# Built-in Software Obfuscation for Protecting Microprocessors against Hardware Trojan Horses

Alessandro Palumbo, Marco Ottavi, Luca Cassano

2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) | 979-8-3503-1500-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/DFT59622.2023.10313534

# Built-in Software Obfuscation for Protecting Microprocessors against Hardware Trojan Horses

Alessandro Palumbo[a], Marco Ottavi[b,c], Luca Cassano[a]
[a]Politecnico di Milano, Italy, [b]University of Rome Tor Vergata, Italy, [c]University of Twente, The Netherlands
[a]{name.surname}@polimi.it.it, [b,c]m.ottavi@utwente.nl

***Abstract*—Hardware Trojan Horses (HTHs) are today a serious issue for both academy and industry because of their dramatic complexity and dangerousness. Indeed, it has been shown that HTHs may be effectively inserted in modern microprocessors allowing the attacker to run malicious software, to acquire root privileges and to steal secret information. We aim at reducing the dangerousness of information stealing HTHs by introducing a hardware security module in the microprocessor under protection. In particular, the proposed module is in charge of interacting with the execution flow in order to introduce software obfuscation during programs execution at runtime. The goal of such obfuscation is to minimize the probability of exposing sensitive information to the HTH by encrypting/decrypting it, by spreading it through microprocessor's registers and by submerging it among garbage data. We implemented a prototype of the proposed hardware security module and we proved its effectiveness and efficiency (in terms of area occupation and working frequency reduction) by integrating it into the RSD 32bit speculative, superscalar and out-of-order RISC-V microprocessor running a set of benchmark programs[1].**

***Index Terms*—Design for Trust, Hardware Security, Hardware Trojan Horses, Microprocessors, Software Obfuscation**

## I. Introduction and Related Work

The design and fabrication process of modern microprocessors continuously seeks for more complexity and performance while keeping low production cost and short time-to-market. These needs pushed the integrated circuits (ICs) market 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]. While, on the one hand, such a globalized supply chain allowed for a dramatic reduction of design cost and time, on the other hand, it caused a significant loss of trust in the final delivered ICs [3]. It is indeed very hard to ensure the trustworthiness of all the parties involved in the supply chain; therefore, the product is exposed to several threats. One of the security threats that raised in the last years is represented by Hardware Trojan Horses (HTHs) [4], [5].

A HTH can be defined as a very-hard-to-detect malicious modification of a digital circuit. HTHs are generally meant to stay silent most of the time and to activate in specific and usually rare working conditions. Once a HTH is activated it can alter or halt the nominal behavior of the system or steal secret information [4]. HTHs may be inserted in any 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 [6]; rogue employees and untrusted CAD tools may maliciously modify the design [7], [8]; finally, untrusted mask providers and silicon foundries may alter the layout of the system [9].

HTHs have been considered as a purely academic issue for a long time. Indeed, they generally exposed limited complexity and, as a consequence, reduced dangerousness. On the other hand, in the very last years, a new menace raised: the *software exploitable HTHs* [10]. Complex and highly dangerous HTHs may be inserted in real-world microprocessors allowing the attackers to execute their own malicious software, to modify the running software, to acquire unauthorized privileges or to steal secret information [11], [12]. Recently, security researchers found a hardware backdoor, the *Rosenbridge* backdoor, in a commercial Via Technologies C3 processor [13]. This hardware backdoor can be activated and exploited via software to enter the supervisor mode of the system[2]. The feasibility of implanting and activating such extremely dangerous software exploitable HTHs in real-world microprocessors makes these attacks a severe concern not only for academy but also for industry.

In the last years a vast literature about HTHs detection methodologies has been produced [14]. Most of the existing approaches attempt to identify the presence of HTHs in the system under analysis before deployment, by exploiting a plethora of techniques, e.g., logic testing, formal property verification, side-channel analysis, optical inspection, proof-carrying hardware. All these techniques suffer from a number of limitations: first of all the difficulty of triggering the HTHs at design time but also the need for a golden reference of the circuit under analysis. Recently, a new paradigm raised: the so-called *Design for Trust (DfT)* [15]: the idea is to develop HTHs tolerance techniques that allow to build trusted systems from untrusted components or to provide trusted execution over untrusted systems. Existing DfT approaches are based on the integration of redundant functionally-equivalent IP cores belonging to different IP vendors, like in [16], or on the deployment of ad-hoc checkers working in parallel with the core under protection, like in [17], [18]. Finally, security-aware task scheduling for systems composed of redundant IP cores belonging to different vendors has been proposed [19].

In this paper we present a built-in software obfus-

[1]This work has been partially carried out when Alessandro Palumbo was a Ph.D. student at the University of Rome Tor Vergata, Italy.

979-8-3503-1500-4/23/$31.00 ©2023 IEEE

[2]After the publication of [13] Via Technologies officially commented that this behavior was due to an undocumented feature meant for debug.

cation methodology for mitigating the dangerousness of information-stealing HTHs in microprocessors. The idea is to introduce a hardware security module between the decode and the execute units of the pipeline of the microprocessor under protection. Such security module is in charge of interacting with the execution flow in order to obfuscate the executed software at runtime. The goal of such obfuscation is to minimize the probability of exposing sensitive information to the HTH by encrypting/decrypting it, by spreading it through microprocessor's registers and by introducing garbage instructions and data. We implemented a prototype of the proposed hardware security module and we proved its efficiency by integrating it into the RSD RISC-V core [20]. The proposed security solution introduces about 10% area overhead and no working frequency reduction. Moreover, we measured the effectiveness of the proposed security solution by running a set of benchmark programs.

The use of software obfuscation for defeating HTHs has been recently proposed in [21], [22], [23], [24]. The works in [21] and [23] propose compile-time software obfuscation methodologies (and its optimization in [22]) to be applied before deploying the program onto the final system. As a consequence of being applied at compile-time, all these techniques cannot afford the obfuscation of loops/jumps and subprograms. On the other hand, the work in [24] proposes a built-in software obfuscation to be integrated within the core under protection thus running during the execution of the program. For this reason, we believe that the work most similar to our proposal is indeed [24]. On the other hand, the authors of [24] achieve software obfuscation by only substituting program instructions with equivalent ones, while we consider more complex software manipulations. Moreover, the proposal in [24] considers sequentially-triggered change-functionality HTHs while we consider always-on information stealing HWHs.

The remainder of this paper is organized as follows: Section II discusses the considered threat model and some background about design obfuscation; Section III presents the proposed software obfuscation architecture while Section IV presents some results and draws some security-related considerations; Finally, Section V concludes the paper.

## II. Background

### A. The Considered Threat Model

Referring to the classical classification of HTHs [4], our proposal takes into account both triggered and always-on HTHs that aim at stealing information from the infested system. From the location point of view, we consider HTHs infesting microprocessor's logic, inserted by a malicious IP provider when selling the microprocessor. On the other hand, we assume that the design team of the company that purchases the microprocessor and the employed foundry are trusted, therefore, we assume that the introduced security checker cannot be infested by HTHs.

We assume a two-level information stealing attack: in the first phase, the HTH repeatedly exfiltrates raw data from a number of registers of the processor and covertly sends it to the attacker; in the second phase, the attacker collects such raw data and post-processes it to retrieve sensitive information. We may reasonably assume that, when injecting the HTH, the attacker knows all the details of the hardware platform under attack. Moreover, we assume that the attacker has a rough idea about which operating system and programs will be executed but that he/she cannot have all the details about software versions and implementations.

We also assume that the injected HTH monitors and exfiltrates raw data from a reduced number of registers of the processor. We believe that this assumption is reasonable if we consider that: i) HTHs need to be small enough not to be detected via optical inspection, ii) HTHs need to have an extremely reduced impact on power consumption, electromagnetic emission and timing, and iii) HTHs cannot occupy the transmission channel for long without being discovered. Therefore, we assume a HTH model that is able to monitor the content of a fixed (at design-time) and small set of registers and exfiltrates 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 assume that the attacker knows all the details of the deployed built-in software obfuscation methodology and that this information 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.

### B. Design Obfuscation

Obfuscation is a widely used technique for protecting both hardware [25] and software [26]. The goal of obfuscation is generally to protect the intellectual property associated with a program or a circuit from unauthorized use or reproduction. Obfuscation of the hardware has been proposed to avoid i) reverse engineering of the circuit's functionality by observing the netlist or of the circuit's netlist by observing the layout and ii) overproduction of unauthorized copies of a circuit. Obfuscation of the hardware is generally achieved through the use of non-standard cells (*camouflaging*) or by "locking" the netlist in order to make the fabricated circuit unusable before unlocking it through a secret key (*logic locking*).

Like for hardware obfuscation, also software obfuscation aims at making intellectual property break unfeasible. For example, a software may be obfuscated to make hard for a reader (for example a decompilation tool) to understand the functionality implemented by the program, the meaning of a given construct or variable, the value of constants, the structure of classes and arrays. Software obfuscation may be achieved by inserting never-executed dummy code, by reordering or hiding instructions, by unrolling, intersecting and extending loops, by opacifying logic conditions and by splitting and merging arrays and data structures.

## III. Built-in Software Obfuscation

### A. Guidelines for anti-HTH software obfuscation

Given the previous discussion on design obfuscation, we argue that so far, obfuscation has always been meant to deal with a *static external* attacker that analyses the circuit or the software to gain knowledge to be then able to reproduce

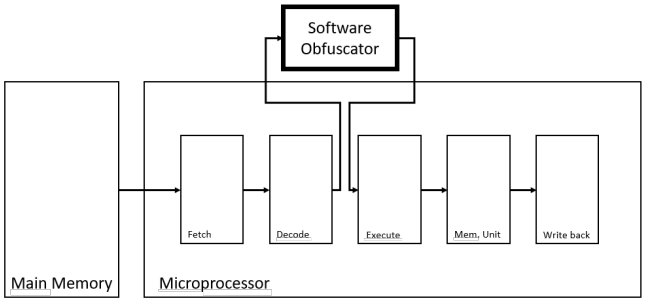


Figure 1: The architecture of the secured system including the built-in software obfuscation module

```
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
```

(a)

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

(b)

Figure 2: Example program (a) and garbage insertion (b)

it. Conversely, we believe that information stealing HTHs represent a *dynamic internal* threat. Indeed, the considered HTHs observe the behavior of the running program from inside with the goal of discovering the sensitive information processed by the running program at runtime. Because of these peculiarities, we believe that novel obfuscation guidelines have to be drawn to deal with information stealing HTHs. Therefore, we defined the following anti-HTH software obfuscation guidelines.

***Guideline 1***: Those variables of the program under protection that need to be hidden should reside in the largest possible set of microprocessor registers during program execution. As a consequence, the probability of exposing sensitive variables to the attacker through the small set of registers monitored by the HTH is kept small.

***Guideline 2***: Those variables of the program under protection that need to be hidden should also be encrypted in the processor's registers for as long as possible; these variables should be decrypted only when they need to be processed and then be encrypted again as soon as possible. As a consequence, again, the probability of exposing sensitive variables to the attacker is kept small.

***Guideline 3***: The amount of non sensitive information per time unit processed by the program under protection should be kept as large as possible. As a consequence, again the probability of exposing the sensitive variables to the attacker is kept small.

The last guideline is not related to security but to efficiency: it is of extreme importance not to excessively degrade performance while ensuring security. In other words, it is mandatory not to excessively increase the execution time of the program under protection w.r.t. the original unprotected version of the program.

### *B. The built-in software obfuscation architecture*

Our solution, whose high-level representation is depicted in Figure 1, relies on the insertion of a dedicated hw module, called the *Software Obfuscator (SOb)*, in the microprocessor under protection. In particular, the SOb implements the proposed built-in software obfuscation methodology to reduce the amount of significant information exposed to HTHs. From a high-level point of view, the strategy adopted by the SOb aims at: i) spreading sensitive information through microprocessor's registers, ii) keeping sensitive information encrypted for as long as possible, iii) submerging sensitive information among garbage data, and iv) periodically scrambling data among microprocessor's registers. In this way, considering that the adopted HTH model is able to monitor and send to the attacker the content of a subset of processor's registers, the proposed methodology achieves three benefits: i) minimizing the amount of exposed sensitive information, ii) maximizing the amount of exposed garbage information, and iii) minimizing the time for which sensitive information is kept in the same register.

As it can be observed in Figure 1, the SOb is inserted between the *Decode* and the *Execute* units of the pipeline. Therefore, the SOb takes in input a decoded instruction and produces in output one or more instructions to be executed. The output of the SOb may be the very same instruction as the one received in input, in case no obfuscation is inserted in that specific execution instant of time, or one or more *obfuscation instructions*. The obfuscation instructions allow the SOb to implement the three considered obfuscation policies (whose details are presented in the following), namely i) **garbage code insertion**, ii) **variable xoring**, and iii) **register scrambling**. After every instruction has been fetched and decoded, the SOb randomly decides whether obfuscation has to be inserted or not, and in case, which obfuscation techniques have to be applied. When inserting one or more obfuscation instructions, the SOb stores the value of the *Program Counter* register before the insertion; in this way, when the execution of the obfuscation instructions terminates, the execution flow of the *nominal* program can be correctly restored (such restoring of the content of the program counter is obviously not required when no obfuscation instruction is inserted, since in this condition the SOb does not interfere with the nominal program).

All the software obfuscation examples in the following are referred to the intentionally simple program shown in Figure 2a that creates two variables and calculates their sum.

#### Garbage code insertion

When garbage code insertion is randomly activated, the

SOb forces the execution of a random number of randomly selected instructions before the execution of the instruction received in input from the decode unit. 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. Since also operands are random, to prevent anomalous working conditions no division instructions (to avoid possible divisions by zero) and no jump instructions (to avoid jumping into unauthorized memory areas) are inserted.

To avoid altering the correct execution of the nominal program, the registers in which garbage instructions write their results are chosen among the unused registers. By applying such garbage code insertion, we maximize the usage of all the registers as well as we break specific instructions patterns whose identification during program execution could be of interest for the attacker. As an example of garbage code insertion, Figure 2b reports six garbage instructions inserted between lines 4 and 5 of the program reported in Figure 2a.

**Variable xoring**

The second random manipulation that the SOb is able to perform is the variable encryption/decryption by *masking* it through xor operations. When this manipulation is activated on a given value, the SOb will encrypt the value by xoring it with a randomly chosen key every time the value is written in a register and it will decrypt the value by xoring again it with the same key every time the value is read. To correctly carry out this task the SOb keeps constantly updated a map of the registers where encrypted data are tracked, and for each of them, the used key is also stored. We randomly chose whether to encrypt a variable or not instead of encrypting all variables, because this increases the amount of *confusion* we introduce during the execution of the program. By means of this variable xoring, we minimize the time during which sensitive information are exposed to the HTH.

**Register scrambling**

The last software manipulation implemented by the SOb is the random scrambling of the information processed by the executed program among the registers in the microprocessor. Each time it is activated, this manipulation randomly selects a *target register* $r_i$ among the registers currently used by the program as well as a *scrambling register* $r_j$ among the unused registers. Then, the execution of the *scrambling instruction*

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

is forced. Then, every time the instruction fed in input to the SOb refers to a register that previously went through register scrambling, i.e., $r_i$ in the example, the SOb will force the execution of the very same instruction but where the target register is substituted with the corresponding scrambling register, i.e., $r_j$ in the example. In order to properly carry out such software manipulation, the SOb needs to keep track of all the previously performed scramblings in terms of a binding between the target register and the corresponding scrambling register. Indeed, since no modification of the code stored in the memory is performed, the instructions fetched after the register scrambling will still refer to the original target register. This software manipulation allows to maximize the usage of all the registers in the microprocessor, thus reducing the probability that a HTH that monitors a subset of the registers may observe sensitive information.

Table I: The considered benchmark programs

| Program | Avg clk (unprotected) | Avg clk (Protected) | Avg Overhead |
|---|---|---|---|
| `RSort` | 21,238 | 48,284 | 127% |
| `QSort` | 247,620 | 428,518 | 73% |
| `Blowfish` | 1,031,302 | 1,504,890 | 46% |
| `Median` | 13,722 | 19,256 | 40% |
| `Coremark` | 686,700 | 1,523,565 | 121% |
| `RC4` | 51,582 | 98,153 | 90% |

## IV. Experimental analysis

### A. Experimental setup

We implemented the proposed software obfuscation hardware module in VHDL and we integrated it into the RSD core [20] which is a 32-bit, speculative, out-of-order, superscalar, two-fetch front-end and five-issue back-end pipelines RISC-V core with 16KByte instruction cache developed at the University of Tokyo. We considered a set of benchmark programs belonging to the well known MiBench suite [27] and from the official RISC-V suite. In particular we considered: `RSort`, `QSort`, `Blowfish`, `Median`, `Coremark` and `RC4`. For every benchmark program we ran 100 executions where the input have been randomly generated and corresponding number of required clock cycles has been calculated. The first two columns of Table I report the program names and the average number of clock cycles required to complete the unprotected executions.

### B. The adopted metrics

Based on the previously discussed design guidelines for security-aware software obfuscation, we adopted the following effectiveness and efficiency metrics to evaluate the proposed solution.

First of all, in order to assess the registers usage, we calculate the percentage of the registers in the microprocessor that are written at least once during program execution; we call this metric $R$. Then, since to reach a *satisfactory* obfuscation it is not enough to write all registers at least once, but also to write all of them for almost the same number of times, for every register $r_i$ we calculate the percentage number of writings performed in $r_i$ over the total number of writings performed by the program; given register $r_i$, we call this metric $R_i$. Then, in order to capture how much uniformly all registers are written, we measure the standard deviation of all these $R_i$ values; we call this metric $S$. Finally, we measure the percentage of clock cycles in which a variable of the program is stored encrypted in the processor's registers over the total number of clock cycles; for a given variable $v_i$ we call this metric $X_i$. We then calculate the average of all the $X_i$ values and we call this metric $X$.

Moreover, to assess the overhead introduced in the protected program execution, we also measure the percentage increase of the number of clock cycles required to complete program execution.

Table II: Effectiveness metrics values

| Program | Unprotected | | | Protected | | |
|---|---|---|---|---|---|---|
| | $R$ | $S$ | $X$ | $R$ | $S$ | $X$ |
| RSort | 75% | 0.076 | 0% | 100% | 0.016 | 90% |
| QSort | 59% | 0.061 | 0% | 100% | 0.009 | 98% |
| Blowfish | 66% | 0.070 | 0% | 100% | 0.009 | 68% |
| Median | 47% | 0.055 | 0% | 100% | 0.008 | 98% |
| Coremark | 94% | 0.052 | 0% | 100% | 0.008 | 98% |
| RC4 | 56% | 0.078 | 0% | 100% | 0.014 | 98% |
| Avg | 66% | 0.065 | 0% | 100% | 0.010 | 92% |

### C. Results

As a first validation note, we highlight that the proposed protection architecture does not alter the nominal functionality of the program. For every considered benchmark, we ran a set of 100 executions where the same randomly generated input was fed into the unprotected program and into the protected one and the result has been that in all cases the protected programs demonstrated to be functionally equivalent to the corresponding unprotected ones, i.e., given the same input the two programs produced the same output.

As for the unprotected executions of the considered programs, also the protected ones, i.e., when the proposed protection architecture was enabled, have been ran 100 times, each one with randomly generated input values. Therefore all the metrics we report below have to be considered as average values over the 100 executions.

Table II reports the values of the previously presented effectiveness metrics for the considered programs in the unprotected and in the protected scenarios. First of all, by looking at the $R$ columns, it has to be highlighted that in the protected executions always 100% of processor's registers are written at least once, while when considering the unprotected executions, on average only 66% of the registers is written.

When looking at the second adopted metric, i.e., the standard deviation of the register-per-register percentage of write operations, reported in the $S$ columns, it can be observed that in the unprotected program executions this value is most of the time one order magnitude larger than in the protected executions. This means that the proposed obfuscation technique is actually able to uniformly write all registers in the microprocessor. To better highlight this result in Figure 3 we show the detailed report of the per-register percentage of write operations for `Blowfish` (we could not report the graphs for all the benchmarks for the sake of space). As expected, the unprotected executions write only few registers most of the time (the blue spikes) while most registers are written very rarely or even never written. On the other hand, the protected executions write all registers for an almost uniform number of times, i.e., the orange bars are almost the same for all registers.

The last result reported in Table II (see the $X$ columns) is that the sensitive information of the program under protection stay most of the time encrypted in the processor's registers (92% of the time on average). On the other hand, in the unprotected executions, the sensitive information are always stored uncrypted in the registers.

Finally, if we look at the last two columns of Table I we can see the average number of clock cycles required to complete the obfuscated executions and the associated introduced clock cycles overhead. The average overhead is about 82%[3], 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 does not require any modification to the adopted microprocessor (thus allowing to deploy commercial legacy processing platforms), iii) no similar solutions exist, and iv) no design space exploration to optimize the obfuscation capability and the corresponding introduced overhead has been performed.

### D. Hardware overhead

We measured the hardware overhead introduced in the system by the proposed protection architecture. The considered RSD RISC-V microprocessor has been synthesized on a Virtex7 `xc7z020clg484-1` FPGA device: it worked at 57MHz and it counted 18,334 LUTs, 10,885 FFs and 4,512 LUT-RAMs. The software obfuscator on which the proposed solution relies counts 2,640 LUTs, 1,498 FFs and 24 LUT-RAMs, therefore, the introduced area overhead is of about 9.56% which, we believe, is totally reasonable. On the other hand, the software obfuscator does not introduce any reduction of the microprocessor working frequency.

### E. Security analysis

The proposed built-in software obfuscation architecture 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 and to encrypt the sensitive information for most of the execution time. To effectively carry out an information stealing attack by exploiting a HTH, the attacker should be able to monitor a much larger set of registers w.r.t. the original program and to monitor them for a longer time. This requires to implant a larger HTH which would transmit much more data. Moreover, since most information is encrypted, but some is not, it would be difficult for the attacker to understand and manipulate the received data. Finally, the identification of the sensitive information among all the received data would be much more difficult for the attacker. Given all these considerations, we believe that the proposed built-in software obfuscation architecture makes the implantation of information stealing HTHs and their exploitation harder.

## V. Conclusions and Future Work

We presented a built-in software obfuscation architecture to protect microprocessor-based systems against information stealing HTHs. The effectiveness and efficiency of our proposal have been assessed on the RSD 32bit RISC-V microprocessor running a set of benchmark programs. The experiments demonstrated that the proposed architecture is actually able to protect the execution of the programs while introducing a limited overhead in terms of program execution time and resource occupation, and with no overhead in terms of the working frequency of the protected microprocessor.

[3]The overhead in terms of clock cycles would bring also an energy consumption increase. Such additional energy consumption would not be proportional to the number of additional clock cycles and its measurement requires an in depth analysis which falls outside the scope of this work.

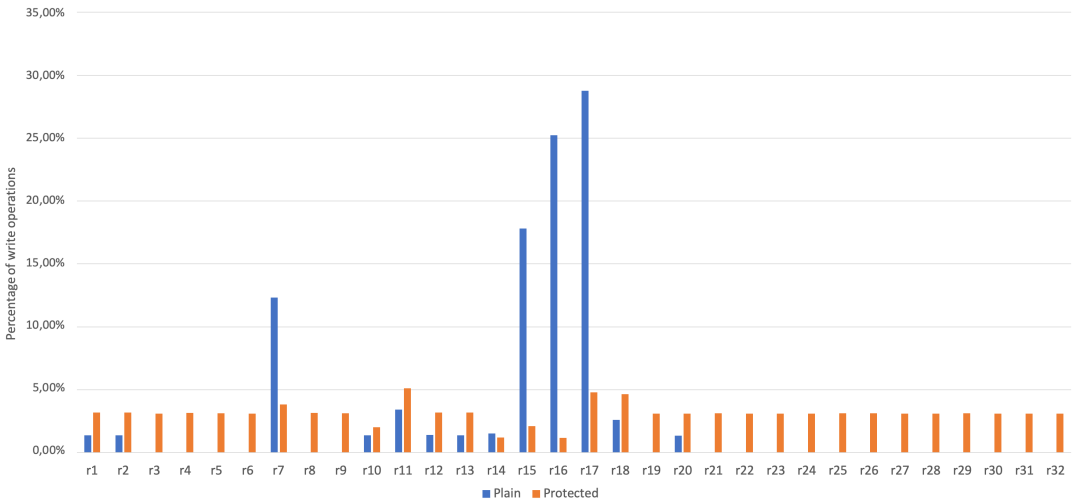


Figure 3: Per-register percentage of write operations for `Blowfish`

## 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*, 2013, pp. 819–823.

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

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

[5] L. Cassano, S. D. Mascio, A. Palumbo, A. Menicucci, G. Furano, G. Bianchi, and M. Ottavi, "Is risc-v ready for space? a security perspective," in *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2022, pp. 1–6.

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

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

[8] Alessandro Palumbo, Luca Cassano, Bruno Luzzi, José Alberto Hernández, Pedro Reviriego, Giuseppe Bianchi, Marco Ottavi, "Is your fpga bitstream hardware trojan-free? machine learning can provide an answer," *To appear in Journal of Systems Architecture*.

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

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

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

[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] 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.

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

[15] 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*, vol. 22, pp. 6:1–6:23, 2016.

[16] J. J. Rajendran, O. Sinanoglu, and R. Karri, "Building trustworthy systems using untrusted components: A high-level synthesis approach," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 9, pp. 2946–2959, 2016.

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

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

[19] A. Malekpour, R. Ragel, T. Li, H. Javaid, A. Ignjatovic, and S. Parameswaran, "Hardware trojan mitigation in pipelined mpsocs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 25, no. 1, Jan. 2020. [Online]. Available: https://doi.org/10.1145/3365578

[20] S. Mitsuno, J. Kadomoto, T. Koizumi, R. Shioya, H. Irie, and S. Sakai, "A high-performance out-of-order soft processor without register renaming," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 73–78.

[21] L. Cassano, M. Iamundo, T. A. Lopez, A. Nazzari, and G. Di Natale, "Deton: Defeating hardware trojan horses in microprocessors through software obfuscation," *Journal of Systems Architecture*, vol. 129, p. 102592, 2022.

[22] L. Cassano, E. Lazzeri, N. Litovchenko, and G. Di Natale, "On the optimization of software obfuscation against hardware trojans in microprocessors," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2022, pp. 172–177.

[23] A. Marcelli, E. Sanchez, G. Squillerò, 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.*, 2018, pp. 1–6.

[24] A. Marcelli, E. Sanchez, L. Sasselli, and G. Squillero, "On the mitigation of hardware trojan attacks in embedded processors by exploiting a hardware-based obfuscator," in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, 2018, pp. 31–37.

[25] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

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

[27] 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 *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.