis, the ability of detecting only a specific sub-class of HTHs. More recently, the need for HTHs tolerance techniques able to build trusted systems from untrusted components or to provide trusted execution from untrusted systems has been pointed out, moving towards the *Design for Trust* paradigm [22, 23]. The existing anti-Trojan Design-for-Trust approaches are based on the integration of redundant functionally equivalent IP cores belonging to different IP vendors, like the proposals in [24, 25]. Moreover, security-aware task scheduling techniques for systems built by integrating redundant IP cores belonging to different vendors have been proposed [26, 27, 28]. The applicability of these approaches is confined to those scenarios where the hardware platform is still to be developed and the designer has the freedom to add redundancy and diversity. Moreover, all these approaches protect the system against those HTHs that aim at modifying the functionality of the system itself, while they are ineffective against those HTHs that steal secret information from the system.

In this paper we propose DETON, a software obfuscation methodology for mitigating the dangerousness of information-stealing HTHs in microprocessors without requiring any modification to the underlying HW platform. The high-level goal of DETON is to produce an obfuscated version of the program under protection in order to allow a trusted execution over a (possibly) untrusted CPU-based system. The obfuscated program will then be the one actually executed on the target hardware platform. DETON exploits software obfuscation to reduce the probability of exposing sensitive information to the HTH. Indeed, starting from the original software, DETON increases the usage of the microprocessor's registers, adds garbage instructions and reduces the time in which any register holds sensitive information. More in details, to achieve such minimization of the probability of exposing sensitive information to the HTH, DETON spreads sensitive information through microprocessor's registers and submerges it among garbage data

With respect to the state-of-the-art anti-Trojan Design- for-Trust techniques, DETON is a pure software-based methodology. It can therefore be applied both when the system is still to be designed as well as on already designed and deployed systems. Moreover, DETON does not require any redundancy or modification to standard microprocessors. The only work proposing a similar idea is the one reported in [29] where software diversity is achieved by substituting program instructions with equivalent ones. Nevertheless, the proposal in [29] only considers sequentially-triggered change-functionality HTHs and equivalent instructions substitution is solely employed. By summarizing, the main contributions presented in this paper are:

- the definition of security-aware guidelines and requirements aimed at driving the process of software obfuscation to alleviate the dangerousness of always-on information-stealing HTHs,
- the subsequent definition of two novel quantitative metrics specifically aimed at assessing the susceptibility of the running software to always-on information-stealing HTHs,
- the DETON framework for anti-HTH software obfuscation, and
- an in depth experimental campaign aimed at assessing the effectiveness and efficiency of DETON on a set of MiBench programs [30] executed on the Ariane RISC V 64bit processor [31].

The remainder of this paper is organized as follows: Section 2 presents the background on HTHs and the considered threat model; Section 3 discusses the defined guidelines for anti-HTH software obfuscation and presents the metrics defined for the evaluation of the susceptibility to information stealing of a running program; Section 4 discusses the DETON framework while Section 5 presents the results of the performed experimental campaign and depicts a security analysis; Section 6 reviews the state-of-the-art approaches for HTHs detection and tolerance while Section 7 concludes the paper.

2. Background

We here first comment about the most common types of HTHs and we then specify the HTH model taken into account by DETON as a threat model.

2.1. Hardware Trojan Horses

As it has been previously discussed, a HTH is a very hard-to-detect modification of a system i) that keeps silent most of the time, and becomes active under specific rare conditions, altering the nominal behavior of the system, or ii) that is always active and covertly steal sensitive information processed by the system. According to the taxonomy presented in [7], HTHs may be classified based on their *triggering mechanism*, *payload* and *insertion phase*.

A HTH may be triggered:

- *internally* by logical signals (or sequences of logical signals, in case of sequential HTHs) or by physical quantities, e.g., internal temperature or voltage, or by a hidden ad-hoc configured counter (the so-called time bombs);
- *externally* by either received messages or commands, or by physical interactions, e.g., again the external temperature or voltage; and
- *always-on*, i.e., HTHs that become active as soon as the system is turned on.

Under the point of view of the payload, i.e., their effect on the infested system, HTHs may be classified in:

- *Change functionality HTHs* that alter the nominal functionality of the infected system, e.g., make the system execute a malicious code;

- *Information stealing HTHs* that steal secret information from the system either through the available communication interfaces, e.g., by sending unauthorized messages to the attacker, or through covert side-channels, e.g., temperature or magnetic field; and
- *Denial-of-service HTHs* that stop the functioning of the system, e.g., by introducing *nop* instructions, by draining the system's batteries, or by jamming the communication interfaces.

Finally, from the insertion point of view, HTHs may be maliciously added in the design by IP providers in the purchased 3PIPs, by rogue designers and by the employed CAD tools possibly in every stage of the design flow and by the foundry during chip fabrication.

2.2. The Considered Threat Model

Referring to the classical HTHs classification [7], DETON takes into account both triggered and always-on information stealing HTHs infesting microprocessor's logic, inserted during any phase of the design process, i.e., by malicious IP providers, employees, CAD tools or by the foundry, and at any level of abstraction, i.e., logic netlist, physical design or layout. More in details, we assume a two-level information stealing attack: first, the HTH repeatedly exfiltrates the content of a number of registers of the processor and covertly sends this raw data to the attacker. Then, the attacker collects this data and post-processes it to retrieve the sensitive information of interest. We assume that, when injecting the trojan 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.

We assume that the injected HTH monitors and exfiltrates data only from a reduced number of registers of the infested processor, i.e., the attacker cannot count on a full dump of the content of all the registers of the infested microprocessor. We believe that this assumption is totally reasonable if we keep in mind that: i) the HTH needs to be small enough not to be detected via optical inspection, ii) the HTH needs to have an extremely reduced impact on power consumption, electromagnetic emission and timing, and iii) the HTH cannot occupy the transmission channel for long without being discovered. Therefore, we assume that the HTH monitors the content of a fixed (at injection-time) and small set of registers and exifiltrates data through a (possibly large) number of clock cycles. On the other hand, because of the previously mentioned limitation to the HTH complexity, we assume that the HTH is not able to change the monitored registers, e.g., in a round-robin fashion. Finally, we may also assume that the attacker knows all the details of the deployed software-based countermeasure but this does not bring him/her any additional advantage.

On the other hand, we do not take into account change the functionality and denial-of-service HTHs.

3. Obfuscation principles

Obfuscation has been widely employed both for hardware [32] and for software protection [33]. The goal of obfuscation is generally to protect the intellectual property associated with a program or a circuit from unauthorized use or reproduction.

Hardware obfuscation has been proposed to avoid i) reverse engineering of the circuit's functionality by observing the circuit's netlist or layout and ii) overproduction of unauthorized chips that could then be sold in the black market. To do so, non standard cells may be employed (*camouflaging*) or the netlist may be "locked" in order to make the fabricated circuit unusable before unlocking it through a secret key (*logic locking*). The metrics generally adopted to evaluate the strength of hardware obfuscation techniques measure i) the number of brute force attempts required to unlock the circuit without knowing the secret key or to guess the secret key itself, ii) the Hamming distance between the outputs of an obfuscated circuit when applying an incorrect key and the output of the unlocked circuit, and iii) the number of input patterns that produce an incorrect output when applying an incorrect key to the circuit.

Similarly, software obfuscation aims at making hard for a reader (for example a decompilation tool) to understand the functionality implemented by a program, the meaning of a given construct or variable, the value of constants, the structure of classes and arrays. As for hardware obfuscation, the goal of obfuscating programs is to avoid intellectual property break. To do so, never-executed dummy code may be inserted, instructions may be reordered or hidden, loops may be unrolled, intersected or extended, logic conditions may be opacified, arrays and data structures may be split or merged. The metrics generally adopted to evaluate the strength of software obfuscation techniques aim at assessing how difficult is for a human reader or a decompiler to understand the behavior of a program.

Given the above considerations, we argue that obfuscation techniques have been traditionally meant to deal with a *static external* attacker while we consider information stealing HTHs as *dynamic internal* threats. In the following of this section we first discuss the deep difference between these two attacks that drive the need for new software obfuscation guidelines and metrics. Then, we give details about the obfuscation guidelines we have identified and integrated into DETON and the metrics we developed to evaluate the effectiveness and efficiency of DETON.

3.1. Guidelines for software obfuscation against HTHs

As discussed above, we believe that obfuscation techniques have been traditionally designed to deal with a *static external* attacker that looks at the circuit's structure or at the program's code with the aim of gaining knowledge and being able to reproduce it. On the other hand, according to the previously discussed threat model, we observe that the considered information stealing HTHs represent a *dynamic internal* attacker threat. Indeed, the considered HTHs act by observing the behavior of the running program from inside and they aim at discovering the data computed by the running program at runtime. Given these peculiarities, we believe that i) novel obfuscation guidelines have to be drawn to deal with information stealing HTHs, and ii) new metrics have to be proposed to capture how well a program has been obfuscated not to expose secret information to such a HTH model.

We defined the following guidelines to drive the process of obfuscating software against information stealing HTHs.

Guideline 1: The variables of the program under protection should reside in the largest possible set of registers during program execution. In this way, the probability of exposing sensitive information to the attacker through the few registers monitored by the HTH is kept small.

Guideline 2: The amount of non sensitive information per time unit processed by the program under protection should be kept as large as possble. In this way again, for each time unit, the probability of exposing sensitive information to the attacker through the few registers monitored by the HTH is kept small.

As a last, non security-related requirement, it is of course fundamental not to excessively degrade performance while ensuring security, i.e., not excessively increase the execution time of the program under protection.

3.2. The defined metrics

In the very last years a number of metrics and methods for measuring the susceptibility of digital systems to side-channel attacks have been proposed [34, 35, 36]. These approaches aim at measuring how much the significant information processed by the system is correlated to side information, e.g., timing behavior, power consumption or electromagnetic emission, and thus, how much significant information the attacker may retrieve by observing such side information. We claim that such metrics do not apply for the considered information-stealing HTH model. Indeed, in our scenario the HTH is assumed to work within the infested microprocessor, thus it is able to observe plain information from processor's registers. Therefore, the problem for an attacker that exploits the considered HTH model is not to infer sensitive information from side information but to identify the sensitive data among all the data chunks leaked and transmitted by the HTH (that possibly contain both garbage and sensitive data) and then to reconstruct the sensitive information based on the received data chunks. Therefore, we believe that novel HWH-aware metrics are required to measure the susceptibility to information leakage of a CPU-based system executing a given software.

Based on these considerations and on the previously discussed design guidelines we defined the following security-related metric plus an overhead metric to assess the effectiveness and the efficiency of DETON.

Registers heat (H): given a processor's register r, this metric, noted as H_r, is a reverse measure of time elapsed since the last data has been written in r. When a data is written in r, H_r is set to a given configurable value H_{MAX} and it is then decreased at each instruction cycle until either a new data is written in r or H_r equals 0. The higher the average heat of a register over time, and more in general of all processor's registers, the larger the amount of data processed by a program. As a consequence, the higher the average registers' heat the harder for an attacker to identify the sensitive information among all the processed data.

Program enlargement (E): given a plain program p and the associated obfuscated program p_o, this metric, noted as $E^p_{p_o}$, measures the difference between p and p_o in terms of number of assembly instructions. The higher $E^p_{p_o}$ the higher the overhead introduced by obfuscation.

4. The DETON Framework

DETON is a design time framework for software obfuscation aimed at mitigating the dangerousness of information-stealing HTHs in microprocessors by implementing the software obfuscation guidelines defined in Section 3.1. The high-level goal of DETON is to produce an obfuscated version of the program under protection in order to allow a trusted execution over a (possibly) untrusted CPU-based system. More in details, DETON takes the program under protection and produces an obfuscated version of the program such that the two versions are functionally equivalent but the obfuscated program has higher values of the previously discussed security metrics. The obfuscated version of the program will then be the one actually executed on the target hardware platform. DETON exploits software obfuscation at the assembly-level to reduce the amount of significant information exposed to HTHs. Starting from the original assembly of a program, DETON produces an obfuscated version of the program to be then deployed in the system.

From a high-level point of view, the strategy adopted by DETON aims at: i) spreading sensitive information through microprocessor's registers and submerging it among garbage information, and ii) periodically scrambling sensitive information among microprocessor's registers. In this way, keeping in mind that the considered HTH model is able to monitor and send to the attacker the content of a reduced number of processor's registers, DETON achieves two benefits: i) reducing the amount of sensitive information exposed to the attacker, and ii) increasing the amount of garbage information exposed to the attacker. More in details, the software obfuscation techniques implemented by DETON are: i) **garbage code insertion**, ii) **constants obfuscation**, and iii) **register scrambling**.

A high-level representation of the DETON flow for software obfuscation is depicted in Figure 1. DETON takes the assembly code of the program to be protected and produces the assembly code of the obfuscated version of the program that is the one actually meant to be executed over the untrusted HW platform. Of course, the original program and generated obfuscated one are functionally identical, i.e., the two programs produce identical outputs when fed with identical inputs. Moreover, DETON takes a description of the architecture on which the program will be executed and a configuration file to drive the obfuscation process (more details about the configuration parameters will be provided in the following subsections). The architectural description file specifies the instruction set of the target

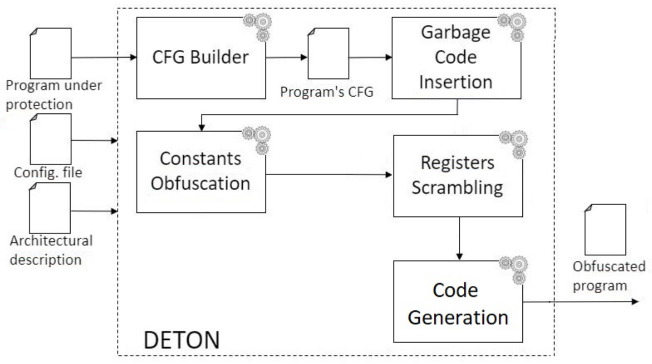


Figure 1: The DETON flow

```
main:
      addi   sp, sp,-32
      sd     s0, 24(sp)
      addi   s0, sp, 32
      li     a5, 10
      sw     a5, -20(s0)
      li     a5, 4
      sw     a5, -24(s0)
      lw     a5, -20(s0)
      addiw  a5, a5, 1
      sw     a5, -20(s0)
      lw     a4, -24(s0)
      lw     a5, -20(s0)
      addw   a5, a4, a5
      sw     a5, -24(s0)
      li     a5, 0
      mv     a0, a5
      ld     s0, 24(sp)
      addi   sp, sp, 32
      jr     ra
```

Figure 2: The assembly code of an example program

microprocessor in terms of instructions' format and operands and the registers' names and size.

The first step performed by DETON is parsing the input assembly code to build an internal Control-Flow Graph (CFG) representation of the program. Then the designed software obfuscation techniques are applied in the following order: i) insertion of garbage code, ii) obfuscation of the constants, and iii) scrambling of registers. As a final step, based on the CFG resulting from the applied manipulation techniques, the output obfuscated program is generated. In the remainder of this section we provide more details about the functioning of DETON. As a final note, keep in mind that all the software obfuscation examples reported in the following will be referred to the intentionally simple original program shown in Figure 2 where two variables are allocated and their sum is calculated.

4.1. Control-flow graph generation

A control-flow graph (CFG) is a graph representation of all execution paths that might be traversed by a program during its processing. Nodes in a CFG represent program's basic blocks, i.e. a sequence of instructions without any jump while directed edges between nodes are used in the CFG to represent jumps in the control flow. Moreover, there are two specially designated blocks: the entry block, through which the control enters into the flow graph, and the exit block, through which all control flow leaves. Given this premises, it is possible to build a CFG univocally describing the structure of the program under analysis. Then, based on the previously built CFG, the *Execution Graph (EG)* of the program is built. An EG is a compact graph representation of all the possible executions of a program. Such EG allows DETON to trace the status of the underlying microprocessor in terms of unused and busy registers. Moreover, based on the EG, for every register DETON keeps track of the time at which the content of the register changed. All these status information are then exploited by the subsequent modules of DETON to drive the obfuscation process.

```
slli   a6, t5, 30
sltu   a4, s3, s11
mv     t4, a7
mulh   t4, s5, a1
sltiu  t0, ra, -1657
sra    s11, t6, s5
```

Figure 3: An example of garbage code insertion between lines 4 and 5 of the program in Figure 2

4.2. Garbage code insertion

The first software manipulation performed by DETON is the insertion of garbage code within the program to be obfuscated. More in details, each time it is invoked, the module in charge of inserting garbage code randomly selects the block of the program, i.e., the node of the CFG, where to insert the garbage code, the code line within the block after which inserting and the length of the garbage code sequence to be inserted. The garbage instructions are randomly selected among the move, shift, arithmetic and logic ones. The operands of the inserted garbage instructions are also randomly generated. Because of the randomness of the operands, to prevent anomalous working conditions no division instructions are inserted (to avoid possible divisions by zero) as well as no jump instructions are inserted (to avoid jumping into unauthorized memory areas).

The registers from which garbage instructions read and in which they write are chosen among the registers that are unused in the block in which the garbage code is inserted. More in details, for each inserted garbage instruction the unused register whose content has not been modified since more time is identified as the destination register of the inserted garbage instruction itself. In this way DETON allows to increase the usage of all the registers in the microprocessor as well as it reduce the time between two consecutive modifications of the content of a register. Moreover, garbage code insertion allows DETON to break specific instructions patterns whose identification during program execution could be of interest for the attacker. As an example of garbage code insertion, Figure 3 reports six instructions inserted between lines 4 and 5 of the program reported in Figure 2.

4.3. Constants obfuscation

The second software manipulation performed by DETON is the obfuscation of the constant values within the program. Each

```
li    t5, 463
slli  s6, t5, 3
srli  s9, s6, 3
ori   t2, s9, -976
slli  t3, t2, 0
andi  a7, t3, 1019
andi  t6, a7, -450
ori   t0, t6, 1668
andi  a2, t0, -1727
ori   t1, a2, 1
slli  t4, t1, 11
srli  s11, t4, 6
add   s0, sp, s11
```

Figure 4: An example of constant obfuscation referred to the original constant at line 4 of the program in Figure 2

time it is invoked, the module in charge of obfuscating the constants randomly selects a block of the program and, within the selected block, it randomly selects a code line where an immediate value, i.e., a constant, is used by the instruction. It is worth mentioning that any constant value in the program may be obfuscated, i.e., both the operands used by arithmetic and logic instructions and the offsets used by memory access instructions. Let us refer to the instruction whose constant operand is going to be obfuscated as the *target instruction* of the obfuscation process; moreover, let us refer to the destination register of the target instruction, i.e., the register in which the result of the target instruction is going to be stored, as the *target register* of the obfuscation process and to the constant value used by the target instruction as the *target value*. The overall goal of the constant obfuscation module is substituting the target instruction with a randomly long sequence of instructions (dubbed the *obfuscation sequence*). The obfuscation sequence is built such that at the end of its execution the target register holds the target value (the functionality of the program is not going to be affected by such constant obfuscation).

The obfuscation sequence is made of load, move, logical and arithmetic instructions. In particular, the obfuscation sequence is generated such that the target value is built step by step by all the instructions in the sequence and it will be available in the target register after the last instruction of the obfuscation sequence has been executed. The first step performed by the constant obfuscation module is substituting the target instruction with a randomly selected instruction having the target register as destination register. Then, being k the randomly chosen length of the obfuscation sequence, a sequence of k randomly selected instructions and associated operands is backward generated such that the result of each instruction is the value required by the subsequent instruction. It is worth mentioning that all the registers employed in the obfuscation sequence are chosen among the registers that are unused in the randomly chosen code block. As an example of constant obfuscation, Figure 4 reports a sequence of 13 instructions inserted in the program reported in Figure 2 to obfuscate the constant `32` used by the `addi` instruction at line 3.

A special case for the constant obfuscation procedure is represented by the `li` (load immediate) instructions. Indeed, the operand of an `li` instruction is a 32bit data, which would re-

```
lui  a5, 0
li   a6, -1379
li   a6, 737
slli a3, a6, 1
andi s2, a3, -1475
ori  t5, s2, 0
ori  s3, t5, 660
ori  s11, s3, -693
slli s5, s11, 0
slli s6, s5, 0
slli s1, s6, 0
slli s10, s1, 0
andi s4, s10, -131
andi t1, s4, -1886
srli s7, t1, 8
ori  s9, s7, 512
ori  t6, s9, 165
xori t2, t6, 685
slli t0, t2, 12
sll  a3, a6, t0
xori s2, a3, -1387
ori  t5, s2, 2
or   a5, a5, t5
```

Figure 5: An example of obfuscation of an `li` instruction referred to the original constant in line 5 of the program in Figure 2

```
addi  sp, sp,-32
mv    t1, sp
sd    s0, 24(t1)
addi  s0, t1, 32
```

Figure 6: An example of register scrambling referred to the code in lines 2 to 4 of the program in Figure 2 where `sp` is substituted with `t1`

quire a two long sequence to be effectively obfuscated. Thus, we split the `li` into an `lui` (load upper immediate) instruction, that loads the upper 16bit part of the operand, and an obfuscation sequence that calculates the remaining 16bit part of the operand. As an example of obfuscation of the operand of an `li` instruction, Figure 5 reports a sequence of 23 instructions inserted in the program reported in Figure 2 to obfuscate the constant `10` used by the `li` instruction at line 5.

The constant obfuscation module allows DETON to reduce the time for which a constant value is exposed in the microprocessor's registers, thus reducing sensitive data exposition. Moreover, since additional registers are used by the instructions in the obfuscation sequence, constant obfuscation allows to increase the usage of all the registers. Finally, as for garbage code insertion, also constant obfuscation adds instructions in the program, thus breaking specific instructions patterns whose identification could be of interest for the attacker.

4.4. Register scrambling

The last software manipulation performed by DETON is the scrambling of the registers within the program to be obfuscated. Each time it is invoked, this module randomly selects a program block and a target register r_i actually used in the selected block. Then, the *validity block* of r_i, i.e., all the code lines of the selected block where r_i is employed, is identified and a scrambling point, i.e., the specific instruction in the validity block after which introducing register scrambling, is randomly selected. Finally, a register r_j that is not used within the selected block

Table 1: The main parameters of the DETON framework

Name	Description
N_{gi}	# times the garbage code insertion module is invoked
L_{gi}	Max. length of the inserted garbage code sequence
N_{co}	# times the constant obfuscation module is invoked
L_{co}	Max. length of the inserted obfuscation sequence
N_{rs}	# times the register scrambling module is invoked

is identified (let refer to this register as the *scrambled register*), the scrambling instruction:

$$\texttt{mv } r_j\texttt{, } r_i$$

is added at the scrambling point and r_j is then substituted to r_i in all the remaining instructions of the validity block of r_i. As it has been previously mentioned, the scrambled register is chosen among the registers that are unused in the selected block. More in details, the unused register whose content has not been modified since more time is identified as the scrambled register; this allows to increase the usage of all the registers in the microprocessor as well as to reduce the time between two consecutive modifications of each register's content. As an example of register scrambling, Figure 6 reports the substitution of register `sp` with register `t1` in lines 2 to 4 of the example program in Figure 2.

4.5. Code generation

The output of the application of all the previously discussed obfuscation techniques (applied for the desired number of times) is a new CFG corresponding to the obfuscated version of the program. The very last step performed by DETON is the automatic generation of the assembly of the obfuscated program based on such obtained CFG.

4.6. DETON parameters

The functioning of DETON may be configured through the set of parameters reported and explained in Table 1. Of course, the larger N_{gi}, L_{gi}, N_{co} and L_{co}, the more obfuscated the obtained program. On the other hand, these parameters highly affect the overhead introduced in the final program. As it may appear straightforward, N_{rs} does not introduce overhead. Finding the best setting of such configuration parameters in order to maximize the obfuscation metrics while minimizing the introduced overhead would require a thorough design space exploration that falls outside the scope of this work.

5. Experimental analysis

We carried out a set of experiments to first validate the functionality of the obtained obfuscated programs, i.e., check whether the original and obfuscated programs are functionally equivalent, and to then assess the effectiveness and the efficiency of the proposed obfuscation framework based on the previously presented metrics.

Table 2: The considered benchmark programs

Program	#lines (Plain)	#lines (Protected)	Overhead
`Bubble`	109	189	73%
`CRC`	26	130	400%
`Dijkstra`	42	143	240%
`MatrMul`	99	255	158%
`QuickS`	115	224	95%
`SHA`	228	481	110%

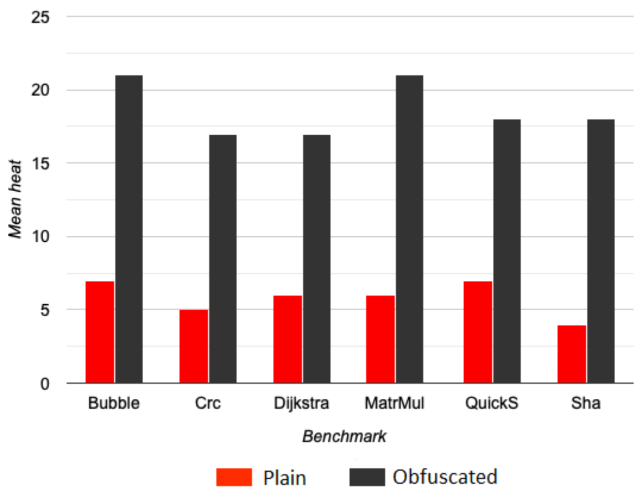


Figure 7: Aggregated average H_r values

5.1. Experimental setup

We implemented DETON as a set of Python scripts and C tools that read the assembly code of the original program, build the CFG, apply the considered obfuscation techniques and generate the assembly code of the obfuscated program. We targeted the 64bit Ariane RISC-V [31] microprocessor that counts 32 user registers, its ISA and toolchain and we considered a set of benchmark programs belonging to the well known MiBench suite [30]. In particular we considered: Bubblesort (`Bubble`), `CRC`, `Dijkstra`, Matrix multiplication (`MatrMul`), Quick sort (`QuickS`) and `SHA`. The first two columns of Table 2 report the program names and corresponding number of assembly lines of the original, unprotected version. Finally, we set the following configuration for the working parameters of DETON: N_{gi} 5, L_{gi} 10, N_{co} 5, L_{co} 20 and N_{rs} 50. Again, we point out that this was just an example configuration of DETON meant to assess it correctness, effeciency and effectiveness. An in depth exploration of the impact of such parameters on the introduced overhead falls outside the scope of this work.

5.2. Results

As a first validation note, after the generation of the obfuscated versions of the considered programs we ran a set of random simulations where the plain program and the corresponding obfuscated one have been fed with the same input. We highlight that the obfuscated programs demonstrated to be functionally equivalent to the corresponding plain ones, i.e., plain and

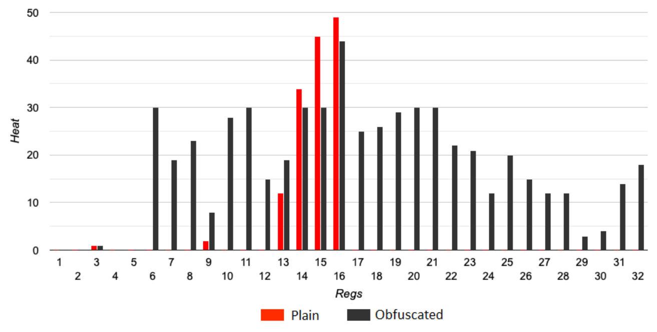


Figure 8: Register per register average H_r for SHA

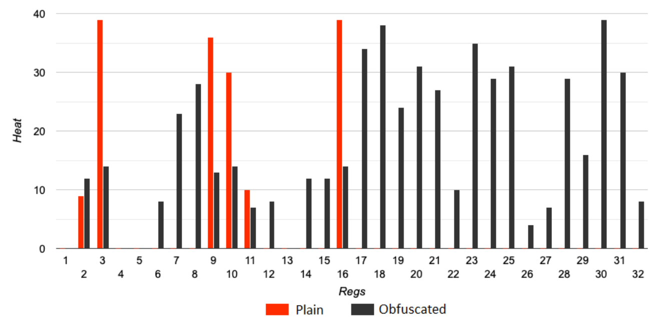


Figure 9: Register per register average H_r for CRC

obfuscated programs always produced the same output when fed with the same input.

When considering the effectiveness of DETON, if we look at Figure 7 we can see that the average aggregated registers' heat (calculated over the entire program execution and considering all the registers) is always much higher in the obfuscated program than in the plain one, with an average increase of about 225%. This actually demonstrates that all the registers are more (and more frequently) used in the obfuscated versions of the programs. More in details, the difference between the average H_r for the plain and obfuscated programs ranges between 177% for `Bubble` and 298% for `SHA`.

If we deepen the analysis for a specific program and we look at a register per register average H_r (calculated over the entire program execution) we have the confirmation that in the plain program only few registers are employed (and thus *hot*) while most registers are almost or totally *cold*. Conversely, in the obfuscated program, almost all registers are always used (and *hot*). Figures 8, 9 and 10 report this analysis for `SHA`, `CRC` and `Bubble`, respectively[3].

To even better visualize the modifications introduced by DETON, Figures 11, 12 and 13 report the heatmap representing the cycle-per-cycle heat of all registers during the executions of the plain and obfuscated `SHA`, `CRC` and `CRC` benchmarks, respectively. By analysing the proposed graphs, it appears evident how DETON allows to increase registers' use during the entire program execution. Moreover, we marked in blue the

[3]For the sake of space we show in the paper graphs regarding this analysis and the subsequent one only for `SHA`, `CRC` and `Bubble`.

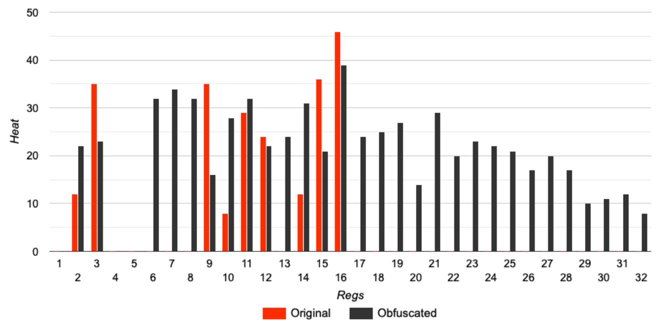


Figure 10: Register per register average H_r for `Bubble`

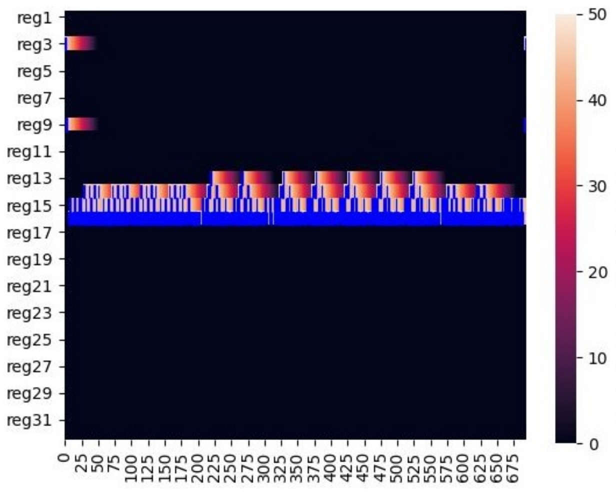
register write operation that are kept unaltered between the two versions of the program and in green those that are scrambled: again it is possible to see how effective DETON is in spreading the *original* data among registers.

Finally, if we look at the last two columns of Table 2 we can see the number of assembly instructions in the obfuscated program and the introduced program enlargement. The average overhead is about 180%[4], which is of course high, but, we believe, reasonable if we take into account that i) the proposed solution would highly alleviate the susceptibility of the system to information stealing HTHs, ii) the proposed solution is purely software and it does not require any modification/hardening to the hardware (thus allowing to deploy commercial legacy processing platforms instead of ad-hoc designed expensive and power-consuming ones), iii) no similar solutions working at the system level exist, and iv) no design space exploration for parameters optimization has been performed.

5.3. *Security analysis*

As it has been demonstrated by the presented experimental campaign, DETON is actually able to enlarge the set of registers employed during a program execution as well as to spread the sensitive information through several registers and instruction cycles. To carry out an information stealing attack, the attacker should be able to monitor a large set of registers and to monitor them for a long time. To do so, the attacker would be required to implant a large HTH, which would likely make the attack more expensive hard to be deployed. Moreover, even in the case the attacker is able to actually implant such HTH in the target CPU, the identification of the information of interest among all the received raw data would be much more difficult for the attacker given the introduction of garbage instructions and register scrambling. Therefore, we believe that DETON makes the implantation of information stealing HTHs and their exploitation harder thus moving in the direction of making software execution trusted also on possibly untrusted hardware platforms.

[4]The introduced program enlargement would of course bring also a energy consumption increase. Such additional energy consumption would not be proportional to the number of added instructions and its measurements would require an in depth analysis which falls outside the scope of this work.


(a) Plain

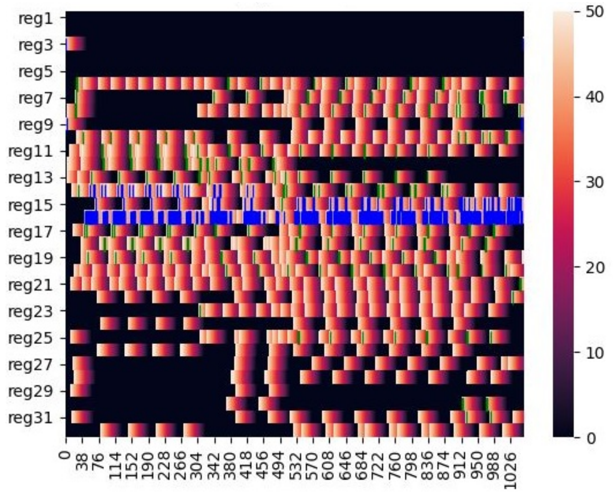
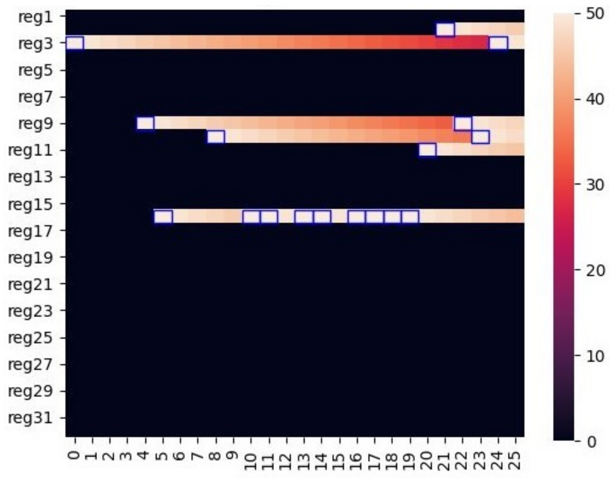

(b) Obfuscated

Figure 11: Application of DETON to `SHA`


(a) Plain

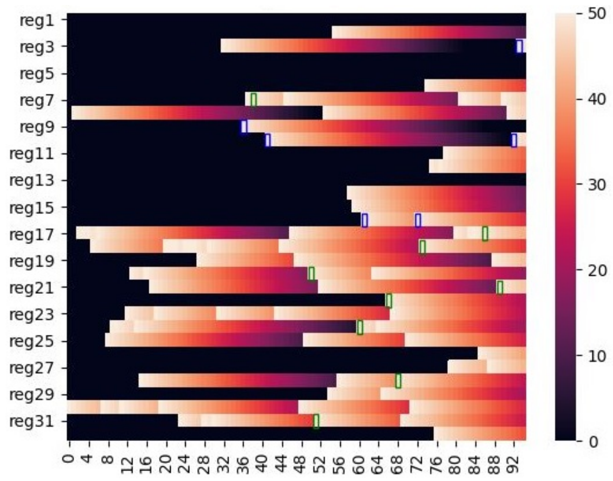

(b) Obfuscated

Figure 12: Application of DETON to `CRC`

On the other hand, as we previously discussed, no protection against change-functionality and denial-of-service HTHs is provided by DETON.

6. Related Work

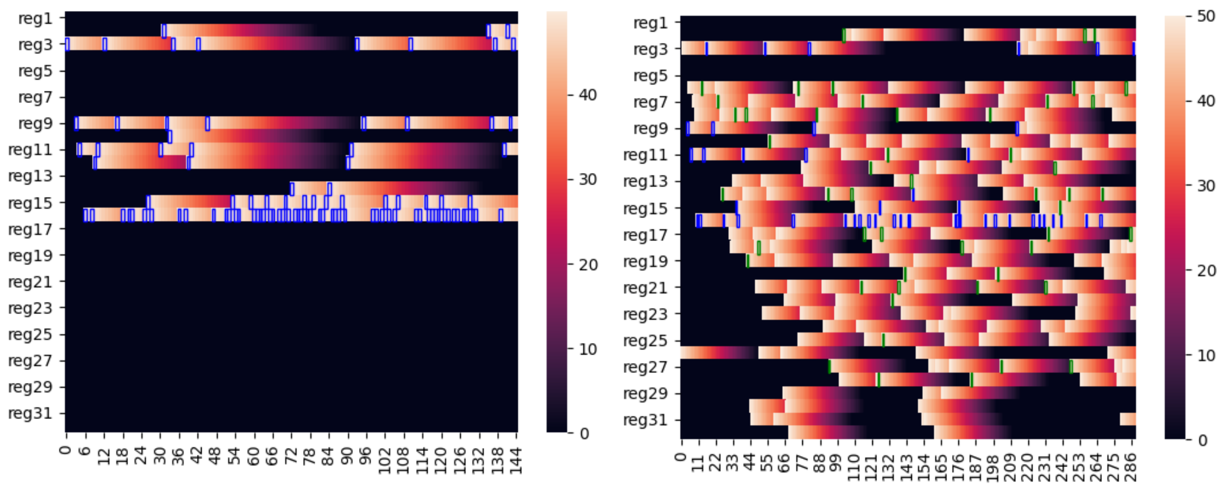
In the last two decades a very large number of techniques for protecting digital systems against HTHs has been proposed. More specifically, both detection and prevention techniques have been proposed, considering several attack scenarios, threat models and design stages where a HTH can be inserted [16]. We here classify these techniques into two main classes: *Design time* and *Runtime* detection techniques.

6.1. Design time techniques

Design time HTHs detection techniques may be further classified into: *optical inspection*, *logic testing*, *side-channel analysis*, *proof-carrying hardware* and *formal property verification* and *static (behavioral and structural) analysis*. In the following we give a brief overview of such techniques.

Optical inspection (also known as visual inspection) relies on the reverse engineering of the circuit to detect HTHs [20]. Accurate techniques for image acquisition and analysis are applied to obtain high resolution photos of the chip under analysis. The photos collected are then used to reconstruct the layout of the chip which is then compared with the layout of the original design. Optical inspection is suitable to detect HTHs inserted during fabrication by the foundry, but it cannot detect HTHs inserted during the design of the system. Moreover, optical inspection implies a very high cost and a long time to be applied. Finally, optical inspection requires the availability of a golden reference of the layout to be used for comparison against the layout obtained through reverse engineering.

Logic testing is the process that us usually applied to detect bugs/faults into chips after fabrication and before shipping.



(a) Plain

(b) Obfuscated

Figure 13: Application of DETON to `Bubble`

Since HTHs may be considered as a special class of fault, logic testing may also be used to detect the presence of HTHs into digital systems. Like for bug/fault detection, the main problem is generating efficient test patterns that are able to activate the HTH and then to make its effect visible [17]. Indeed, HTHs have usually an extremely low activation probability which makes them hard to trigger with standard test patterns. Several proposals have been made to improve the efficiency of logic testing for HTH detection [37]. The advantage of applying logic testing is that it is not invasive/destructive. Moreover, these techniques may be applied to detect testing inserted at any level of the design flow. On the other hand, there is no guarantee to generate test patterns able to activate and detect HTHs.

A promising perspective for HTHs detection has been formal verification. Two main families of silutions have been proposed in the last years: *proof-carrying hardware* and *property verification*. Proof-carrying hardware techniques, e.g., the ones presented in [21], rely on a set of security requirements defined by customer and system designer. After fabrication, the customer performs a verification of the previously agreed requirements before accepting the shipped chips. On the other hand, property verification techniques, as the ones presented in [18, 38], rely on a set of HTHs behavioral models defined by the customer. Once the netlist/system has been defined, a formal description of the system itself is generated and the presence/absence of the HTHs models within the system is formally proved. These techniques are not invasive/destructive and can be applied at any level of abstraction. The major drawback of proof-carrying hardware and formal verification is the need of an extremely accurate definition of the security requirements and a precise modeling of the functional behavior of the HTH which may be a very hard task. As a consequence, several strong assumptions on the HTH behavior are generally done on order to make such techniques feasible.

Side-channel analysis aims at detecting HTHs by analysing physical characteristics, e.g., power consumption, electro-magnetic emission, timing, of the device under analysis and by comparing them against a reference circuit [19]. The basic idea of such techniques is that the activation of a HTH in the circuit may alter the leakage current [39], the dynamic current [40] or some internal delays [41]. Like logic testing, also side-channel analysis is not invasive/destructive and it is theoretically able to detect HTHs inserted in any stage of the design process. On the other hand, side-channel analysis suffers from three main limitation: i) like for optical analysis, a golden reference of the circuit under analysis is required; ii) the monitored physical characteristics may be modified also by other factors, e.g., process variation, and not only by the HTH; and iii) the selected physical characteristic might be hard to measure precisely .

All the available design time HTH detection techniques suffer from several applicability or effectiveness limitations which make the deployed system still vulnerable. As discussed in [22], there is a lack of techniques able to prevent, detect and tolerate the activation of a HTH at runtime, without requiring the availability of a circuit golden copy, without needing strong assumptions on the considered HTH model and without destroying the circuit.

6.2. *Runtime techniques*

More recently both hardware- and software-based runtime solutions for HTHs prevention and detection have been proposed [25, 26, 29]. In [24, 25], multiple functionally equivalent copies of the same 3PIP belonging to different vendors (and thus structurally different) are employed to compose a trustworthy system from untrusted components. In [42] Bloom filter-based checking modules are included in the HW architecture between the microprocessor and the memories to detect the activation of HTHs within the memory elements. Similirly, in [43] security checking modules are introduced between the fetching unit and the instruction but to detect the activation of change-functionality HTHs that aim at interfering with the fetching activity of the microprocessor. In [26, 27, 28], tasks are scheduled in such a way that those 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 HTHs. The problem of this approaches is that their applicability is limited to those scenarios where the hardware platform is still to be developed and the designer has the freedom to add redundancy and diversity. A genetic algorithm-based software obfuscation techniques aimed at preventing the activation of triggered HTHs has been proposed in[29]. Finally, a set of checks specifically designed to detect HTHs have been included in the operating system in [44].

7. Conclusions and future work

We presented DETON, an automatic methodology for software obfuscation aimed at protecting program execution over possibly untrusted microprocessor-based systems against information stealing HTHs. The correctness, effectiveness and efficiency of DETON have been assessed on the Ariane 64bit RISC-V microprocessor running a set of MiBench programs.

As future work we plan to define a larger set of metrics to assess the strength of software obfuscation against HTHs and to exploit design space exploration to identify the best configurations of DETON to guarantee security while not introducing excessive overhead.

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, R. Karri, Hardware security: Threat models and metrics, in: Proc. Int. Conf. Computer-Aided Design, 2013, pp. 819–823.

[3] M. Tehranipoor and C. Wang, Introduction to Hardware Security and Trust, Springer-Verlag New York, 2012.

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

[5] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, Y. Makris, Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain, Proc. IEEE 102 (2014) 1207–1228.

[6] A. Carelli, C. A. Cristofanini, A. Vallero, C. Basile, P. Prinetto, S. Di Carlo, Securing bitstream integrity, confidentiality and authenticity in reconfigurable mobile heterogeneous systems, in: 2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2018, pp. 1–6.

[7] M. Tehranipoor, F. Koushanfar, A survey of hardware trojan taxonomy and detection, IEEE Design & Test of Computers 27 (2010).

[8] A. Bhardwaj, S. K. Roy, Defeating hatch: Building malicious ip cores, in: International Symposium on VLSI Design and Test, Springer, 2017, pp. 345–353.

[9] V. Jyothi, P. Krishnamurthy, F. Khorrami, R. Karri, Taint: Tool for automated insertion of trojans, in: 2017 IEEE International Conference on Computer Design (ICCD), 2017, pp. 545–548.

[10] G. T. Becker, F. Regazzoni, C. Paar, W. P. Burleson, Stealthy dopant-level hardware trojans, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2013, pp. 197–214.

[11] X. Wang, T. Mal-Sarkar, A. Krishna, S. Narasimhan, 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), 2012, pp. 55–58.

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

[13] N. G. Tsoutsos, M. Maniatakos, Fabrication attacks: Zero-overhead malicious modifications enabling modern microprocessor privilege escalation, IEEE Trans. Emerging Topics in Computing 2 (2014) 81–93.

[14] 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.

[15] https://github.com/xoreaxeaxeax/rosenbridge, ????

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

[17] M. L. Flottes, S. Dupuis, P. S. Ba, B. Rouzeyre, On the limitations of logic testing for detecting hardware trojans horses, in: Proc. Int. Conf. Design Technology of Integrated Systems in Nanoscale Era, 2015, pp. 1–5.

[18] J. Zhang, F. Yuan, L. Wei, Y. Liu, Q. Xu, Veritrust: Verification for hardware trust, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 34 (2015) 1148–1161.

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

[20] F. Courbon, P. Loubet-Moundi, J. J. A. Fournier, A. Tria, A high efficiency hardware trojan detection technique based on fast sem imaging, in: Proc. Design, Automation &#38; Test in Europe Conf., 2015, pp. 788–793.

[21] E. Love, Y. Jin, Y. Makris, Proof-carrying hardware intellectual property: A pathway to trusted module acquisition, IEEE Trans. Information Forensics and Security 7 (2012) 25–40.

[22] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, M. Tehranipoor, Hardware trojans: Lessons learned after one decade of research, ACM Trans. Design Automation of Electronic Systems 22 (2016) 6:1–6:23.

[23] T. Hoque, P. Slpsk, S. Bhunia, Trust issues in microelectronics: The concerns and the countermeasures, IEEE Consumer Electronics Magazine (2020) 1–1.

[24] R. Kalayappan, S. R. Sarangi, Seccheck: A trustworthy system with untrusted components, in: 2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2016, pp. 379–384.

[25] J. J. Rajendran, O. Sinanoglu, R. Karri, Building trustworthy systems using untrusted components: A high-level synthesis approach, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 24 (2016) 2946–2959.

[26] C. Liu, J. Rajendran, C. Yang, R. Karri, Shielding heterogeneous mpsocs from untrustworthy 3pips through security- driven task scheduling, IEEE Trans. Emerging Topics in Computing 2 (2014) 461–472.

[27] N. Wang, M. Yao, D. Jiang, S. Chen, Y. Zhu, Security-driven task scheduling for multiprocessor system-on-chips with performance constraints, in: 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2018, pp. 545–550.

[28] A. Malekpour, R. Ragel, T. Li, H. Javaid, A. Ignjatovic, S. Parameswaran, Hardware trojan mitigation in pipelined mpsocs, ACM Trans. Des. Autom. Electron. Syst. 25 (2020). URL: `https://doi.org/10.1145/3365578`. doi:`10.1145/3365578`.

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

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), 2001, pp. 3–14.

[31] F. Zaruba, L. Benini, The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 27 (2019) 2629–2640.

[32] M. Rostami, F. Koushanfar, R. Karri, A primer on hardware security: Models, methods, and metrics, Proceedings of the IEEE 102 (2014) 1283–1295.

[33] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, V. Leppänen, Diversification and obfuscation techniques for software security: A systematic literature review, Information and Software Technology 104 (2018) 72–93.

[34] J. Demme, R. Martin, A. Waksman, S. Sethumadhavan, Side-channel vulnerability factor: A metric for measuring information leakage, in:

2012 39th Annual International Symposium on Computer Architecture (ISCA), 2012, pp. 106–117.

[35] J. Wichelmann, A. Moghimi, T. Eisenbarth, B. Sunar, Microwalk: A framework for finding side channels in binaries, in: Proceedings of the 34th Annual Computer Security Applications Conference, 2018, p. 161–173.

[36] S. Chattopadhyay, M. Beck, A. Rezine, A. Zeller, Quantifying the information leakage in cache attacks via symbolic execution, ACM Trans. Embed. Comput. Syst. 18 (2019).

[37] X. Chuan, Y. Yan, Y. Zhang, An efficient triggering method of hardware trojan in aes cryptographic circuit, in: Proc. Int. Conf. Integrated Circuits and Microsystems, 2017, pp. 91–95.

[38] N. Veeranna, B. C. Schafer, Hardware trojan detection in behavioral intellectual properties (ip's) using property checking techniques, IEEE Transactions on Emerging Topics in Computing 5 (2017) 576–585.

[39] M. Potkonjak, A. Nahapetian, M. Nelson, T. Massey, Hardware trojan horse detection using gate-level characterization, in: 2009 46th ACM/IEEE Design Automation Conference, IEEE, 2009, pp. 688–693.

[40] H. Salmani, M. Tehranipoor, Layout-aware switching activity localization to enhance hardware trojan detection, IEEE Transactions on Information Forensics and Security 7 (2011) 76–87.

[41] Y. Jin, Y. Makris, Hardware trojan detection using path delay fingerprint, in: 2008 IEEE International workshop on hardware-oriented security and trust, IEEE, 2008, pp. 51–57.

[42] A. Bolat, L. Cassano, P. Reviriego, O. Ergin, M. Ottavi, A microprocessor protection architecture against hardware trojans in memories, in: 2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS), 2020, pp. 1–6.

[43] 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 34th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2021, pp. 1–6.

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