On the optimization of Software Obfuscation against Hardware Trojans in Microprocessors

Luca Cassano, Elia Lazzeri, Nikita Litovchenko, Giorgio Di Natale

On the optimization of Software Obfuscation against Hardware Trojans in Microprocessors

Luca Cassano, Elia Lazzeri, Nikita Litovchenko
Politecnico di Milano
Milano, Italy
{first_name.last_name}@polimi.it

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

***Abstract*—The quest of low production cost and short time-to-market, as well as the complexity of modern integrated circuits pushed towards a globalization of the supply chain of silicon devices. Such production paradigm raised a number of security threats among which Hardware Trojan Horses (HTHs), that became a serious issue not only for academy but also for industry in the very last years. Indeed, it has been demonstrated that HTHs can be inserted into microprocessors allowing the attacker to run malicious software, to acquire root privileges or to steal secret information. In this paper we present the use of software obfuscation to protect systems against HTHs that aim at stealing information from the microprocessor while it is executing a program. Moreover, we present a Genetic Algorithm-based approach to optimize such anti-HTH methodology by maximizing the obtained obfuscation while minimizing the introduced overhead. We proved the effectiveness and efficiency of the proposed methodology on the Ariane 64bit RISC-V microprocessor running a set of MiBench benchmarks and cryptographic programs.**

***Index Terms*—Genetic Algorithm, Hardware Security, Hardware Trojan Horses, Microprocessors, Software Obfuscation**

I. Introduction and related Work

The complexity of modern integrated circuits (ICs) and the requirements related to low production cost and short time-to-market, pushed the design and fabrication of ICs towards a globalized supply chain [1]. This new paradigm brought a dramatic reduction of design cost and time, but, on the other hand, it came at the cost of a significant loss of trust in the final system [2]. Several security threats raised in the last years, among which Hardware Trojan Horses (HTHs) [3].

From a very high-level point of view, a HTH is a hard-to-detect malicious modification of a design meant to stay silent most of the time and to activate in a specific and rare condition [3]. The goal of HWHs is to alter or stop the nominal behavior of the system or to steal secret information. HTHs may be inserted in any stage of the design process and at any level of abstraction: untrusted vendors may sell infected IP cores [4], rogue employees and untrusted CAD tools may alter the design [5] or untrusted mask providers and silicon foundries may maliciously modify the layout [6].

HTHs have been traditionally considered as a purely academic issue because of their reduced complexity and limited dangerousness. In the last years, a new menace raised: the *software exploitable HTHs* [7]. Complex and highly dangerous HTHs may be implanted in real-world microprocessors allowing the attacker to execute a malicious software, to modify the running software or to steal secret information [8], or even to acquire root privileges on the host system [9]. Moreover, in 2018, security researchers found a HTH, the so-called *Rosenbridge backdoor*, in a commercial Via Technologies C3 processor [10]. This HTH could be activated and exploited to enter in supervisor mode by simply executing a predefined instructions sequence. The feasibility of implanting and activating software exploitable HTHs in commercial microprocessors makes such attacks not only a concern for academy but also a serious threat for industry.

A large number of methodologies for detecting HTHs has been proposed in the last two decades [11]. Most of these methodologies attempt to detect the presence of HTHs in the system before deployment, by exploiting logic testing, formal property verification, side-channel analysis, optical inspection and proof-carrying hardware. All these techniques suffer from a number of limitations, e.g., the difficulty of triggering the HTHs at design time, the need for a golden reference of the circuit under analysis, the ability of detecting only a specific classes of HTHs.

The need for building trusted systems from untrusted components and for providing trusted execution over untrusted systems pushed the definition of HTHs tolerance techniques, moving towards the new *Design-for-Trust* paradigm [12]. Existing HTH-ralated Design-for-Trust approaches are based on the integration of redundant functionally equivalent IP cores belonging to different IP vendors, like in [13], or on the deployment of ad-hoc checkers working in parallel with the core under protection, like in [14], [15]. Moreover, security-aware task scheduling has been proposed [16]. Issues related to the adoption of these methodologies are related to the fact that they can be applied only when 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 change functionality HTHs, while they are ineffective against information stealing ones.

In this paper we introduce the use of software obfuscation for mitigating the dangerousness of information-stealing HTHs in microprocessors. Moreover, we present a genetic algorithm-based approach for the optimization of the software obfuscation procedure such that we increase the confusion in the program (without altering its nominal functionality) while reducing the introduced overhead. The advantage of such approach is that no modification of the underlying HW platform is required. We exploit software

978-1-6654-9431-1/22/$31.00

obfuscation to minimize the probability of exposing sensitive information to the HTH. Indeed, starting from the original software, we increase the usage of the microprocessor's registers, by adding garbage instructions, by periodically scrambling the variables among registers and by substituting constant values with instructions sequences.

With respect to the existing anti-Trojan Design-for-Trust techniques, our methodology is purely software-based; therefore, it can be applied both when the system is still to be designed as well as on already designed and deployed systems. Moreover, our methodology does not require any redundancy or modification to target microprocessor. The only similar idea has been proposed in [17] where software diversity is achieved by substituting program instructions with equivalent ones during the fetching procedure. Nevertheless, the proposal in [17] only considers sequentially-triggered change-functionality HTHs and equivalent instructions substitution is solely employed. The main contributions of this paper are:

- the definition of software obfuscation procedures to protect the execution of a program from information stealing HTHs infesting the underlying microprocessor,
- the definition of a Genetic Algorithm-based engine to optimize the software obfuscation procedure, and
- an experimental campaign aimed at assessing the feasibility of software obfuscation for security purposes and its optimization on a set if MiBench benchmarks [18] and other criptographic programs executed on the 64bit Ariane RISC-V microprocessor [19].

The remainder of this paper is organized as follows: Section II discusses the considered threat model and some background on design obfuscation; Section III presents the proposed security-aware software obfuscation procedure while Section IV presents the companion genetic algorithm-based optimization engine; Section V reports from an experimental campaign while Section VI discusses the security analysis; Section VII concludes the paper.

II. Background

A. The Considered Threat Model

Withe respect to the classical HTHs classification [3], we take into account both *triggered* and *always-on* HTHs that aim at stealing information from the infected microprocessor. Moreover, we consider HTHs infesting microprocessor's logic, inserted during any phase of the design process and at any level of abstraction.

We assume a two-level information stealing attack that is carried out by means of the inserted HTH: first, the HTH repeatedly exfiltrates the content of a number of registers of the microprocessor and it covertly sends it to the attacker. The attacker collects this data to then post-process it to retrieve sensitive information. We further assume that, when injecting the HTH at design- or fabrication-time, the attacker knows all the details of the hardware platform. Moreover, we assume that the attacker has an idea about which operating system and programs will be executed on the attacked microprocessor but, on the other hand, he/she cannot have all the details about software versions and implementations.

In order to make our threat model dangerous and realistic at the same time, we assume that the HTH is able to monitor and exfiltrate data from a reduced number of registers of the infected microprocessor. We believe that this assumption is totally reasonable if we keep in mind that: i) HTHs need to be small enough not to be detected via optical inspection, ii) HTHs need to have a very small impact on power consumption, electromagnetic emission and timing of the infected system, and iii) HTHs cannot occupy the transmission channels for a long time in order not to be discovered. Therefore, we assume that the HTH monitors (at runtime) the content of a fixed (at the attack design-time) and small set of registers and exifiltrates data through a (possibly large) number of clock cycles. Given the above discussed limitations, we assume that the HTH is not able to change the monitored registers, e.g., in a round-robin fashion. Finally, based on these limitations, we also assume that the attacker knows all the details of the deployed countermeasures but this does not bring any advantage.

The proposed software obfuscation methodology does not consider change-functionality and denial-of-service HTHs.

B. Design Obfuscation

Obfuscation has been largely employed both for hardware [20] and software protection [21]. Generally speaking, the goal of obfuscation is to protect the intellectual property associated with a program or a circuit from unauthorized use or reproduction. The goal of hardware obfuscation is to avoid i) reverse engineering of the circuit's netlist by observing the circuit's layout and of the circuit's functionality by observing the circuit's netlist and ii) overproduction of unauthorized chips to be sold in the black market. This is achieved through the use of non-standard cells (the so-called *camouflaging*) or by "locking" the netlist in order to make the fabricated circuit unusable before unlocking it through a secret key (the so-called *logic locking*).

Software obfuscation aims at making hard, e.g., for a decompilation tool, to retrieve the functionality implemented by a program, the meaning of a given construct or variable, the value of constants, the structure of classes and arrays by observing the object code. As for hardware obfuscation, the goal of obfuscating the software is to avoid intellectual property break. This is achieved by inserting never-executed dummy code, by reordering or hiding instructions, by unrolling and extending loops, by opacifying logic conditions and by splitting and merging arrays and data structures.

III. Security-aware Software Obfuscation

We propose to exploit software obfuscation at the assembly-level to mitigate the dangerousness of information-stealing HTHs in microprocessors by reducing the amount of significant information exposed to HTHs. More in details, starting from the original version of a program, we produce a functionally equivalent obfuscated version that is the one that would actually be deployed in the final system. When looking at the (possibly infected) microprocessor architecture executing the program, software obfuscation aims at: i) spreading sensitive information through microprocessor's registers and submerging it among garbage data, and ii) periodically

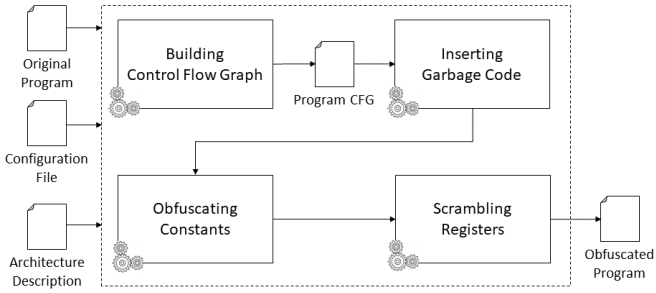


Figure 1: The software obfuscation framework

scrambling sensitive information among microprocessor's registers. 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, we can argue that the presented software obfuscation strategy achieves the following three benefits: i) it minimizes the amount of sensitive information exposed to the attacker, ii) it maximizes the amount of garbage data exposed to the attacker, and iii) it minimizes the time for which sensitive information is kept in the same register. The proposed software obfuscation relies on: i) **garbage code insertion**, ii) **constants obfuscation**, and iii) **register scrambling**.

Our methodology is depicted in Figure 1: it takes the plain assembly code of the program to be protected and it produces the assembly code of the obfuscated version. The input program and the corresponding output program are functionally identical, i.e., the two programs produce identical outputs when fed with identical inputs. Moreover, the methodology takes a description of the architecture of the target microprocessor (in terms of instruction set, instructions' format and operands and registers' names and sizes) and a configuration file to drive the obfuscation process. The first step of the proposed methodology is parsing the input assembly code to build an internal Control-Flow Graph (CFG) representation. Then, the software obfuscation techniques are applied in the following order: i) insertion of garbage code, ii) obfuscation of the constants, and iii) scrambling of registers.

A. Garbage code insertion

As a first step we perform garbage code insertion. This activity consists in randomly selecting the block of the program where to insert the garbage code, the line of the block after which inserting and the length of the garbage code sequence to be inserted. The garbage instructions to be inserted are randomly selected among the move, shift, arithmetic and logic ones. Moreover, also the operands of the inserted garbage instructions are randomly generated. Therefore, in order 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. The registers from which the inserted garbage instructions read and in which they write are chosen among the registers that are unused for most time in the block in which the garbage code is inserted. In this way, we maximize the usage of all registers as well as we minimize the time between two consecutive modifications of a register's content. Moreover, by inserting garbage code we also break specific instructions patterns whose identification during program execution could be of interest for the attacker.

Table I: The parameters of the SW obfuscation 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

B. Constants obfuscation

The second software manipulation is the obfuscation of the constant values in the program. This activity consists in randomly selecting a block of the program and a code line where an immediate value, i.e., a constant, is used. Let call this identified instruction the *target instruction*, the register where the constant value is stored the *target register* and the constant value itself the *target value*. The goal of the constant obfuscation activity is to substitute the target instruction with a randomly long sequence of instructions (dubbed the *obfuscation sequence*) such that, at the end of its execution, it leaves the target value in the target register. The obfuscation sequence is composed of load, move, logical and arithmetic instructions and it employs only the registers that are unused in the randomly chosen code block.

C. Register scrambling

The last software manipulation activity is the scrambling of the registers in the input program. This activity consists in selecting a program block and a target register r_i actually used in the selected block. The *validity block* of r_i, i.e., all the code lines of the selected block where r_i is employed is the identified and a scrambling point, i.e., the specific instruction in the validity block after which introducing register scrambling, is randomly selected. Finally, the register r_j that has not been used for most time in the selected block is identified (let refer to this register as the *scrambled register*), the scrambling instruction

`mv` r_j`,` 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.

D. Software obfuscation parameters

The proposed software obfuscation methodology can be configured through the set of parameters reported in Table I. The larger N_{gi}, L_{gi}, N_{co} and L_{co}, the more obfuscated the obtained program. On the other hand, these parameters (except for N_{rs}) highly affect the introduced overhead, i.e., the number of additional instructions in the output program.

E. Evaluating Security-aware Software Obfuscation

Based on the previously discussed design guidelines for security-aware software obfuscation, we defined the following effectiveness and efficiency metrics, namely *register heat* and *program enlargement*.

Given a register r, we define the register heat of r, dubbed H_r, as the 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 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. It is straightforward that 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 more obfuscated garbage instructions and register scrambling, the higher the average registers' heat and therefore the harder for an attacker to identify the sensitive information among all the processed data.

Moreover, to assess the introduced overhead, we measure the program enlargement as the percentage increase of the number of assembly lines between the plain and the obfuscated program, dubbed ΔE.

IV. Software Obfuscation Optimization through a Genetic Algorithm-based Engine

Finding the best setting of the configuration parameters of the software obfuscation procedure (reported in Table I) would allow to maximize the effectiveness of the obfuscation itself while minimizing the introduced overhead. This is the purpose of the Genetic Algorithm-based optimization engine described in the following.

A Genetic Algorithm (GA) is a search method based on the analogy with the mechanisms of the biological evolution. GAs require solutions to a problem to be encoded, i.e., represented as a sequence of symbols, that stands for a *chromosome* (a sequence of *genes*) in the biological analogy. A GA starts from an initial set (a *population*) of tentative solutions, ranks them according to a problem-specific *fitness function* and selects the best ones according to a *parental selection function*. The selected chromosomes are then combined (through a *cross-over* operator) and mutated (through a *mutation* operator) to produce a new population. These operations have a degree of randomness, depending on probability distributions whose parameters can be tuned, thus allowing both exploitation and exploration of the solution space. The process is repeated until a termination criterion is met, e.g., the maximum number of populations has been produced or the fitness function of the best solution has not increased for a given number of consecutive populations.

In the following we present the adopted chromosome encoding, fitness function, parental selection function and cross-over and mutation operators.

A. Chromosome Encoding

Since the goal of the designed GA is the identification of the *best* configuration parameters for the security-aware software obfuscation procedure, the defined chromosome consists of five genes (one for each of the parameters reported in Table I). The genes have integer values and they are randomly initialized when the first population is generated.

B. Fitness Function

Since the *best* solution identified by the GA has to maximize the average register heat (H_r) while minimizing the introduced overhead (ΔE), we defined two distinct values that are calculated based on the execution the obfuscated programs. The first value, dubbed *heat*, measures how hot all the registers in the microprocessor are kept during the program execution. The second value, dubbed *enlargement*, measures how longer is the obtained obfuscated program w.r.t. the original one. The value range of both heat and enlargement is $[-10000, 10000]$; the fitness of each solution is calculated as the sum of these two values, thus ranging between -20000 up to 20000.

In order to calculate the value of heat, the execution time of the program is divided into *windows* of c instruction cycles each (being c the number of registers in the microprocessor). In each time window tw_i, the value h_i is calculated as follows: h_i is 10000 if the content of all registers in the microprocessor changed during tw_i; it is -10000 if the content of none of the registers in the microprocessor changed during tw_i and it is linearly scaled between 10000 and -10000, otherwise. The final value of heat is calculated as the average of all the previously calculated h_i values. Similarly, the value of enlargement is 10000 if the length of the obfuscated program is the same as the original program; it is -10000 if the length of the obfuscated program doubles the length of the original program and it is linearly scaled between 10000 and -10000, otherwise.

C. Parental Selection Function

After every population has been generated and the obtained configurations have been processed by the software obfuscation procedure, the chromosomes in the population are ranked based on their fitness. We then select the first 10% of the top-ranked individuals and we directly place them in the next population. We then select the first 80% of the top-ranked individuals and we apply the cross-over operator to get the 80% individuals of the next population. Eventually, we take the 10% worst-ranked individuals of current population and we put them in the next population for exploration purposes.

D. Cross-over Operator

In genetic algorithms, the cross-over operator is meant to provide exploitation of the solution space. We implemented a single cut-point cross-over: given two parent chromosomes, pc_1 and pc_2, we randomly select a cut point and we apply it to both chromosomes, thus obtaining four chromosome sections, pc_1^{head}, pc_1^{tail}, pc_2^{head} and pc_2^{tail}, where pc_1^{head} has the same length as pc_2^{head}, as well as pc_1^{tail} has the same length as pc_2^{tail}. The two descendent chromosomes, dc_1 and dc_2, are then obtained by exchanging the two tail parts: dc_1 will be composed as $< pc_1^{head}, pc_2^{tail} >$ as well as dc_2 will be composed as $< pc_2^{head}, pc_1^{tail} >$.

E. Mutation Operator

The mutation operator is meant to provide exploration of the solution space. We implemented a mutation operator consisting in a single bitflip in the value of a gene. In order to provide solution space exploration without compromising the exploitation provided by the cross-over operator, the probability of applying mutation to a chromosome grows with the drop of positions in the chromosome ranking. Finally, it is worth mentioning that, since mutation is applied to every gene individually, it is possible to have multiple mutations in a single chromosome.

Table II: The considered benchmark programs

Program	#lines (Plain)	#lines (Protected)	Overhead
SHA	214	282	31%
RSA	624	705	13%
CRC	552	610	11%
Idea	1706	1965	15%
MatrixMul	483	552	14%
Patricia	870	1137	31%

V. Experimental analysis

We implemented the proposed software obfuscation methodology and the companion Genetic Algorithm-based optimization engine as a set of automatic Python scripts and C programs. We targeted the 64bit Ariane RISC-V [19] microprocessor that counts 32 user registers, its ISA and toolchain and we considered the set of benchmark programs reported in Table II, where the program names and corresponding number of assembly lines of the plain, unprotected version of the program are reported in the first two colums.

As a first validation note, we checked that the obfuscated programs were always functionally equivalent to the corresponding plain ones through a set of random simulations.

When considering the effectiveness of proposed optimized software obfuscation procedure, if we look at Figure 2 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 about 281%. 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 99% for SHA and 375% for Idea.

When deepening the analysis for a specific program and when looking 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 used. Figure 3 reports this analysis for CRC: it is evident that only 6 registers are employed in the plain version, while this number grows up to 29 in the obfuscated one.

More in depth, Figure 4 reports the heatmap representing the cycle-per-cycle heat of all registers during the executions of the plain and obfuscated CRC. Again, it is evident how our proposal allows to maximize registers' usage.

Finally, the last two columns of Table II report the number of assembly instructions in the obfuscated program and the introduced program enlargement. The average overhead is about 19%, with a maximum overhead of 31% for SHA and Patricia. We believe that this overhead is totally reasonable if we consider that the proposed software obfuscation methodology would make information stealing harder (as it will be also discussed in the subsequent security analysis).

As a final remark related to the efficiency of the proposed GA, Figure 5 reports the value of the fitness function for the best chromosome when applying the software obfuscation methodology and the companion search engine to CRC for a number of subsequent populations. It can be observed how few generations are required to get to the final solution.

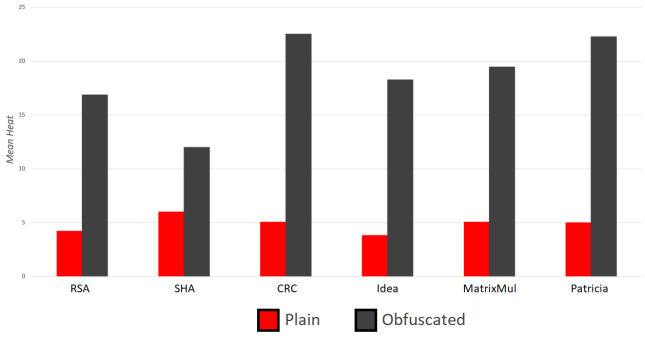


Figure 2: Aggregated average H_r values

VI. Security analysis

The proposed software obfuscation methodology and companion optimization engine are 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. In order to effectively carry out an information stealing attack, 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 would of course require to implant a larger HTH which would send much more data to the attacker, thus making the HTH itself either harder to be implanted or easier to be detected. Moreover, the identification of the sensitive information among all the received data would be much more difficult for the attacker. Therefore, we believe that our proposal would make information stealing attack through implanted HTHs much harder.

VII. Conclusions

We presented an automatic methodology for software obfuscation aimed at protecting program execution over possibly untrusted microprocessor-based systems against information stealing HTHs. Moreover, we presented a Genetic Algorithm-based engine for the identification of the best parameters for the software obfuscation procedure, thus obtaining an optimal obfuscation both in terms of effectiveness and introduced overhead. We proved the correctness, effectiveness and efficiency of our proposal by applying it to the Ariane 64bit RISC-V microprocessor running a set of MiBench and cryptographic programs.

References

[1] DIGITIMES, "Trends in the global ic design service market." http://www.digitimes.com/news/a20120313RS400.html?chid=2.

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

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

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

[5] 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)*, pp. 545–548, 2017.

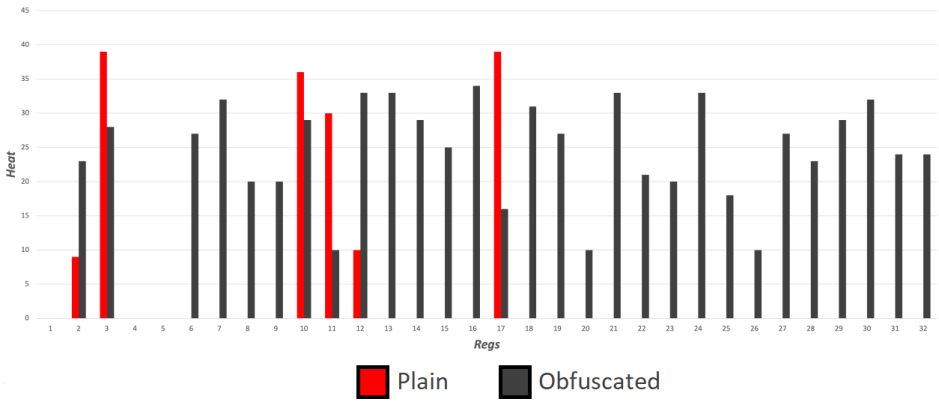
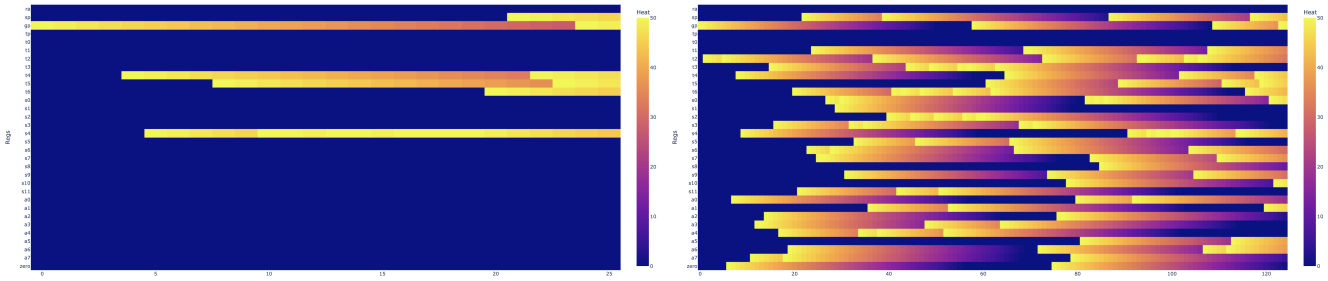


Figure 3: Register-per-register average H_r for CRC



(a) Plain (b) Obfuscated

Figure 4: In depth cycle-per-cycle register-per-register analysis of the application of the proposed methodology to CRC

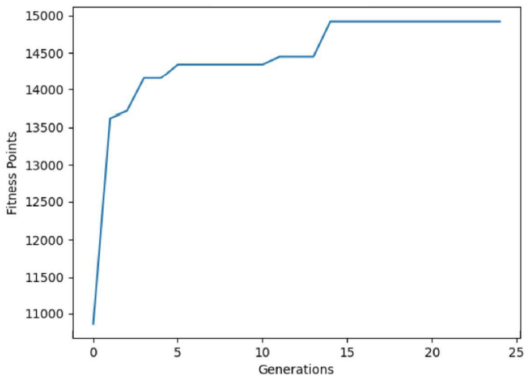


Figure 5: Fitness function of the best chromosome for CRC

[6] 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*, pp. 197–214, Springer, 2013.

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

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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, Jan. 2020.

[17] 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.*, pp. 1–6, 2018.

[18] 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)*, pp. 3–14, 2001.

[19] F. Zaruba and 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*, vol. 27, no. 11, pp. 2629–2640, 2019.

[20] 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.

[21] 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.