TAD: Time Side-Channel Attack Defense of Obfuscated Source Code

Alexander Fell
Nanyang Technological University
Singapore
afell@ntu.edu.sg

Hung Thinh Pham
Nanyang Technological University
Singapore
pham_ht@ntu.edu.sg

Siew-Kei Lam
Nanyang Technological University
Singapore
assklam@ntu.edu.sg

ABSTRACT

Program obfuscation is widely used to protect commercial software against reverse-engineering. However, an adversary can still download, disassemble and analyze binaries of the obfuscated code executed on an embedded System-on-Chip (SoC), and by correlating execution times to input values, extract secret information from the program. In this paper, we show (1) the impact of widely-used obfuscation methods on timing leakage, and (2) that well-known software countermeasures to reduce timing leakage of programs, are not always effective for low-noise environments found in embedded systems. We propose two methods for mitigating timing leakage in obfuscated codes. The first is a compiler driven method, called TAD, which removes conditional branches with distinguishable execution times for an input program. In the second method (TADCI), TAD is combined with dynamic hardware diversity by replacing primitive instructions with Custom Instructions (CIs) that exhibit non-deterministic execution times at runtime. Experimental results on the RISC-V platform show that the information leakage is reduced by 92% and 82% when TADCI is applied to the original and obfuscated source code, respectively.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; • **Software and its engineering** → *Source code generation*; • **Computer systems organization** → Embedded systems;

KEYWORDS

Software obfuscation, hardware diversification, time side-channel, reverse-engineering.

ACM Reference Format:
Alexander Fell, Hung Thinh Pham, and Siew-Kei Lam. 2019. TAD: Time Side-Channel Attack Defense of Obfuscated Source Code. In *Proceedings of ACM Asia South Pacific Design Automation Conference (ASP-DAC 2019)*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Advancements in power management, computational capacities, and the availability of affordable System-on-Chips (SoC) resulted in the omnipresence of embedded systems. These systems, which are often interconnected and accessible through the Internet, perform a specific task over a long period but lack the mechanism to include new functionalities or security updates. Even if an update mechanism is available, manufacturers often neglect obsolete devices due to economic reasons. This presents opportunities for adversaries, who have access to these devices, to extract secret and private information, or to launch Distributed Denial of Service (DDoS) attacks on large Internet services [4, 10, 28].

To increase the costs of obtaining information from a program, obfuscation hides internal implementation details and has been successfully used to delay reverse-engineering attacks (refer to Section 2.1). Another form of attack, called the side-channel attack, can extract secret information from a program on embedded systems [2]. During side-channel attacks, the adversary correlates emerging patterns in power consumption, temperature variations, radiation emissions, periphery access patterns, or execution times to obtain hidden information.

This work considers the problem of time side-channels, wherein the observable execution time of a victim program depends on the secret information and its inputs. While time side-channel attacks and obfuscation techniques have been extensively, yet separately, discussed in literature (refer to Section 2), this work investigates the impact of source code obfuscation techniques on the information leakage that can be exploited in time side-channel attacks.

In this paper an extension to the LLVM compiler framework [27] is introduced, which obfuscates the software not only to harden it against reverse-engineering but also to counteract the efflux of information during those attacks. The extension is included in a diversification framework that compiles security sensitive programs for a RISC-V SoC composed of a Rocket core, a custom defined Rocket Chip Co-processor (RoCC), instruction and data caches, and a Floating Point Unit (FPU) [7]. Our framework generates an extended instruction set that includes hardware support for timing diversity in the form of Custom Instructions (CI) with variable execution times (refer to Section 3). To obtain the results in Section 4, the RoCC is further synthesized for the Zedboard featuring a Xilinx Zynq-7000 FPGA [8], and the hardened programs are executed on the bare metal RISC-V processor to emulate low-noise environments commonly found in embedded systems. Finally, Section 5 draws conclusions.

2 RELATED WORK

Obfuscation techniques and time side-channel attacks, have been extensively studied in literature for many years. This section summarizes the state-of-the-art in both domains.

ASP-DAC 2019, Jan. 2019, Tokyo, Japan
© 2019 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

Algorithm 1 Example Listing

```
function modExp($y, k$)
    $r \leftarrow 1$
    for all $\langle k_i | i \in k \rangle$ do
        if $k_i = 1$ then $r \leftarrow (r \times y) \mod n$ end if
        $y \leftarrow y^2 \mod n$
    return $r \mod n$
```

2.1 Obfuscation

Due to the high accessibility of embedded systems and their software uniformity, an attacker can easily obtain such a system, and extract, disassemble, analyze and alter the binary program code. To prevent such a breach, the software is obfuscated [3, 18, 20, 23, 24]. An obfuscation function $O(P) = P'$ converts a program P such that $P(x) = P'(x)$ for all inputs of x. Ideally, there is no relation between P' to its origin except what can be inferred by observing its inputs and outputs. Since no cryptographic guarantee of security is given [5, 22] and its effectiveness is therefore discussed controversially [38], the aforementioned methods cannot prevent a reverse-engineering attack entirely. As a result, they only increase the costs to analyze and understand the program through static and dynamic analysis [37].

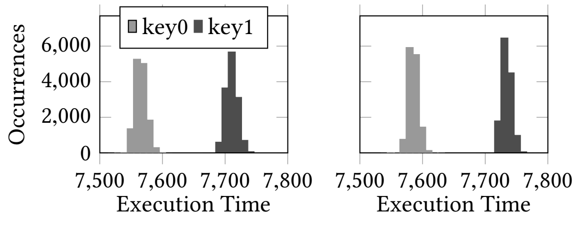
2.2 Side-Channel Attacks

Side-channel attacks aim to extract secret information from an SoC like keys for encryption algorithms. By monitoring the behavior of a victim program such as the input/output sequences caused by memory and periphery access [12, 36], cache performance [1, 13, 21], power consumption, execution time [14, 15, 19], or by combining multiple attack vectors [29], the attacker is able to correlate emerging patterns to the actual value of the key.

To prevent time side-channel attacks, the execution time for program $P(x)$ needs to be constant and independent of its input operands x. Varying instructions of conditional branches, cache hits and misses, branch predictions, and variable-latency arithmetic instructions cause distinguishable execution times. In the example given in Listing 1, it can be observed that the execution time depends on the number of set bits in key k.

Cross-copying (CC) [33] is an existing countermeasure against time side-channel attacks. The method aims to equalize the execution time of conditional branches through program transformation. For example, an else-branch is inserted into the example in Listing 1, which ideally should consist of the same instruction sequence. However, to ensure correctness, the added instructions must not exhibit any side effects and hence cannot be the same resulting in distinguishable execution times especially in low-noise environments as shown in Figure 1a.

Another method to prevent time side-channel attacks is called conditional assignment (CA) [31], which embeds the predicate of a condition in a mask. The security sensitive program sequence is modified such that the values (r in Listing 1) are always calculated. Before the changes are committed to memory, the mask is applied to choose the correct value. This method removes the entire condition and therefore the divergence of different execution paths in the Control Flow Graph (CFG).



(a) Cross-copying **(b) Conditional Assignment**

Figure 1: Distinguishable execution times for the example given in Listing 1 after applying CC and CA.

While both methods reduce the information leakage by normalizing the execution time across the CFG, they do not prevent information leakage due to instructions, whose latencies depend on the operand values. Such instructions may result in distinguishable execution times especially in low-noise environments such as SoCs without an OS and its scheduling mechanisms (refer to Figure 1). Therefore, any program that is hardened against time side-channel attacks, cannot have several execution paths in the CFG and must avoid any instruction, which leaks secret information.

Similar to our work, a processor architecture and compiler, called Øzone, reduce information leakage in [9]. Instead of a mask, a conditional move instruction (CMOV) of the x86 architecture, is used with the condition as a predicate. To further reduce the time side-channel leakage, complex hardware components such as uncached scratchpad memories, pipeline flushes and hardware schedulers, are employed. In our work, Custom Instructions (CIs) are utilized to reduce the leakage effects without incurring major hardware modifications of the base processor. The main contributions of this paper are:

- We show that obfuscated codes exhibit timing leakage and widely used software countermeasures on the obfuscated codes are not always effective for low-noise environments often found in embedded systems [32].
- An LLVM pass, called Time Side-Channel Attack Defense (TAD), automatically creates conditional assignments in the obfuscated codes during the compilation process to harden security-critical functions against timing attacks.
- TADCI as an extension to TAD, addresses the problem of value dependent instruction execution times by automatically replacing these instructions with Custom Instructions (CI) that exhibit a pseudo-random execution time, which further reduces the information leakage.

3 IMPLEMENTATION

The proposed method to harden an obfuscated input program P' against time side-channel attacks automatically, is summarized in Listing 2. The source code of the program, which comprises security critical functions, is compiled into the LLVM intermediate representation (IR) format [26, 27] and has already been obfuscated by the methods described in [23] to increase the costs of reverse-engineering. In the first step, a Control Flow Graph (CFG) is created from P' consisting of Basic Blocks (BB) and their interdependencies. Since only conditional BBs result in multiple graph traversal possibilities, they are identified and marked in graph G_{if}.

Algorithm 2 Timing Attack Defense (TAD) LLVM Pass

```
 1: function TimingAttackDefensePass($P'$)
 2:     $G_{if} \leftarrow$ RecursiveSearchIf(CFG($P'$))
 3:     for all $if_{BB} \in G_{if}$ do
 4:         $m \leftarrow$ ComputeStoreMask($if_{cond}$)
 5:         SetTerminator($if_{BB}, ifTrue$)
 6:         AdaptBranch($m, if_{True}$)
 7:         AdaptBranch($\neg m, if_{False}$)
 8:         SetTerminator($if_{True}, if_{False}$)
 9:     return InsertCI($P'$)
10: function AdaptBranch($mask, branch$)
11:     for all $inst \in$ GetBB($branch$) do
12:         if $s =$ StoreInst($inst$) then
13:             $o \leftarrow$ LoadOriginalValue($s_{address}$)
14:             Replace($s, (mask \wedge s_{value}) \vee (\neg mask \wedge o)$)
```

For each BB in G_{if}, the condition is converted into a mask computation such that all bits are set, if the condition is true, and all bits equal 0 otherwise. This value is stored (line 4), followed by replacing the conditional branch with an unconditional jump instruction into the first BB of the if-branch if_{True}. Therefore, all instructions in this branch always execute independently of the result of the condition (refer to Figure 2b).

However, to ensure correctness, any store instruction found in the BBs of the if-branches, needs to check the mask first, before the computed value is committed to the memory. For instance, in the case where the condition is false, all the computations of if_{True} need to be discarded (refer to line 14).

If an else-branch exists, the inverse mask is applied to all store instructions followed by the insertion of an unconditional branch instruction from the last BB of if_{True} to the first BB in the else-branch if_{False}. Therefore, if the algorithm is applied to the example in Listing 1, all traversal options are removed. Hence, a constant program execution time is ensured.

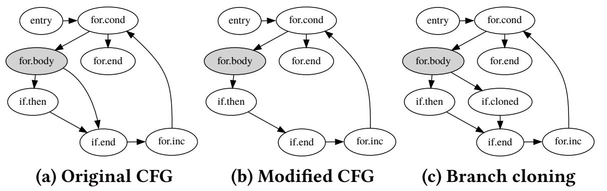


Figure 2: Possible branch modifications to ensure constant execution times for Listing 1.

In the final step, all instructions, whose execution times depend on the operand value, are replaced by Custom Instructions (CI) in line 9. The CIs consist of 2 parts:

(1) A software interface extending the instruction set of the RISC-V architecture, which comprises opcodes dedicated to the RoCC [7]. Further it carries configuration information about the CI to be executed, which registers to use for loading and storing operands, etc.
(2) A hardware diversification representing the intended functionality. It is implemented in a Hardware Description Language (HDL) and uploaded into the RoCC. The RoCC is a decoupled extension to the execution stage of the ALU and can be configured with an arbitrary functionality without major modifications to the processor. While the RoCC performs the computation, the ALU stalls until the instruction completes.

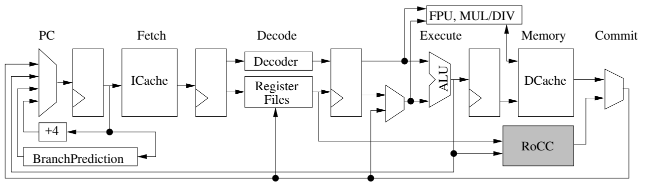


Figure 3: Overview of the RISC-V architecture with RoCC

Thus, a tight relationship between the software interface and the RoCC implementation exists and the developer must be aware of the supported CIs for a particular RISC-V implementation. Their equivalent functionalities are implemented in Verilog HDL and synthesized for the RoCC. In addition, a pseudo-random generator stalls the ALU for a non-deterministic amount of clock cycles, when a CI is executed [34]. For the examples given in section 4, CIs replace the multiplication and modulo instructions. However they can also consist of a conglomerate of multiple instructions. Keeping the functionalities of CIs secret together with the embedment into obfuscated program code, prevents the attacker from removing the TAD without breaking the program functionality (remove-and-replace attacks). The remaining instructions execute in the base processor as before.

A high-level block diagram representing the layout of the processor is given in Figure 3. Depending on the configuration, the RISC-V core is equipped with dedicated arithmetic units, an RoCC, an FPU, and caches. As it can be observed in the figure, the proposed method requires no modifications to the base processor as only the RoCC implements the hardware diversification.

3.1 Conditional Assignment versus Branch Cloning

Another method to modify a program to prevent timing information leakage is to clone entire branches (refer to Figure 2c). To ensure correctness, the cloned branches must not have any side effects. Therefore, all addresses of store instructions are replaced by newly allocated dummy memory locations. However, at later compilation stages, the compiler may discover that those dummy locations are always written but never read, and the corresponding code is removed. In addition, although the instruction execution requires the same time for each of the cloned BB, it was observed that the load and store instructions were subjected to delays arising from cache misses when the execution path of the CFG took a different branch. As a result, an attacker can perform side-channel attacks by extracting timing information and correlate it to the number of branch switches between consecutive loop iterations.

4 EXPERIMENTAL RESULTS

The threat model includes an attacker, who knows the functionality of the program and has physical access to the embedded system. Through debugging interfaces and port probing, any program stored in the memory either outside or within the SoC, can be extracted, disassembled and analyzed. Although the adversary can

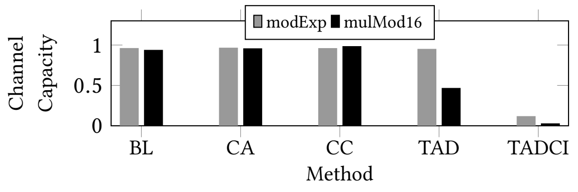


Figure 4: Impact of the proposed algorithm on the channel capacity on the unobfuscated implementation executed in a low-noise environment.

Algorithm 3 mulMod16 Example

```
function MULMOD16($a, b$)
    $a \leftarrow a \wedge 65535$; $b \leftarrow b \wedge 65535$
    if $a = 0$ then $a \leftarrow 65537 - b$
    else
        if $b = 0$ then $a \leftarrow 65537 - a$
        else
            $p \leftarrow a \times b$; $b \leftarrow p \wedge 65535$; $a \leftarrow b - p/2^{16}$
            if $b < a$ then $a \leftarrow a + 1$ end if
    return $a \wedge 65535$
```

measure the execution time accurately, he is unaware of the internal implementation details nor has access to the original source code used to create the binary under investigation. Other leakage channels caused by power consumption, temperature or electromagnetic variations are not considered in this work.

This section discusses and analyzes two representative examples to demonstrate the effectiveness of TAD and TADCI: (1) *modExp* provides the RSA modulo exponent functionality [35] to encrypt and decrypt a message from the benchmark suite introduced in [31] (refer to Listing 1). It consists of a loop with a static iteration count, often found in cryptographic algorithms such as AES, and (2) a modular multiplication from the IDEA cipher [25] implemented in *mulMod16* (Listing 3).

Both implementations are obfuscated using various sequences of program transformations and their obfuscation and leakage levels are computed. Since the obfuscator generates a different binary with every compilation run, the source codes of both examples have been compiled 15 times for each experiment by Clang 7.0.0 without compiler optimizations. The resulting binaries are uploaded to the RISC-V softcore processor with its RoCC, which are hosted on the Xilinx Zynq-7000 FPGA. No operating system boots on that core to emulate an execution environment with low noise levels. 1000 iterations of all compiled binaries are executed for two distinguishable keys each. An integrated performance counter, which increments with every clock cycle spent in the security sensitive section, gives an exact measurement of execution times for this region. LeakiEst [16] iteratively estimating the channel capacity using the Blahut-Arimoto algorithm [6, 11], computes the relationship between the given input and the observed time.

4.1 Time Side-Channel Capacity

During a time side-channel attack, the unintentional leakage of information represents a communication channel that transmits data to the adversary. Like any other communication channel, its capacity can be computed by Shannon's Theorem [39], which is equal to the upper bound of transmission rate through that channel.

Figure 4 shows the information leakage of the original implementation (baseline, BL) of modExp and mulMod16 and their modified versions such as the manual insertion of CA and CC. These modifications have a negligible impact on the channel capacity. The information leakage on the mulMod16 example is primarily caused by a large else-branch that includes a complex multiplication (Listing 3). In addition, despite that the proposed TAD has normalized the execution times of modExp (refer to the histograms in Figures 5a and 5b), it is insufficient to compensate for the jitter caused by variable-latency arithmetic instructions, cache hits and misses and memory access latencies in low-noise environments. Therefore, the conditional branches that are normalized using TAD, do not result in a reduction in channel capacity. When TAD also replaces instructions (TADCI), whose execution time depends on the value of the operands (Figure 5c), the overall leakage is reduced by 88% and 98% for modExp and mulMod16, respectively (see Figure 4).

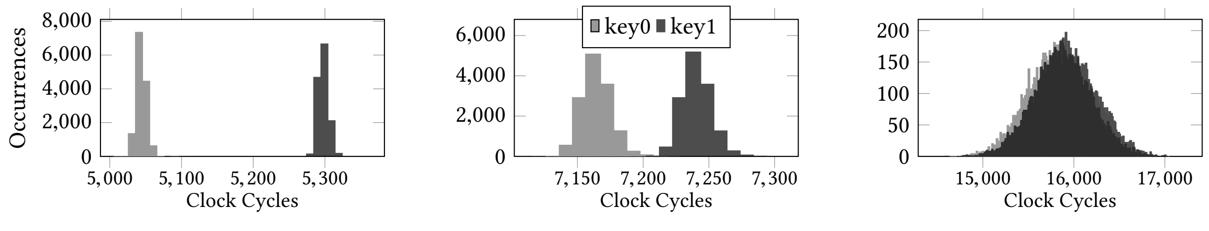
In Figure 6, the information leakage is shown for various obfuscation strategies. Numerical values in the description of the obfuscation strategy represent the iterations the corresponding obfuscation method has been applied to the source code. As can be observed in Figure 6b, if the original source code of mulMod16 is obfuscated, the leakage of information is reduced to 19% in average by normalizing the execution time (TAD) without the need to replace variable-latency arithmetic instructions. TADCI has only a marginal effect on the channel capacity. However, for modExp, it is necessary to replace the variable-latency arithmetic instructions (TADCI) to reduce the leakage sufficiently to 16% in average (Figure 6a). Here, these instructions are nested in conditional branches executed multiple times in a loop, while in the mulMod16 example, the number of invocations of the instructions is limited.

Furthermore, as shown in the figure, incorporating bogus flow control (BFC) in the obfuscation pass has a positive impact on the channel capacity. On average, the leakage reduces by 52% (modExp) and 43% (mulMod16), when compared to the original source code. Where the source code has been obfuscated in several iterations of substitution (Sub3), bogus flow control (BCF2) and flattening (Fla), the information leakage reduces to 30% even without hardening the code using TAD or TADCI. Code obfuscation leads to a significant increase in code size, which reduces the execution time ratio between the sensitive code sections and the remaining ones. The inherent increase in memory accesses and cache latencies overshadows the otherwise observable variations, indirectly reducing the information leakage.

The results show that for low-noise environments, existing countermeasures cannot sufficiently thwart time side-channel attacks. A combination of hardware diversification in the form of CIs and side-channel attack centric code obfuscation is required so that an attacker cannot draw conclusions from emerging execution time patterns.

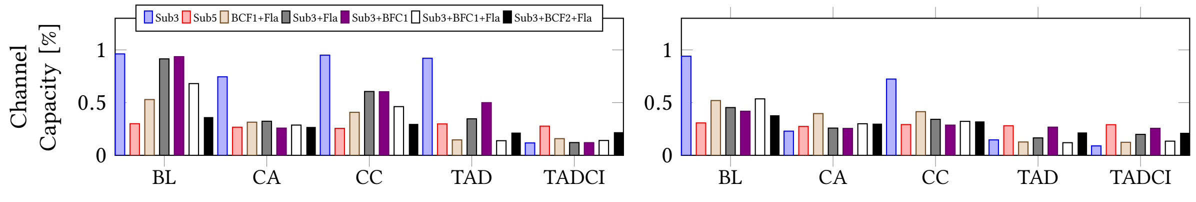
4.2 Normalized Compression Distance

The effectiveness of countermeasures against reverse-engineering is measured by analyzing the complexity of the obfuscated $P' = O(P)$ and calculating its distance from the original program P. In this paper, the Normalized Compression Distance (NCD) [17] which is



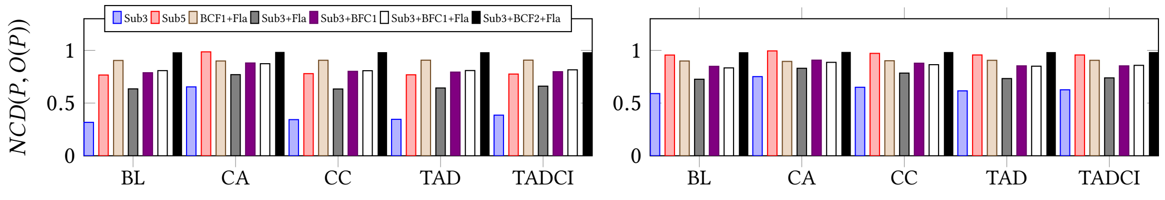
(a) BL (b) TAD (c) TADCI

Figure 5: Histogram of execution times in clock cycles of the original (BL) code of modExp and with TAD/TADCI



(a) modExp (b) mulMod16

Figure 6: Time side-channel information leakage, when various obfuscation techniques (Sub - substitution, BCF - bogus flow control, Fla - flattening) are applied to the baseline (BL) source code.



(a) modExp (b) mulMod16

Figure 7: Similarities between the baseline (BL), CA, CC, and the effect of obfuscation.

approximated by the Kolmogorov complexity [30], is used as the metric for the effectiveness of code obfuscation. An NCD value close to 0 indicates a high level of similarity between $O(P)$ and P. Figure 7 shows that the difference in NCD of CA, CC, TAD and TADCI on the baseline (BL) for the same obfuscations is negligible. In contrast, the application of arithmetic substitutions or the insertion of BFC significantly removes similar structures in the code and hence increases the cost for a reverse-engineer attack.

4.3 Performance and Hardware Utilization

This section discusses the additional penalties in terms of program execution time incurred by applying TAD to the baseline and obfuscated source code. In addition, since TADCI requires a co-processor, its hardware costs are presented.

4.3.1 Performance. Figure 8 shows the relative performance penalties when TAD and TADCI alter the baseline and obfuscated source codes, respectively. If TAD/TADCI is applied to the baselines, the execution times increase significantly. The relative overhead decreases, if the source code is obfuscated, since the negative performance effect of the obfuscations is larger than the one caused by TAD/TADCI.

The impact of TADCI depends on the program and the selected obfuscation sequences. While for Sub3, Sub5, BCF1+Fla, Sub3+Fla, and Sub3+BCF1 the differences in execution times between TAD and TADCI are negligible in the mulMod16 example, they are prominent for other obfuscation methods and in modExp. In modExp, the obfuscated code and the CIs execute in a loop (refer to Listing 1). Hence binaries of modExp modified by TADCI, exhibit an increase of execution time by 36% on average when compared to TAD, while the impact of TADCI for mulMod16 is only 3%.

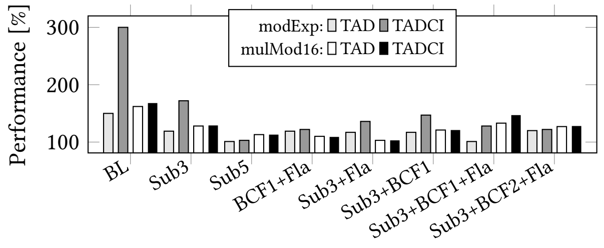


Figure 8: Average impact of TAD and TADCI on the BL source code of modExp and mulMod16 including various obfuscations applied

4.3.2 Resource Utilization. Table 1 shows the increase in Look-Up-Tables (LUT) and DSPs, when the RoCC is added to provide the hardware diversification functionality. The RoCC inclusion to support CIs, increases the LUT requirements of a Xilinx Zynq-7000 FPGA marginally by 2%. Due to the CIs consisting of multipliers,

Table 1: Hardware Resource Utilization

	RISC-V	RISC-V with RoCC	Change
LUTs	32310	32953	+2%
DSPs	15	25	+67%
BRAM	24	24	–

additional DSPs are used. These results confirm that the proposed method can be realized with very small resource overhead.

5 CONCLUSION

Code obfuscation can be vulnerable to time side-channel attacks for embedded systems operating in low-noise environments. We propose a compiler driven method called Time Side-Channel Attack Defense (TAD) to automatically harden the obfuscated code against time side-channel attacks. In addition, we extend TAD to replace arithmetic operations, whose the execution times depend on the operand values, by Custom Instructions (CI) that exhibit non-deterministic latencies. The proposed method, called TADCI, requires no changes to the base processor, since the CI functionalities are encapsulated in the Rocket Chip Co-processor (RoCC). Extensive experiments were conducted on a RISC-V system with the RoCC hosted on a Xilinx Zynq-7000 FPGA. They show that the proposed method reduces the average information leakage by 92% for the original source code and 82% for obfuscated binaries with negligible resource overhead. In particular, only 2% of LUTs and additional 10 DSPs are required to achieve hardware diversification. In addition, when the proposed solution is applied to obfuscated binaries, only an average of 16% and 27% performance overheads are incurred for TAD and TADCI, respectively.

ACKNOWLEDGMENTS

The research described in this paper has been supported by the National Research Foundation, Singapore, under grant number NRF2016NCR-NCR001-006.

REFERENCES

[1] Onur Aciicmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer Verlag LNCS 6225, 110–124.

[2] Jude Angelo Ambrose, Roshan G. Ragel, Darshana Jayasinghe, Tuo Li, and Sri Parameswaran. 2015. Side channel attacks in embedded systems: A tale of hostilities and deterrence. In *International Symposium on Quality Electronic Design*. 452–459. https://doi.org/10.1109/ISQED.2015.7085468

[3] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. 2006. Proteus: Virtualization for Diversified Tamper-resistance. In *Proceedings of the ACM Workshop on Digital Rights Management*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/1179509.1179521

[4] Kishore Angrishi. 2017. Turning Internet of Things (IoT) into Internet of Vulnerabilities (IoV): IoT Botnets. *CoRR* abs/1702.03681 (2017). arXiv:1702.03681 http://arxiv.org/abs/1702.03681

[5] Daniel Apon, Yan Huang, Jonathan Katz, and Alex J. Malozemoff. 2014. Implementing Cryptographic Program Obfuscation. *IACR Cryptology ePrint Archive* 2014 (2014), 779. http://eprint.iacr.org/2014/779

[6] Suguru Arimoto. 1972. An algorithm for computing the capacity of arbitrary discrete memoryless channels. *IEEE Trans. Information Theory* 18, 1 (1972), 14–20.

[7] Krste Asanović, Rimas Avižienis, and Jonathan et al. Bachrach. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley.

[8] Krste Asanović, Rimas Avižienis, and Jonathan et al. Bachrach. 2017. Rocket Chip on Zynq FPGAs. (nov 2017). Retrieved November 2017 from https://github.com/ucb-bar/fpga-zynq

[9] Zelalem Birhanu Aweke and Todd M. Austin. 2017. Øzone: Efficient Execution with Zero Timing Leakage for Modern Microarchitectures. In *HOST*. IEEE Computer Society, 153.

[10] E. Bertino and N. Islam. 2017. Botnets and Internet of Things Security. *Computer* 50, 2 (Feb 2017), 76–79. https://doi.org/10.1109/MC.2017.62

[11] Richard E. Blahut. 1972. Computation of Channel Capacity and Rate-Distortion Functions. *IEEE Transactions on Information Theory* IT-18, 4 (July 1972), 460–473.

[12] Dan Boneh, Henry Corrigan-Gibbs, and Stuart E. Schechter. 2016. Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, Vol. 10031. 220–248.

[13] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *Int. Conf. on the Theory and Application of Cryptology and Information Security*, Vol. 5912. Springer, 667–684.

[14] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *16th European Symposium on Research in Computer Security*, Vol. 6879. Springer, 355–371.

[15] David Brumley and Dan Boneh. 2005. Remote Timing Attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.

[16] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. *A Tool for Estimating Information Leakage.* Springer, CAV 2013, 690–695.

[17] R. Cilibrasi and P. M. B. Vitanyi. 2005. Clustering by compression. *IEEE Transactions on Information Theory* 51, 4 (2005), 1523–1545. https://doi.org/10.1109/TIT.2005.844059

[18] C. S. Collberg and C. Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering* 28, 8 (Aug 2002), 735–746. https://doi.org/10.1109/TSE.2002.1027797

[19] Jean-François Dhem, François Koeune, Philippe-Alexandre Leroux, Patrick Mestré, Jean-Jacques Quisquater, and Jean-Louis Willems. 1998. A Practical Implementation of the Timing Attack. In *CARDIS*, Vol. 1820. Springer, 167–182.

[20] M. Fyrbiak, S. Rokicki, N. Bissantz, R. Tessier, and C. Paar. 2018. Hybrid Obfuscation to Protect Against Disclosure Attacks on Embedded Microprocessors. *IEEE Trans. Comput.* 67, 3 (March 2018), 307–321. https://doi.org/10.1109/TC.2017.2649520

[21] D. Gullasch, E. Bangerter, and S. Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. https://doi.org/10.1109/SP.2011.22

[22] Máté Horváth. 2015. Survey on Cryptographic Obfuscation. *IACR Cryptology ePrint Archive* 2015 (2015), 412. http://eprint.iacr.org/2015/412

[23] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proc. Int. Workshop on Software Protection*. IEEE, 3–9.

[24] Meha Kainth, Lekshmi Krishnan, Chaitra Narayana, Sandesh Gubbi Virupaksha, and Russell Tessier. 2015. Hardware-Assisted Code Obfuscation for FPGA Soft Microprocessors. In *DATE*. ACM, 127–132.

[25] Xuejia Lai. 1992. On the design and security of block ciphers. (1992).

[26] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[27] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Int. Symp. on Code Generation and Optimization*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[28] F. Lau, S. H. Rubin, M. H. Smith, and L. Trajkovic. 2000. Distributed Denial of Service Attacks. In *Int. Conf. on Systems, Man, and Cybernetics (SMC)*, Vol. 3. 2275–2280. https://doi.org/10.1109/ICSMC.2000.886455

[29] Cédric Lauradoux. 2005. Collision attacks on processors with cache and countermeasures. In *Western European Workshop on Research in Cryptology*, Vol. 74. GI, 76–85.

[30] Ming Li and Paul Vitányi. 2008. *An Introduction to Kolmogorov Complexity and Its Applications* (3rd ed.). Springer.

[31] Heiko Mantel and Artem Starostin. 2015. *Transforming Out Timing Leaks, More or Less.* Springer, ESORICS 2015, 447–467.

[32] Michael Melkonian. 2000. Get by Without an RTOS. *Embedded Systems Programming* 13, 10 (2000).

[33] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. *The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks.* Springer, ICISC 2005, 156–168.

[34] Thinh Hung Pham, Alexander Fell, Arnab Kumar Biswas, Siew-Kei Lam, and Nandeesha Veeranna. 2018. CIDPro: Custom Instructions for Dynamic Program Diversification. *Int. Conf. on Field-Programmable Logic and Applications (FPL)* (Aug. 2018).

[35] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM* 21, 2 (Feb. 1978), 120–126.

[36] F. L. Sang, V. Nicomette, and Y. Deswarte. 2011. I/O Attacks in Intel PC-based Architectures and Countermeasures. In *2011 First SysSec Workshop*. 19–26. https://doi.org/10.1109/SysSec.2011.10

[37] Sebastian Schrittwieser and Stefan Katzenbeisser. 2011. Code Obfuscation against Static and Dynamic Reverse Engineering. In *13th International Conference on Information Hiding (IH)*, Vol. 6958. Springer, 270–284.

[38] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *Comput. Surveys* 49, 1, Article 4 (April 2016), 37 pages. https://doi.org/10.1145/2886012

[39] C. E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27, 3 (July 1948), 379–423.