Constant Time Secure Embedded Systems Through Hardware/Software Cooperation

Jean-Loup Hatchikian-Houdot

Constant Time Secure Embedded Systems Through Hardware/Software Cooperation

Jean-Loup Hatchikian-Houdot
Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Side-channel attacks exploit power consumption, execution time, or any other physical effect caused by an implementation. Our work focuses on timing attacks (side-channel attacks exploiting only execution time). There are already several countermeasures to prevent or limit timing attacks, either on the hardware part or on the software part. However, these security mechanisms are working often separately from each other. The software part only knows the functional behavior of the hardware, but with little knowledge of the micro-architecture details, and the hardware does not know the security expected by the software. The goal of our Ph.D. is to establish a contract between the hardware and the software. This contract would allow cooperation between them to have better control regarding timing security and achieve full resistance against timing attacks at a minimal cost.**

***Index Terms*—Side-channel attacks, Constant-time secure, ISA, Compilation.**

I. Introduction

Timing attacks are especially threatening because they can be done remotely [1]. A malicious program running on the target system, even with low privileges, can learn critical secrets of a victim program running on the same system. The attacker does not need physical access to the target system, contrary to most attacks exploiting other side-channels like EM-radiations or power consumption, which require physical proximity with the targeted device.

Our Ph.D. started in October 2021 in the context of the SCRATCHS project funded by the Labex CominLabs. SCRATCHS is a collaboration between researchers in the fields of formal methods (Celtique, Inria Rennes), security (Cidre, CentraleSupélec Rennes), and hardware design (Lab-STICC). The project's goal is to co-design a RISC-V processor and a compiler to ensure by construction that a security-sensitive code is immune to timing side-channel attacks while running at maximal speed. This work only considers mono-core in-order processors typically used in Internet-of-Things objects.

In the rest of this article, we first present a simple timing vulnerability and define the fundamental concepts. Then we describe some existing protections against these vulnerabilities and their limitations. Finally, we explain how we expect to achieve high-level security regarding timing attacks while limiting the performance costs of the protections.

II. An Example of Timing Vulnerability

The function of Figure 1 computes modular exponentiation b^e mod m. Some implementations of the RSA cryptographic algorithm have been relying on this implementation of the modular exponentiation (like GnuPG [2]), with the exponent e being the secret key.

However, the exponent is used in the end condition of a while loop and in a branching condition. This means that the execution time will vary depending on the value of the exponent. An attacker able to precisely measure the execution time or to test which operations are executed could retrieve the value of the exponent, thus breaking these RSA implementations. This kind of attack can be done if the attacker runs one of his programs on the same device as this implementation of RSA [3], [4]. In this case, the value of the exponent is leaking, because it impacts a behavior observable by attackers.

```
function modular_exponentiation(b,e,m){
    ...
    while (e>0)
        //this leaks the secret e
        if (e mod 2) == 1
            result := (result * b) mod m
        e := e >> 1
        ...
    return result
```

Fig. 1. Vulnerable modular exponentiation

III. Leakage and Non-Interference Property

We call leakage any data observable by attackers from side-channels. In the attacker model we consider, all branching choices and memory access indexes are observable through timing side-channel.

The non-interference property states that the leakage of a program must only depend on its public inputs, but not on the secret ones. This implies that attackers cannot deduct any secret from the leakage.

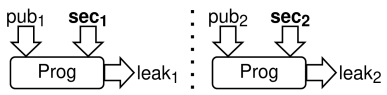

Fig. 2. Two executions of the same program with different inputs

For example, the program of Figure 2 respects the non-interference property if $pub_1 = pub_2 \implies leak_1 = leak_2$. Here, sec_1 and sec_2 are the confidential data. We also assume

that the value of pub_1, pub_2, $leak_1$, $leak_2$, and $Prog$ can be observed by anyone, including attackers.

The vulnerable implementation of Figure 1 does not respect this property because two executions with the same public input might have a different timing leakage, depending on the secret value of the exponent.

IV. Existing Countermeasures

There are currently two main approaches to prevent timing attacks: those implemented in hardware [5], [6], and those implemented in software [7].

A. Timing Attacks Countermeasures in Hardware

Timing attacks often rely on resource sharing, especially stateful resources like caches. In this case, the potential victim first alters the content of a cache when using it. Then, this content can impact the execution time of other programs, potentially controlled by the attacker, making it a vector for timing attacks [8].

Different hardware protections against timing attacks have been proposed, such as cache randomization [5] or cache partitioning [6]. For example, attackers cannot exploit such vulnerabilities if the hardware strongly isolates each task and does not allow them to share any resource.

Unfortunately, such hardware protections have a significant impact on performance. For example, complete partitioning would be too costly since resource sharing is a standard and efficient optimization. A better solution would be to allow sharing for every public data but to make a small requisition of resources when a task is handling a secret (e.g., data used in cryptographic operations).

B. Timing Attacks Countermeasures in Software

On the software side, the state-of-the-art defense against timing attacks is to write programs that are Constant Time Secure (CTS) [7].

Constant time programming achieves the non-interference property. A CTS program never uses a secret value in a branching condition or as a memory index. In practice, when the program would need to branch on a condition depending on a secret, it will instead execute both sides of the branching and then ignore results from the unwanted branch. An attacker observing timing leakage from such a program should not be able to deduce any secrets, as the leakage does not depend on any secret.

Modern compilers like GCC or Clang do not always preserve the CTS of a program during its compilation. Indeed, they implement aggressive optimizations, which could revert the code to a non-CTS state. However, a specialized compiler like Jasmin [9] or a modified version of CompCert [10] preserves the constant time security of a program during its compilation.

Non-interference property is sufficient to prove there is no leakage. However, depending on how leakages are defined, this property can be very restrictive. In the attacker model we use, all branching choices and memory access indexes are in the leakage, i.e., we consider attackers can infer which branch the program takes at every branching point and the value of memory indexes for every memory access.

Unfortunately, this attacker model is weaker than some real attackers. Thus, respecting the non-interference property for our leakage definition is not enough to prevent all timing attacks. The compiled program is executed with micro-architecture instructions defined by the ISA (Instruction Set Architecture). But this ISA only defines the functional behavior of the instructions, not their temporal behaviors. (e.g., some instructions like division take a variable amount of time to execute, which depends on their operands). The micro-architecture used to run software can introduce new timing leakage that is not considered by our leakage model. Thus, real attackers can exploit this vulnerability to deduct secrets, despite constant time security on the software part [11].

C. Hardware / Software Cooperation

Both existing approaches prevent timing attacks, but they both have limitations. Several previous works argue that a contract between hardware and software is required. This contract would restrict the behavior of the hardware such that our current attacker model is sufficient to include every possible timing attack. Thus, software and hardware both respecting this contract could not leak a value unless it is a branching condition or a memory access index).

Heiser et al. [12], [13] demonstrate that current processors are still exposed to timing attacks because of this lack of cooperation, hence the need for this contract to truly achieve constant time security. But they do not propose an implementation of this contract.

Yu et al. [14] propose a Data Oblivious ISA (OISA) which would not only design the intended functional behavior of instructions, but also the security they provide regarding side-channel leakage. This OISA would be sufficient for a software programmer to make secure programs, without the need to care about how this OISA is implemented in the underlying hardware. They provide and implementation of this OISA for an out-of-order RISC-V processor, but this adds restrictions only on the hardware part.

V. Our Contribution

Our Ph.D. will focus on the software part of the solution. We propose to define a secure ISA specification based on the RISC-V instruction set. This contract will define several requirements both on the hardware and software parts.

A. Hardware Requirement: Instruction Operands Safety

The OISA will define which instruction operands are safe regarding timing leakages [14]. A compiler will then be able to use the right instruction depending on the confidentiality of the arguments. For example, consider a division. Depending on the criticality level of the operands, a compiler would emit either a classical division instruction (which leaks the value of its operands) or a leak-free (but potentially slower) division for sensitive input.

The execution time of an operation with safe operands (i.e., operands that do not interfere with the timing behavior of the operation [14]) will probably be the worst-case execution time of its unsafe version. So the compiler will use the unsafe operation whenever this operation is done on public variables.

B. Hardware Requirement: Security Mechanisms for Resources Management

A resource is any component used during the computation. However, our primary concerns are storage components like caches, prefetchers, or buffers. Such stateful components of a computer are common targets of timing attacks. A critical task must have mechanisms to protect its secret data from leaking through timing side-channels. This mechanism could either be via partitioning as previously discussed or via a reset at every context switch.

The compiler will have to choose the appropriate mechanism for each case. The partitioning would have a memory space overhead, and a reset at every context switch would increase execution time.

Other resources will also need protection. Stateful resources are the primary target of known attacks, but stateless resources like transfer or computation resources could also leak data :

The availability of a stateless resource can reflect how and when it is used by other processes [15]. This can leak data about these other processes if their resources usage depends on variables.

We will investigate potential vulnerabilities in these components during this thesis, and security solutions when needed.

These data protection mechanisms must have a constant execution time (their execution time must not depend on the protected data). Otherwise, this would defeat their purpose.

C. Software Requirements

The contract will also dictate prerequisites that the software needs to satisfy to get a security guarantee. A simple example could be on a memory load. Currently, memory loads expose the memory index to side-channel attacks. As a result, a CTS program must never perform a memory load using a secret value as an index. This memory index can leak because a memory load is not performed in constant time. For example, if the data fetched in memory is already present in a cache, the memory load will be faster. A naive solution could be a constant time load with no optimizations, but this would severely reduce the program's efficiency. Our contract could establish a protocol for safe memory loads: the software must signal what load it intends to do in advance and if it wants it to be constant time.

Of course, this software will have to be written in a CTS way and compiled with a specialized compiler. One of the main objectives of our Ph.D. will be to upgrade CompCert so it can compile code complying with this contract. We will also have to prove that the produced version of CompCert works as intended. We are considering using Coq proof assistant for this last part.

VI. Conclusion

Timing attacks, which can be done remotely, are a significant threat to confidentiality. There are already countermeasures for the hardware and software parts, but the lack of cooperation between hardware and software results in costly and imperfect protection.

Our project is to establish a contract between both sides to achieve non-interference property while minimizing the impact on the performances. A program should achieve non-interference, thus be immune to timing attacks if it respects the following requirements. It must be written in a Constant Time Secure way, compiled by a compiler complying with our contract, and executed on a hardware device complying with the same contract.

Our objective is to modify CompCert [10] to implement this contract. The first step would be to specify the safety of operands for each instruction of the ISA and to use this specification in CompCert.

References

[1] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Comput. Networks*, 2005.

[2] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *USENIX Security 14*, 2014.

[3] W. Schindler, "A timing attack against RSA with the chinese remainder theorem," in *CHES* (Ç. K. Koç and C. Paar, eds.), 2000.

[4] P. C. Kocher, "Cryptanalysis of diffie-hellman, rsa, dss, and other systems using timing attacks (extended abstract)," in *Advances in cryptology, CRYPTO '95: 15th Annual International Cryptology Conference*, Springer-Verlag, 1995.

[5] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *42nd IEEE Symposium on Security and Privacy*, 2021.

[6] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," 2005.

[7] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time implementations," in *25th USENIX Security Symposium*, 2016.

[8] C. Su and Q. Zeng, "Survey of CPU cache-based side-channel attacks: Systematic analysis, security models, and countermeasures," *Secur. Commun. Networks*, 2021.

[9] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *CCS* (B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), ACM, 2017.

[10] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proc. ACM Program. Lang.*, 2020.

[11] O. Aciiçmez, J. Seifert, and Ç. K. Koç, "Micro-architectural cryptanalysis," *IEEE Secur. Priv.*, 2007.

[12] Q. Ge, Y. Yarom, and G. Heiser, "No security without time protection: We need a new hardware-software contract," in *APSys*, ACM, 2018.

[13] G. Heiser, "For safety's sake: We need a new hardware-software contract!," *IEEE Des. Test*, 2018.

[14] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *NDSS*, The Internet Society, 2019.

[15] M. Tan, J. Wan, Z. Zhou, and Z. Li, "Invisible probe: Timing attacks with pcie congestion side-channel," in *42nd IEEE Symposium on Security and Privacy*, 2021.