Universidade Estadual de Campinas
Instituto de Computação

Otávio Oliveira Napoli

Timing Side-Channel Analysis of Dynamic Binary Translators

Análise de Canais Laterais de Tempo em Tradutores Dinâmicos de Binários

CAMPINAS
2019

Otávio Oliveira Napoli

Timing Side-Channel Analysis of Dynamic Binary Translators

Análise de Canais Laterais de Tempo em Tradutores Dinâmicos de Binários

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Edson Borin
Co-supervisor/Coorientador: Prof. Dr. Diego de Freitas Aranha

Este exemplar corresponde à versão final da Dissertação defendida por Otávio Oliveira Napoli e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2019

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

N162t

Napoli, Otávio Oliveira, 1994-
Timing side-channel analysis on dynamic binary translators / Otávio Oliveira Napoli. – Campinas, SP : [s.n.], 2019.

Orientador: Edson Borin.
Coorientador: Diego de Freitas Aranha.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Tradução binária dinâmica. 2. Criptografia. I. Borin, Edson, 1979-. II. Aranha, Diego de Freitas, 1982-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Análise de canais laterais de tempo em tradutores dinâmicos de binários
Palavras-chave em inglês:
Dynamic binary translation
Cryptography
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Anderson Faustino da Silva
Julio César López Hernández
Data de defesa: 12-04-2019
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-9606-4751
- Currículo Lattes do autor: http://lattes.cnpq.br/6642011418568107

Universidade Estadual de Campinas
Instituto de Computação

Otávio Oliveira Napoli

Timing Side-Channel Analysis of Dynamic Binary Translators

Análise de Canais Laterais de Tempo em Tradutores Dinâmicos de Binários

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC/UNICAMP
- Prof. Dr. Anderson Faustino da Silva
  DIN/UEM
- Prof. Dr. Julio Cesar López Hernández
  IC/UNICAMP

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 12 de abril de 2019

Acknowledgements

I would like to thank my family, my friends and my professors for the support. I also would like to thank the *Fundação de Amparo à Pesquisa do Estado de São Paulo* (FAPESP) and Intel for their financial support from the project "Secure Execution of Cryptographic Algorithms", process FAPESP#: 2014/50704-7, the *Conselho Nacional de Desenvolvimento Científico e Tecnológico* (CNPq) for the proccess #313012/2017-2, Fapesp for the proccess #2013/08293-7 and the Multidisciplinary High Performance Computing Lab (LMCAD) for its infrastructure and technical support.

Resumo

Ataques de canal lateral são um importante problema para os algoritmos criptográficos. Se o tempo de execução de uma implementação depende de uma informação secreta, um adversário pode recuperar a mesma através da medição de seu tempo. Diferentes abordagens surgiram recentemente para explorar o vazamento de informações em implementações criptográficas e para protegê-las contra esses ataques. Para tanto, a implementação de criptografia em tempo constante é uma pratica amplamente adotada visando descorrelacionar a dependência entre um dado secreto e suas amostras de tempo. Apesar das contra-medidas serem eficazes para garantir execução dos algoritmos em um sistema evitando canais laterais de tempo, emuladores podem modificar e reintroduzir pontos de vazamento durante sua execução. Trabalhos recentes discutem os impactos dos compiladores *Just-In-Time* (JIT) de linguagens de alto nível no vazamento de informações a partir do tempo de execução [1, 2]. Entretanto, pouco foi dito sobre a emulação entre ISAs e seu impacto em vazamentos de tempo.

Neste trabalho, investigamos o impacto de emuladores (tradutores dinâmicos de binários) entre ISAs na propriedade de tempo constante de implementações criptográficas. Utilizando métodos estatísticos e rotinas criptográficas, afirmamos a viabilidade de vazamentos de tempo em códigos gerados por tradutores dinâmicos de binários, mesmo utilizando diferentes técnicas de formação de regiões. Mostramos que a emulação pode ter um impacto significante, inserindo construções de tempo não constante durante sua tradução, levando a vazamentos de tempo significantes. Esses vazamentos podem ser observados em tradutores dinâmicos como o QEMU e o HQEMU durante a emulação de rotinas de bibliotecas criptográficas conhecidas, como a mbedTLS e podem ser rapidamente verificados. Por fim, para garantir a propriedade de tempo constante nós implementamos uma transformação de compilador baseada na transformação *if-conversion* nos tradutores dinâmicos, mitigando os canais laterais de tempo inseridos.

Abstract

Timing side-channel attacks are an important issue for cryptographic algorithms. If the execution time of an implementation depends on secret information, an adversary may recover the latter through measuring the former. Different approaches have recently emerged to exploit information leakage on cryptographic implementations and to protect them against these attacks. Therefore, implementation of constant-time cryptography is a widely adopted practice aiming to decorrelate the dependency between a secret data and its timing samples. Despite the countermeasures are effective to guarantee the execution of algorithms in a system by avoiding timing side-channels, emulators can modify and reintroduce leakage points during their execution. Recent works discusses the impact of high level language Just-In-Time (JIT) compilers in leakages through execution time [1, 2]. However, little has been said about Cross-ISA emulation through DBT and its impact on timing leakages.

In this work, we investigate the impact of emulators (dynamic binary translators) on constant-time property of cryptographic implementations. By using statistical methods and cryptographic routines we asserted the feasibility of timing leaks in codes generated by a dynamic binary translator, even using different Region Formation Techniques. We show that the emulation may have a significant impact by inserting non constant-time constructions during its translations, leading to a significant timing leakage. This leakage is observed in dynamic binary translation systems such as QEMU and HQEMU when emulating routines from known cryptographic libraries, such mbedTLS and can be quickly verified. Finally, to guarantee the constant-time property we implemented a compiler transformation based on the if-conversion transformation in the dynamic binary translators, mitigating the inserted timing side-channels.

List of Figures

List of Tables

List of abbreviations and acronyms

CFG	Control-Flow Graph
DBT	Dynamic Binary Translator
ISA	Instruction Set Architecture
IR	Intermediate Representation
HLL	High Level Language
JIT	Just-In-Time
RFT	Region Formation Technique
RISC	Reduced Instruction Set Computer
TCC	Translated Code Cache
TCG	Tiny Code Generator
VLIW	Very Long Instruction Word

Contents

Chapter 1

Introduction

Cryptography is a practice used for centuries aiming to keep private information confidential. Along the years, several cryptographic algorithms have been proposed. Although cryptographic algorithms are designed to be secure against analytic attacks, valuable information can still be extracted by peculiarities in their implementation and execution. This undesired leakage of information, coming from variations in execution [3, 4, 5, 6], system power consumption [7, 8] , memory access pattern [9, 10, 11, 12] and branch prediction behavior [13, 14] is statistically exploited by side-channel attacks aiming to infer secret information used by the algorithms [15].

Among the different types of side-channel attacks, timing attacks is a class of side-channel attacks that try to infer secrets based on the execution time behavior of the algorithm [6]. Roughly speaking, if this behavior depends on a secret value, information can be leaked. These attacks against modern cryptography algorithms were demonstrated possible even when including network noise in remote attacks [16]. Since then, they have become more relevant to various applications in different scenarios, such as IoT, cloud computing and others.

In this way, several implementation techniques to achieve constant-time execution were proposed to protect against timing attacks [17], and have become widely used. In general, these techniques aim to ensure that variations in execution time are not correlated with a secret data and it is possible to find them in various libraries, such as OpenSSL (`openssl.org`), mbedTLS (`tls.mbed.org`) and BearSSL (`bearssl.org`). The rules for constant-time implementation are very conservative and consist of avoiding: branching based on secret data, variable-latency instructions that operates with secret data, table look-ups with indexes derived from secret data, among others. Thus, the binary code must be carefully generated and inspected to guarantee constant-time execution, which can be difficult for larger cryptographic libraries. Furthermore, a compiler can remove some of these modifications and reintroduce leakage points to the code [2]. This means that the final binary needs to be carefully reexamined every time a change is made and the code is recompiled, complicating the task even more.

For this reason, many automatic countermeasures [1, 18, 19, 20, 21] and leakage detection tools [22, 23] were introduced in the literature. Some of them make use of (difficult to create) computation models to remove classes of leakage with soundness while others remove the complexity of setting up the computational model but only mitigate the leak-

age with no guarantee of removing it. In this scenario, dynamic execution systems such as emulators may also have an impact on these implementations either by adding noise to mitigate leakages or by applying optimizations or code transformations that could create them [24].

An emulator is a piece of software that enables the execution of a binary compiled to one Instruction Set Architecture (guest ISA) by another (host) [25]. The guest ISA can be a real one, like emulating an ARM binary on an x86 processor, or a virtual one (has no real hardware implementation) like Java Bytecode. One of the fastest and most common ways to implement an emulator is using dynamic compilation (a.k.a. Dynamic Binary Translation, DBT or JIT Compilation), a technique which starts by interpreting the guest code, then selects hot regions with a heuristic (Region Formation Techniques, RFTs) and lastly compiles/translates these regions into host ISA for faster (native) execution. With emulation becoming more popular, recently studies on JIT protection have also been performed. A new class of JIT side-channel attack also emerged [26]. These attacks consists in inducing timing leakages on JIT code from High-Level Languages JIT compilers. However, little have been studied about timing leakage on dynamic Cross-ISA translators that also addresses other challenges such as: emulating complex instruction sets, status flags register emulation and others.

In a paper published in "Simpósio de Sistemas Computacionais de Alto Desempenho (WSCAD) 2018" [24], we investigate the impact of multiple-target Cross-ISA emulation in constant-time property using a Cross-ISA DBT named OI-DBT, which emulates OpenISA code (an architecture designed for emulation). Thus, we have presented the following contributions:

- We showed that an emulator can interfere with the time-leakage property of some implementations, i.e. mitigating it. With OpenISA DBT, we showed that an emulator can add enough noise into the execution to the point in which it can actually mitigate the timing leakage, by obfuscating it.

- Although we did not see in the experiments the opposite result (an emulator adding leakage to a constant-time implementation), we argued that this is also possible to happen.

- To the best of our knowledge, it was the first work performing an analysis on timing leakages in Cross-ISA Dynamic Binary Translator using a statistical method.

In this work we investigate the impact of multiple-target Cross-ISA emulation in constant-time property of known cryptographic routines, on popular DBT engines, namely: QEMU and HQEMU. We investigated the feasibility of timing leakages in DBT scenarios and observed the introduction of some non constant-time constructions on the dynamic generated code with both DBTs. These constructions leads to a timing side-channel, where a branch controlled by secret data is generated and which can be noted with about 20000 execution time samples, verified by adopting a timing leakage detection model based on Dudect Tool, presented by Reparaz et. al. [22] to a dynamic execution environment, and carefully analysing translated code. Finally, we also discussed and implemented a

solution for generating code with QEMU Just-In-Time compiler mitigating the timing leakage introduced through a safe compiler transformations.

1.1 Contributions

The main contribution of this work include a comprehensive analysis of timing side-channel in Cross-ISA Multiple-target Dynamic Binary Translators, which emulates complex instruction sets (CISC) and Reduced Instruction Sets (RISC) by using valid statistical methods. In summary:

- We show that an emulator can interfere with the time-leakage property of some implementations and verified the occurrence of known timing leakages on cryptographic implementations, by using statistical methods.
- We demonstrate that timing side-channels can be introduced by QEMU and HQEMU JIT compilers when translating code from ARM32 and x86 architectures, due to emulation of conditional codes. We implemented and experimentally validated a solution to mitigate the timing side-channels generated.
- We tested 12 known cryptographic routines on 2 different emulators, some of them using 3 different region formation techniques. We show that the RFTs may have a minimal impact on the leakages presented in the translated codes.

1.2 Organization

This dissertation is structured as follows: Chapter 2 presents a theoretical background of the concepts used in this work including a discussion about Cross-ISA emulation using dynamic binary translation and presenting an overview of QEMU and HQEMU. It also discusses timing side-channel on popular cryptographic constructions and the Timing Leakage Detection model used to detect such existence. Chapter 3 performs a discussion about side-channels in Dynamic Binary Translation and presents results of the timing leakage detection model. It shows the proposed a solution based on if-conversion technique to securely generate dynamic code. Finally, Chapter 4 presents the conclusions and future works.

Chapter 2

Background

This section explain the basic concepts about Virtual Machines and Binary Translation that is used along the work (2.1). It also discusses the common side-channel leakages encountered in the popular cryptographic algorithms/primitives (2.2) and the timing leakage analysis model used to detect the existence of these leakages (2.3).

2.1 Virtual Machines

In general, an emulator consists of a virtualization software executing on a real system (called host), which provides means to execute code compiled for another architecture with the same or different Instruction Set Architecture (ISA) [25]. The virtualized system have their resources mapped to the real system bypassing existing impositions of compatibility and hardware restrictions, allowing a better degree of portability and flexibility of the software. This technology has been useful in a handful of applications such as Virtual Machines (VMs) [24, 27, 28, 29], support of legacy code [30], simulators [31, 32], among others. However, as the ISA of modern CPUs is huge, performing an efficient emulation between architectures is not a trivial task, especially when both ISA have significant differences in their semantics. Also, as Cross-ISA emulators must exploit architecture-specific semantics in order to improve its performance (e.g vectorization) and it may be a challenging task to design an efficient emulator that can be quickly retargetable [33].

The two main ways of implementing an emulator are by interpretation or by translation. The former usually is simpler and assures better portability. It is used to emulate cold (seldom executed) parts of the binary and consists on implementing the fetch, decode and execution cycle, mimicking the behavior of a simple CPU. This approach causes a great degradation on performance, as the emulation of every single instruction alone will require tens of actual host instructions to be executed [25]. The latter, on the other hand, is the one that results in faster emulators (more than 10x faster than an interpreter [34]), but it is less portable and far more complicated to implement. The translation of an ISA can be done either statically, by translating the whole binary application beforehand (Static Binary Translation, SBT), or dynamically, translating hot regions of code from the binary while interpreting it (Dynamic Binary Translation, DBT). Usually a SBT is simpler to implement than a DBT, but given binary properties such as code discovery

problem, self-modifying code and indirect branches, it cannot translate all programs, thus, DBT is more common in real scenarios.

DBT engines usually start by interpreting the binary code and only after some execution it starts translating code. The translation can be done by using various heuristics aiming to achieve a better performance and a better quality of translated code. While interpreting, they also collect execution frequency information which is used by heuristics (Region Formation Techniques, RFTs), further explained, to detect regions of code that form a cycle or have a high probability of being executed many times in the future. Once selected, these regions of code are disassembled and sent to a JIT compiler to produce native code (for the Host ISA) mimicking the behavior of the region in the guest ISA. These translated code are put in a cache, known as Translated Code Cache (TCC), and every time the emulator needs to execute one instruction that is in the start address of one of the TCC regions, the emulator jumps to the native code in the TCC, executing the translated code. Translation (or compilation) in a DBT happens together with binary emulation and its execution time impacts directly to the final emulation performance. However, as the execution of translated code is faster than interpretation, if a region is executed enough times, the translation cost is paid off by the speedup achieved by using the translated code instead of interpretation.

As most of the emulation time is spent executing translated code, the quality of region translation is extremely important for the final emulation performance [27] and RFTs impact directly in this quality because they are responsible for defining the compilation/-translation unit of a DBT engine. For instance, RFTs that select larger regions may have a better performance as they open opportunities for more optimization and, most importantly, reduce the transitions between regions. In multiple-target DBT engines, such as OI-DBT [24], QEMU [28] and HQEMU [27], transitions are expensive because it is not trivial to map register banks between all pairs of architectures they support and then the DBT needs to apply register allocation separately for each region, creating different mappings and forcing each one of the regions to load/save the values of the guest registers when entering/exiting.

An example is showed in Figure 2.1. Selected and translated regions of a guest program are illustrated as R1, R2 and R3. When finish executing a region, DBT engines may jump to another compiled region (using an Instruction Branch Target Cache or Direct Block Chaining mechanisms) or return to the interpreter, if region is not yet compiled. Multiple-target DBT engines add several extra store and loads which are executed every time a region-to-region (as arrows (c), (d) and (e) in the Control Flow Graph in Figure 2.1) or a region-to/from-interpreter (as arrows (a), (b)) transition happens (also called prologue and epilogue codes).

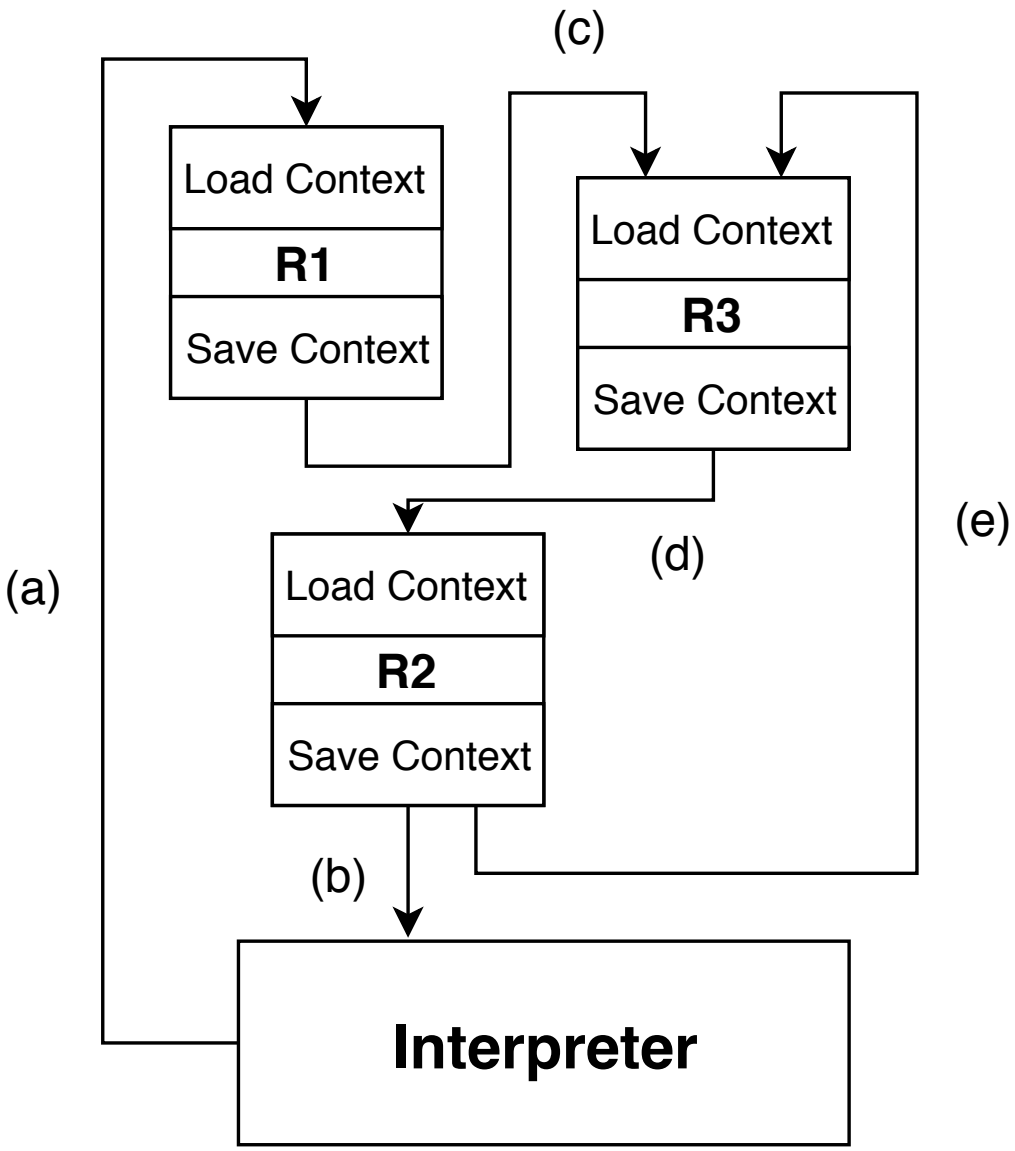

Figure 2.1: Guest machine context load and store when transiting between regions and interpreter.

2.1.1 Region Formation Techniques (RFTs)

A DBT engine needs to decide which regions of code it is going to translate and optimize. This is a responsibility of the RFT which determines when to start recording a region, what to record and when to stop and send it to be translated/compiled. There are several factors that affect the choice of policies for the RFTs: size of the created regions, execution frequency, amount of duplicated code, etc. Hence, RFTs have different purposes.

A näive and simple approach consists in translating one simple region at time (such as basic blocks), when executed by the first time. This is well suited option for small programs, with few basic blocks, but the transition overhead for larger programs is very high. Therefore, different techniques have been proposed in the literature to address this challenge.

NET (Next-Execution Tail)

Initially, Bala *et al.* [35] presented a dynamic binary optmization system called Dynamo, aiming to transparently optimize applications performance. By using a JIT compiler, native instructions are generated from determined pieces of guest code selected by the NET (Next Execution Tail) heuristic. Previously named MRET (Most Recently Execution Tail), NET aims to create super-blocks composed of instructions that are executed in sequence. It considers that every target of backward branches or super-blocks exit is a candidate for starting a region. A region starts to be recorded when the execution of one these points reaches an established execution threshold. When recording, every instruction emulated is added to the new region and it only stops when: a backward branch is executed; another region entrance is found or; a predefined number of instructions included in the region reached a determined threshold.

MRET2

MRET2 [36] executes the NET technique two times and selects the code from the intersection of the result of the two executions. The goal is to reduce tail-duplication.

NETPlus

Davis *et al.* [37], proposes the NETPlus, which extends traces formed by the NET RFT. NETplus first runs NET and then instead of just finishing the super-block formation, also runs a static forward search looking for paths which could extend the region. The search looks for paths which exits the NET region and which returns to its entrance with less than a given number of steps (branches). The authors conclude that this technique tends to select larger regions of code and reduce region fragmentation of NET.

NET-R

Hong *et at.* [27] also mentioned that NET RFT has a problem of region fragmentation, large number of traces and early exits. To address this problem, they presented a modified version of NET, a relaxation of it, called NET-R (Next Execution Tail Relaxed), making it similar to the cyclic-path-based repetition detection scheme [38]. NET-R does not end recording a region when a backward branch is found, instead, it ends when a cycle is found (repeated instruction address), creating larger regions and making room for better intra-procedural optimizations, thus reducing the transition cost.

NETPlus-E-R

Hong *et al.* [27] also proposed the NETPLUS-e-r RFT, which is an extended and relaxed version of NETPlus. It uses NET-R to form regions and when expanding, it does not only include paths which return to the entrance of the region but all paths which return to any part of the region. This technique aims to reduce the region fragmentation, creating larger regions with multiple cyclic paths.

For multi-target DBTs, reducing the number of transitions between regions is essential to achieve good performance (because of the need to save and restore register context). Techniques such as NETPlus-e-r and NET-R have the potential to reduce amount of transitions and may provide the best performances on DBT engines [27].

2.1.2 Dynamic Binary Translators

To address the challenge of Cross-ISA emulation, several Dynamic Binary Translators were proposed and implemented along the years.

FX!32

The FX!32 emulator [39] was developed by Hewlett Packard (HP) in order to support the transition from Windows NT programs compiled for x86 to Alpha hosts. FX!32 combines a x86 machine emulator with a static binary translator, both controlled by a FX!32 server. On the first execution of a binary the emulator captures a profile containing information that supports the static binary translator, such as: addresses of interpreted call instructions, source/target address pairs of indirect jumps, addressees of memory accesses, among others. After the program termination, the server invokes the static binary translator which uses the collected information to translate the x86 instructions into Alpha instructions and optimize the binary. The generated code is stored in a database (translated code cache) and is used when the same binary is executed again. Using this approach, a set of original x86/Windows NT programs is incrementally translated to the Alpha host machine.

Transmeta's Code Morphing Software

Another approach to dynamic binary translation was Transmeta's Crusoe, from Dehnert *et al.* [40]. The custom VLIW processor uses the Code Morphing Software (CMS) that is composed by an interpreter, a dynamic binary translator, an optimizer and a runtime system to emulate the x86 ISA binaries. Transmeta's CMS is a system-level implementation of a complete dynamic binary translation framework that interprets and translates guest machine instructions reliably, implements precise exceptions and handles self-modifying code. The structure of the CMS resembles any common dynamic binary translator and uses an interpreter that decodes and interprets the x86 guest instructions and observes their execution frequency. If a sequence of interpreted instructions exceeds a certain threshold, the CMS passes the address of start of that trace to the translator. The translator then produces a native VLIW translation of the x86 instructions which is stored into the Translated Code Cache (TCC) to be latter executed.

In order to translate x86 instructions into native VLIW instructions, the CMS implements speculations about the interpreted instructions. These speculations are exploited by the translator and verified at runtime by a combination of hardware and software mechanisms. Crusoe registers that holds the x86 machine state are shadowed during the execution of a region (i.e. besides the working registers the CMS contains a copy of the

register file). After finishing the execution of a region, the emulated registers are committed to the working registers (which maintain the system consistency). A failed execution during the execution of a region triggers a native exception that rolls back to the last committed state, and then invokes the interpreter to guarantee precise execution.

OI-DBT

The OI-DBT is an open source Multi-target OpenISA Dynamic Binary Translator[1] that is composed by two major components: one that interprets, profiles and executes dynamically compiled OpenISA code and another one that dynamically compiles selected OpenISA code to native host architecture, based on LLVM On-Demand ORC JIT. The DBT uses at least two threads and its architecture is illustrated in Figure 2.2.

Taking a binary code compiled to OpenISA virtual architecture, the OI-DBT starts the execution by interpreting one instruction per time and profiling it in one thread. The profiling is performed when interpreting a branch instruction and, after the defined threshold is reached, the code trace is then selected using one of the described RFTs. Traces are feed to the JIT engine by placing in a Compilation Queue, that emits native code to the host native architecture, using LLVM IR Builder. The code is placed in the regions cache (the Translated Code Cache, TCC) and every time instructions from that address are interpreted, it jumps directly to the compiled correspondent. Moreover, OI-DBT implements in the architecture a mechanism that dumps the traces selected by the RFTs when interpreting the code. The selected OpenISA code can be later loaded and compiled and also optimized by merging regions.

The JIT engine executes on another thread and is based in the on-demand JIT compiler: LLVM ORC JIT from LLVM 6.0. It takes regions from the Compilation Queue as input, translates, and insert them on the TCC. The JIT takes all the benefits from LLVM back-end and optimizer support to emit optimized native code to several host architectures, on-the-fly.

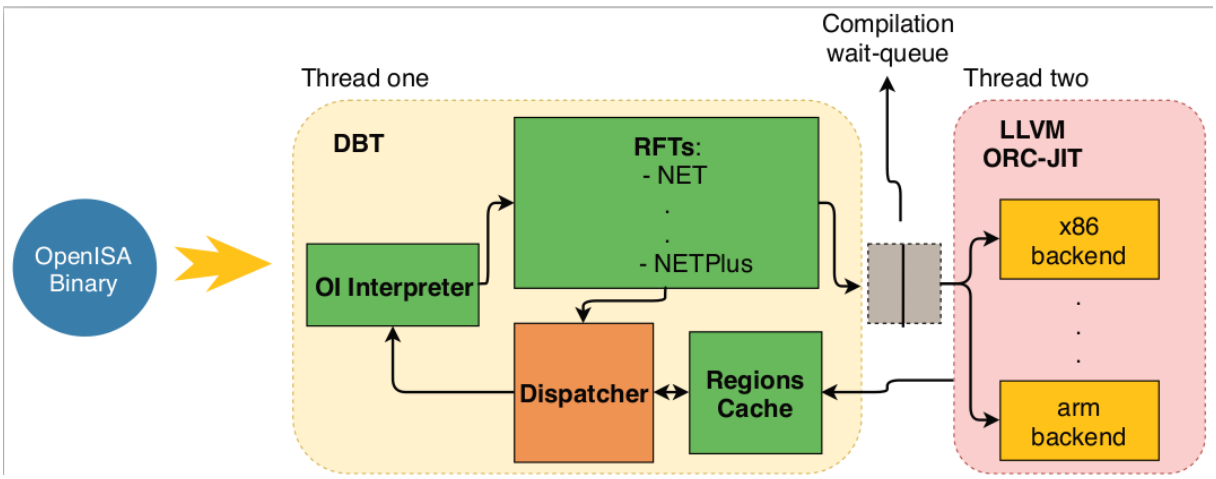

Figure 2.2: Architecture of OpenISA Dynamic Binary Translator (OI-DBT).

[1] https://github.com/OpenISA/oi-dbt

QEMU

QEMU [28] is a widely known retargetable dynamic binary translation system that allows full-system and process-level emulation from several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS).

To perform the emulation, QEMU consists in a main loop that executes a single basic block at time. If the block was already translated, QEMU simply jumps to it, passing through epilogue and prologue codes, when entering and exiting the block, respectively, to maintain internal state coherence. Otherwise, QEMU fetches the basic block from the guest code (named translation block), disassembles it and then maps it to a small set of intermediate representation operations (named TCG Operations) provided by the Tiny Code Generator (TCG), the core of QEMU DBT Engine, to latter be translated to the host architecture, as illustrated in Figure 2.3.

The TCG intermediate instruction set is RISC-like, thus there are only few straightforward instructions supported directly. In this way, a single guest instruction could be mapped to a set of TCG instructions, possibly resulting in poor performance for more complicated ones. Once mapped, TCG improves the quality of its intermediate code by using simple lightweight optimizations passes: register liveness analysis, store forwarding optimization and dead code elimination. Lastly, the intermediate code is translated to the native code, for the host ISA and inserted in the Translated Code Cache to be further executed.

Every transition from a translated block to another incurs passing through an epilogue and a prologue code, that is used to maintain the state coherence (when entering and exiting the translated block, respectively), such as saving the emulated PC state, saving the status flags, among others. In order to reduce transition overhead, QEMU uses Direct Block Chaining and after the translation of each block, it patches unconditional branches directly to the next translation block (if it was already translated), forming chains.

Although the simple and efficient mechanism to translate code, the performance degradation of QEMU generated code is still very severe. Without further optimizations and intraprocedural analysis, there are often many redundant load and store operations left in the generated host code, incurring in a poor quality translated code. Even so, due to lightweight translation system, QEMU is an ideal choice for emulating short-running applications, with few hot blocks.

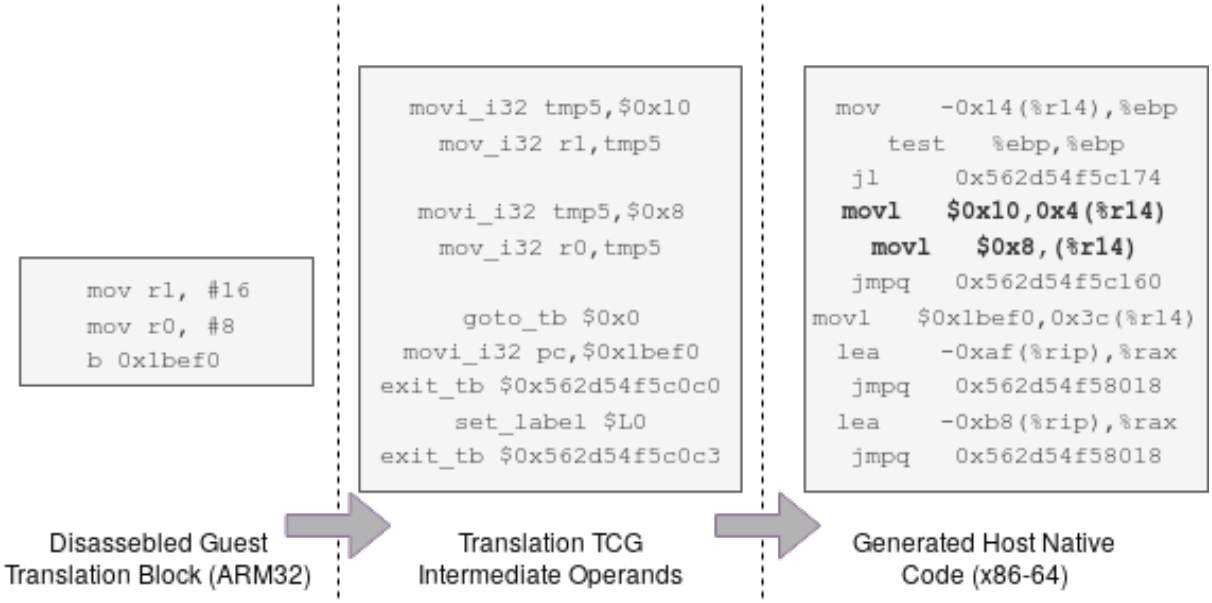

Figure 2.3: An example of QEMU DBT translation. The basic block is fetched from the ARM32 binary (leftmost part), translated to TCG Operations (middle part) and then translated to native x86-64 code.

HQEMU

HQEMU (Hybrid QEMU) is a cross-ISA, retargetable and multi-threaded dynamic binary translator that integrates the LLVM compiler to QEMU. The authors proposed a hybrid scheme using QEMU TCG as a fast emulator and once hot regions are detected, by using the NETPlus-E-R Region Formation Technique, they are optimized by aggressive LLVM toolchain optimizations on another thread. With this scheme, HQEMU generates a high quality and a faster translated code, leading to a slowdown of about 2.5X than native execution (not emulated one) and a speedup of about 4X in comparison to QEMU.

To achieve this HQEMU uses QEMU and translates a single guest binary basic block at time, and emits translated codes to the TCC, named block code cache in HQEMU, as shown in Figure 2.4. A profile is executed in HQEMU and when the emulation module detects that some code region has become hot (some of the translation blocks were executed often, using NETPlus-E-R RFT) it places a request in the optimization request FIFO queue together with the translation blocks in its TCG IR format. On another thread, the requests will be serviced by the HQEMU backend translator/optimizer, similar to a producer-consumer method. When the LLVM optimizer receives an optimization request from the queue, it converts its TCG IRs to LLVM IRs directly instead of converting guest binary from its original ISA.

As the TCG IR consists in about 150 different instructions, this approach simplifies the backend translator instead of translating a much larger instruction sets in most guest ISAs. Many LLVM compiler optimization passes are performed on the LLVM IR and, finally, a highly optimized host code is emitted to the trace cache. Then, every time HQEMU execute a block, it first checks the trace cache to execute a highly optimized code. If the requested block is not in the trace cache (probably because the region is still

cold), the respective block is fetched and executed from the QEMU Block Cache.

The LLVM toolchain counts with a huge set of program analysis facilities and powerful optimization passes that is crucial in generating high quality code. For instance, redundant memory operations that are usually left on QEMU code, could be eliminated via the LLVM register promotion optimization. Also, LLVM aims to select the best host instructions sequences. For instance, it could replace several scalar operations by one SIMD instruction.

These analysis and optimization passes are not simple nor fast and incur in a considerable overhead on the translation system. Nonetheless, since the LLVM translator is running on another thread these overheads are hidden and the translation mechanism are done without interfering with the execution of the guest program [27]. With this design, HQEMU can have a speedup of 4X than QEMU, making it a great choice for long-time running programs.

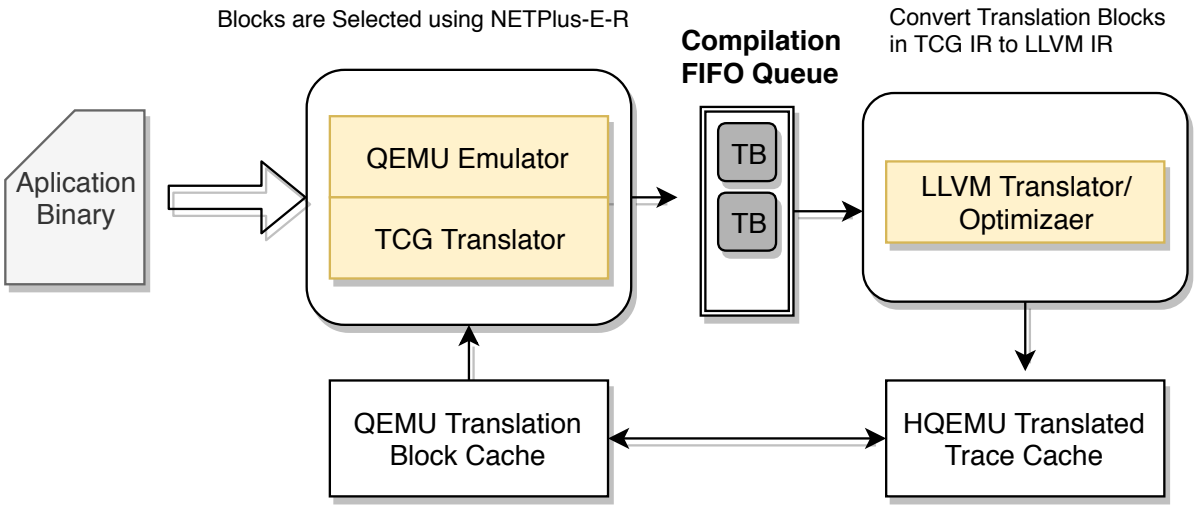

Figure 2.4: Architecture of HQEMU Dynamic Binary Translator.

2.2 Side-Channel Information

Side-channel information is undesired and non intentional information produced by the execution of a cryptographic algorithm. This leaked information (such as variations in execution time, system power consumption, memory access pattern, branch prediction behavior, among others) may be exploited leading to side-channel attacks. These attacks are known by its simplicity and low-cost implementations and proved to be devastating on modern architectures. Although there are many side-channels, a popular class of side-channel comes trough the execution time.

A timing leakage happens when the execution time of a program or the emulation of a program depends on secret information. It was firstly explored by Kocher [6], that demonstrated a successfully key-extraction attack against implementations of the RSA cryptosystem by simply measuring the time taken by the private key operation under execution of several inputs. Kocher concluded that the non constant-time execution of

cryptographic operations could leak some information through timing channels. Thus, in practice, if there is a data dependence between a secret information and the execution time, an attacker executing a program with different inputs and measuring its execution time can infer secret information from the program using statistical methods, even remotely [41]. Along the years, several characteristics that lead to timing channels were explored and showed to be a problem when designing and implementing cryptographic codes.

The three most common implementation/architecture characteristics which end up creating a dependence between data and execution time are: (1) having control-flow depending on secret information, which will lead to different executions paths with a different number of instructions or memory pattern access when using different secrets; (2) changing the memory access pattern depending on the secret information, as it could lead to different cache performance, for instance, indexing an array access with secret data; (3) manipulating secret information with processor instructions that vary their execution time depending on the processed data.

The following Sections describe the common implementation issues in high-level language implementations of cryptographic algorithms that leak secret information through timing channels.

2.2.1 Control-Flow Constructions

The most impacting timing behaviour comes from the variations in the control-flow execution. When a routine executes a different number of instructions for different inputs, it can leak some information if the secret material is used different times. This can be observed in the C function `memcmp` implementation showed in Listing 2.1, from GNU libc. This routine simply compares two byte arrays stored at memory (variables `pq` and `p2`) returning zero for the equality. However, this implementation performs an early exit when the first different value between the two pointers are found (i.e. the value of `v`, in the loop condition, in line 6, is different than 0), causing a significant impact on execution time when the data are different. This leaked timing information, which is the number of iterations of a loop that depends on secret material operations, was also successfully exploited in Google's Keyczar cryptographic library, that uses a similar memory comparison routine to verify signatures codes allowing an attacker to forge signatures for data that was authenticated with the SHA-1 HMAC algorithm.

To mitigate this timing leakage, control-flow statements are usually transformed to branchless versions, using arithmetic and/or bitwise operations. Listing 2.2 shows an equivalently constant-time version of the prior one, using logical XOR and OR operation over the input. Despite this simple example construction, may be a difficult task writing correct constant-time code in high-level languages such as C. Developers must often avoid common language features, like control-flow statements and structure their code to prevent the compiler from introducing timing variabilities during optimization passes.

```
1 EXTERN_C int __cdecl memcmp(const void *Ptr1, const void *Ptr2, size_t
    Count) {
2     int v = 0;
```

```
    BYTE *p1 = (BYTE *)Ptr1;
    BYTE *p2 = (BYTE *)Ptr2;

    while(Count-- > 0 && v == 0) {
        v = *(p1++) - *(p2++);
    }

    return v;
}
```

Listing 2.1: Non constant-time memory comparison.

```
int util_cmp_const(const void * a, const void *b, const size_t size)  {
  const unsigned char *_a = (const unsigned char *) a;
  const unsigned char *_b = (const unsigned char *) b;
  unsigned char result = 0;

  for (size_t i = 0; i < size; i++) {
    result |= _a[i] ^ _b[i];
  }

  return result; /* returns 0 if equal, nonzero otherwise */
}
```

Listing 2.2: Constant-time memory comparison.

Another significant timing variation occurs when performing branches that depend on secret values or when executing constructions that uses secret values in different execution paths.

Modern microprocessors are usually designed to extract a maximum of Instruction Level Parallelism [42] and its pipeline contains many stages, such as instruction fetching and decoding, register allocation/renaming, micro-instructions reordering, execution and retirement. Thus, the microprocessor is able to execute an instruction while next ones are being fetched and decoded. However, conditional jumps proved to be a big challenge in pipeline behaviour, since the execution path is not known before evaluating the respectively condition. To address this problem, the microprocessor usually uses the speculative execution, where the most probable branch of the conditional jump is fetched and fed into the pipeline. The execution result is not retired into the permanent register file, and memory writes are pending until the branch instruction is finally resolved. If the branch was incorrect, the pipeline is flushed, the results of the speculative execution are discarded and the other branch is fetched and fed into the pipeline, resulting in a waste of clock cycles[2] and also a significant variation in execution time.

For instance, the Binary Square-and-Multiply Exponentiation Algorithm is a binary version of the Square-and-Multiply Algorithm (SM), being the simplest way to perform modular exponentiation. The goal is to compute $M^d(modN)$, where d is a n-bit number. The algorithm, showed in Listing 2.3, consists in a fixed count loop (differently from memcmp function) that squares the number M and reduces modulus N and if the private i-th bit of the key is 1, a multiplication is also performed. Aciçmez *et al.* [14]

[2]This missprediction may result in a 15-20 clock cycles in modern Intel Processor.

demonstrated that due to branch predictor behaviour on modern systems a significant timing leakage could be noticed in cryptosystems that uses modular exponentiation, such as implementations of RSA cryptosystem without using the Chinese Reminder Theorem.

```
S = M
for i from 1 to n-1 do
   S = S*S (mod N)
   if d[i] == 1 then
       S = S*M (mod N)

return S
```

Listing 2.3: Binary Square-and-Multiply Exponentiation Algorithm.

A Note for Boolean Expressions

Boolean resulting expressions (i.e. logical and relational expressions) and boolean variables are targets for compiler optimizations which may end up with non constant-time behaviour constructions.

The code showed in Listing 2.4 presents an "unsafe" branchless conditional selection (function `ct_select_u32`, in line 10) between two unsigned integers `x` and `y`, depending on the value of the `bit`, using logical expressions. This code is usually used in asymmetric cryptography to perform a conditional swap. Popular compilers such as `clang` and `gcc` may detect the value range of Boolean resulting operations and variables (in this case the variable `bit` of the type bool, from C's `stdbool.h` library), which usually ranges between 0 and 1. As the concrete range of values can be inferred, Boolean expressions do not lead to Undefined Behaviours and it guarantees stronger assumptions, giving the compiler a chance to optimize the code even more. For instance, the C/C++ as-if rule allows any and all code transformations that do not change the observable behavior of the program, as timing is not considered observable behavior in such languages, compilers can generate some code using different set of instructions than expected, which may affect the constant-time property of implementations. Trough local and interprocedural analysis passes, such as: interprocedural constant propagation, interprocedural propagation of value ranges and interprocedural bitwise constant propagation[3], clever compilers may reduce these series of bitwise operations, within different functions, to a simple evaluation of `bit` variable. The generated assembly, is showed in Listing 2.5 for x86 architectures and in Listing 2.6 for ARM32 architecture. For the x86 architecture, the binary was compiled with `gcc` (version 5.4) using `-m32` flag (for a 32-bit compatibility) and the `O3` optimization level. For the ARM32 architecture, the binary was compiled with `arm-linux-gnueabi-gcc` (version 5.4) using the `O3` optimization level

It's worth to notice that both generated Assembly codes comes with conditional (predicated) instructions (`cmovne` for x86-64 architectures, in line 4, and `moveq` for ARM32 architectures, in line 3), which were selected by the compiler Instruction Selection Phase, aiming to speedup the code generating a branchless comparison version [43]. The use of

[3]These optimizations can substantially increase performance if the application has constants passed to functions and is enabled by default at `-O2` optimization level, in gcc and `-O1` optimization level, in clang.

hardware conditional instructions showed to be executed in constant-time by Coppens *et al.* [44], since it doesn't suffer from branch stalls. Therefore, conditional instructions are widely used on timing side-channel mitigation models, to mitigate control-flow dependable behaviour constructions in binary codes [1, 18, 21, 45].

However, the same code when compiled to an architecture without conditional execution support, such as i386, results in a branched behaviour, as shown in Listing 2.7. The binary was compiled with `gcc` (version 5.4) using `-m32 -march=i386` flags and the `O3` optimization level. In this way, implementations that use Boolean values, must be carefully disassembled and checked to assures the constant-time property.

```
1  /* Return 1 if condition is true, 0 otherwise */
2  int ct_isnonzero_u32(uint32_t x) {
3      return (x|-x)>>31;
4  }
5  /* Generate a mask: 0xFFFFFFFF if bit != 0, 0 otherwise */
6  uint32_t ct_mask_u32(uint32_t bit) {
7      return -(uint32_t)ct_isnonzero_u32(bit);
8  }
9  /* Conditionally return x or y depending on whether bit is set.
       Equivalent to: return bit ? x : y */
10 uint32_t ct_select_u32(uint32_t x, uint32_t y, Bool bit) {
11     uint32_t m = ct_mask_u32(bit);
12     return (x&m) | (y&~m);
13 }
```

Listing 2.4: Constant selection function.

```
1 ct_select_u32:
2   testb %dl, %dl
3   movl  %esi, %eax
4   cmovne  %edi, %eax
5   ret
```

Listing 2.5: Constant-time select generated Assembly code for x86 ISA.

```
1 ct_select_u32:
2     cmp r2, #0
3     moveq r0, r1
4     bx      lr
```

Listing 2.6: Constant-time select generated Assembly code for ARM32.

```
1 ct_select_u32:
2   cmpb  $0, 12(%esp)
3   jne .L8
4   movl  8(%esp), %eax
5   ret
6 .L8:
7   movl  4(%esp), %eax
8   ret
```

Listing 2.7: Non constant-time select generated Assembly code for i386 architecture.

The use of high-level language logical and relational operators (in C: equal, not equal, less than, or, and, etc.) is also a target for the same optimization since it also produces Boolean values, which may not always compile down to constant-time code.

To avoid behaviors like this, some cryptographic libraries (e.g., OpenSSL and libsodium) avoid the use of C's built-in operators and instead, rely on helper functions that implement the operators in constant time. This approach requires re-implementing almost all of C's operators. Other libraries (e.g., mbedTLS and Crypto++) only transform logical and relational expressions when necessary. To this end, their developers inspect the Assembly code and, if the generated Assembly contains branches, they rewrite the C code into a series of expression-statements using bitwise operators. This approach is delicate because it requires developers to examine Assembly generated by different compilers and optimization levels. Even so, carefully used conditional movement instructions can have a great speedup in constant-time constructions, being a good choice for performance and code size.

2.2.2 Table lookups indexed by secret data

Another impacting variation in execution time comes from the memory access behaviour. Based on prior analysis of timing side-channels [6], Kelsey, *et al.* [46] warned that attacks based on cache hit ratio in ciphers with many table look-ups were possible.

In brief, the cache memory is a high-speed Static Random Access Memory (SRAM), which is added to the CPU to reduce the performance gap between the main memory and the CPU. The cache's operation mechanism is very simple. If the CPU needs data, it verifies in the cache first. Using part of the address which represent a cache line (cache-tag) a comparison is made with the values of tag-RAM. If both the values were equal, the data is found in the cache (called a cache hit) and the data is supplied to the CPU to execute without accessing the main memory. In the other case (called a cache miss) the data is fetched from the memory and stored into the cache, for future reuse. Always an entire cache line is fetched from the memory and more clock cycles, compared to a cache hit, are needed until the CPU recieves the data.

In 2005, Bernstein [5] claimed that indexing arrays with secret information leaks information through timing channels due the microprocessor cache performance and demonstrated a successfully key-extraction attack against the symmetric cipher AES.

The Advanced Encryption Standard [47] (AES) is a widely used symmetric-key cipher, based on substitution–permutation network principle [48] and data is encrypted in blocks of 128 bits. Each data block is modified by several rounds of processing (10, 12 and 14 rounds, for 128 bits, 192 bits, 256 bits key material, respectively) into four stages: SubBytes, ShiftRows, MixColumns and AddRoundKey. These operate using mathematical field operations in $GF(2^8)$, which are computationally expensive. In order to improve performance, several implementations make use of pre-computed look-up tables (named S-Boxes), in their SubBytes stage, avoiding the use of field arithmetic. These tables, that are accessed with secret information, usually remains on cache during the encryption process. For instance, some implementations use four 1KB tables, occupying 64 cache

lines[4].

Based on the assumption that execution time of AES is connected to its input, Bernestein carried out the attack by using a reference machine as an identical reference machine with the same implementation as the target machine and collecting a huge number of samples, in this case, the execution time of encryption processes with different plain-texts. Based on the samples, the mean time and variance of each plain-text byte are calculated for all possible values. Afterwards, the correlation between the different positions are calculated. The same analysis steps are performed for the sample of the target device. The attacker compares the results and tries to find similarities in order to extract the secret key information of the target system.

Shortly after Bernstein's report, Bonneau *et al.* [49] defined a general attack strategy using a simplified model of the cache to predict timing variation due to cache-collisions in the sequence of lookups performed by the encryption. Latter Aciccmez *et al.* [3] showed that an attack can be handled even with a network noise and even without the need of knowing the plain-text beforehand, introducing remote cache-timing attacks. This led to several other attacks based on the vulnerability of the data cache and the exploration of shared hardware resources, not limited uniquely to the AES cipher, such as: Last Level Cache (L3-cache) [50], Content-Page Sharing [9], Cross virtual machines [51, 12], among others.

As these attacks proved to be dangerous, popular counter-measures aim to decorrelate the dependency between the memory accesses and the secret material, which is not simple, since several versions of cache attacks comes to spotlight over and over. For this, an interesting implementation strategy aiming to provide constant-time operations and resistance to cache-related attacks is bitslicing[5] [52]. The concept consists in expressing a function in terms of single-bit logical operations (and, or, not, xor, etc.). Thus, instead of having a single variable storing an 8-bit number (as example), eight variables (slices) store each bit of the number (leftmost to rightmost). This may seem a very inefficient way to replace a table lookup, but the bitslicing technique comes with a huge bonus through parallelism. Since all these operations are done with bitwise operators, then they operate on several instances in parallel, simultaneously. Also, bitslicing techniques provided a way to implement efficient code using SIMD operations[6] [17].

Even so, software counter-measures to cache-timing attacks are not limited to bitslicing. Page [53], suggested an obfuscation model by manually adding noise to encryption to make cache timing side-channel attacks more difficult. For instance, Page manually inserted garbage instructions and random loads into the encryption routine. Crane, *et al.* [19] proposed a similar solution through a combination of control flow randomization and garbage code insertion, providing a security trough diversification. Cleemput *et al.* [21] proposed defenses that do not require manual program modification. In particular, they described the use of compiler transformations to reduce timing variability. In their posterior work [1], they proposed an adaptive mitigation model to High Level JIT environments, that apply mitigation transformations to only regions annotated with their

[4]Assuming a cache block of 64 bytes.

[5]Its also known on other areas of computation as data orthogonalisation.

[6]SIMD stands for Single Instruction Multiple Data instructions.

attribute tag named “balanced”.

2.2.3 Variable-latency instructions

Another variation in execution time can come from variable-latency arithmetic instructions. Some older CPUs and even newer ones, may offer shortcuts to perform multiplication and/or division, in a non constant-time fashion, via early-terminating mechanisms, which depend on the input values.

Integer Division

On architectures that provides integer division instructions, hardware implementations often uses early-terminate mechanisms to trigger a faster execution path. Signed and unsigned 32-bit and 64-bit integer division instructions usually execute under different amount of cycles in several modern Intel and ARM CPUs.

For Intel processors, the execution time of an integer division instruction depends on arguments being zero and on the number of quotient bits that need to be generated. This number equals the distance in bit positions between the most significant bits of the divisor and the dividend. Cleemput *et al.* [21] demonstrated that there are limited number of distinct latencies by using early-terminating mechanisms (six for the Core 2 processor, and seven for the Xeon Nehalem core), but the exact latencies and their patterns differ from one architecture generation to the other.

Despite these problems, integer divisions are not common in cryptographic parts that handle secret data and are usually avoided beforehand. Several implementations avoid division instructions by optimizing it into shifts and masking operations.

Data-dependent Shift and Rotation

Even so, architectures without barrel shifter[7] or similar designs may also leak some information through data-dependent shift and rotations based on the shift/rotation count. The Pentium IV processor, for instance, was a processor that did not offer barrel shifters and shift and rotation counts could leak through timing. Also, older instruction set architectures may not be able to perform data-dependent shift operations efficiently. Since many instruction sets only have a fixed shift (e.g. 1-bit) or can only shift by an immediate (e.g. a constant), a data-dependent shift would require a loop that could be a source of side channel attacks in ciphers due to the difficultly of making them operate in constant time. Even on an architecture with an instruction for data-dependent shifts, such as the x86, those shifts will be slower than constant shifts.

Attacks against ciphers using data-dependent rotation have focused on trying to avoid differences in the rotation amounts. This point impacts mostly algorithms that use shift or rotations by amounts that depend on potentially secret data. This was the case for the RC5 lightweight cipher.

[7] A barrel shifter is a logic circuit for shifting a data word by a varying amount, without the use of sequential logic. Usually implemented as a series of multiplexers with the output of one multiplexer connected to the input of another. In modern microprocessors, barrel shifter usually spends a single clock-cycle.

Integer Multiplication

Multiplication instructions may also behave in a non constant-time fashion via early-terminating mechanisms. Similar to integer division, some architectures try to get a speedup in the multiplication by looking at the input operands. This is a problem in some older architectures (80386, 80486) and also in recent ones (ARMv7, ARMv9). For instance, on the ARM Cortex-M3, the `umull` instruction takes 3 to 5 cycles to complete. If both operands are numerically less than 65536 or both operands are powers of 2 the instructions takes 3 or 4 cycles, otherwise it counts 5 cycles [21]

Even so, on systems where the multiplication opcode results in only the lower 32 bits (as ARM platforms in Thumb mode), a 64-bit result will use a software routine invoking the multiplication opcode several times and the operands will be truncated. The truncation may results in smaller values, hence a short cycle count in some cases. The in-use routine may also have conditional branches. Finally, that 32-bit multiplication opcode may perform early exits when the high bits are all equal to one (as ARM9T processors in Thumb mode).

2.3 Timing Leakage Detection Model

To quantify timing leakages, several methodologies were proposed in the literature coming from diverse areas. Gianvecchio and Wang [54] showed an entropy-based model using a Kolmogorov-Smirnov test to detect covert timing channels based on estimation. Chen and Venkataramani [55] presented an algorithm to detect the existence of a covert timing channel tracking contention patterns on shared processors and memory. Becker [56] uses statistical tests with different vector assignments. In this work, we used the Dudect approach, presented by Reparaz *et al.* [22], mainly because of its efficiency and simplicity to reproduce. The same is described next.

2.3.1 The Dudect Timing Leakage Detection Model

This methodology consists in maintaining two vectors with executions times from two different classes of inputs and then comparing the two vectors using some statistical test to evaluate if they represent the same measurement of time. If not, both vectors have different execution times for two different classes of input and, therefore, the implementation is not constant-time.

For each test, what we will call an execution, a cryptography key is chosen and N inputs are randomly generated with an uniform distribution of two classes of inputs. The first class, C_1, consists in only one random plain text which is repeated among all C_1 inputs. The second class, C_2, consists of different random plain texts, one for each input of its class. In other words, N inputs are generated with near to half of them fixed (C_1) and another half of them varying (C_2). A measure of the execution time for each one of these inputs is collected and inserted into its respective vector depending on its class.

Finally, a Welch's t-test [57] is performed (described further), to infer if both vectors are measuring the same execution time. After every execution and before running the

t-test, a post processing is applied in the two vectors, removing some time outliers. As statistical tests such as Welch's t-test can have their results accumulated, we can run this test (of N inputs) E times, each time cleaning the two time vectors, generating a new cryptography key and new fixed plain-texts for C_1 and C_2, and running the t-test to accumulate knowledge about the algorithm being constant or not. This fixed-vs-random kind of test is very popular in the literature and is considered one of the most powerful schemes for detecting timing leakages [58].

Welch t-test

The leakage detection model is based on the Welch t-test [57], a generalization of Student's t-test for unequal variance populations. This model of detection emerged as a convenient solution to perform black-box evaluations of resistance against side-channel analysis by it's simplicity and low sampling complexity [58].

To assess the constant time execution of a piece of code, the assumptions for the test are as follows. The population μ_1 obey a Gaussian Distribution $N(\mu_1, \sigma_1^2)$ as well as μ_2 that obey a $N(\mu_2, \sigma_2^2)$ and both have unknown variances. The Null Hypothesis (H_0) tests if "The two timing distributions are equal", that is, if $\mu_1 = \mu_2$.

Let $\bar{X}_1, \bar{X}_2$ be the means of the populations μ_1, μ_2 and s_1, s_2 the variance from the same populations, which have n_1 and n_2 samples each. The Welch t-test defines the statistical t value by the Equation 2.1. Different from Student's test it does not assume homoscedasticity, which means that the test can be performed over populations with different variances (and not using a common variance) [59]. Besides that, it has a faster convergence ratio with a low number of samples.

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{2.1}$$

Thus, by comparing only two classes one reduces the detection problem to a simpler estimation task. Since these tests are generally applied independently to many leakage samples (e.g. full-block cipher encryption, key agreement calculations), they take advantages of the larger signals that occur for some samples with high probability (i.e. the larger difference of means between the fixed and random classes) dealing better with the presence of interrupts and interference [58].

However, this lower sampling complexity has a risk of false negatives and positives. Regarding false negatives, it may for example happen that for some informative samples, the mean values of the fixed and random classes are identical (or very similar), which makes detection impossible. Yet, by applying the methodology to large enough traces (possibly with a few different fixed classes), the risk that significant leakages remain unnoticed for a complete (e.g., block cipher) implementation is usually expected to remain negligible [58].

Data Pre-Processing

Timing distributions are skewed towards larger execution times. This may be caused by measurement artifacts such as the main process being interrupted by the OS or other extraneous activities. To speed up the test process, measurements with very large cycle counts are discarded (outliers for extraneous activities). Thus, execution time samples which exceeds the average by 100% are removed. For tests with a high number of samples (in our case, more than 100000) a high-order processing is performed, by applying the centered product under the samples.

2.4 Summary

Chapter 2 presented the fundamental concepts of this work, such as emulation through dynamic binary translation (2.1); concerns raised by the sensitive data manipulation and common timing channels introduced in the binary by the programming language semantics (2.2) and; the methodologies commonly used to assess the feasibility of timing leakages in cryptographic codes (2.3). Implementing constant-time counter-measures may be a challenging task due to several optimizations used by compilers and microprocessors to achieve a better performance and parallelism level. In this way, carefully designed constructions must be used to mitigate these timing channels and guarantee the constant-time property. In dynamic compiled environments an extra care also must be taken because the translation may alter this property, which is discussed next.

Chapter 3

Timing Analysis on Dynamic Binary Translators

Although not being a recent problem, side-channel analysis is still relevant and, because of new attacks appearing from time to time, it is frequently in the spotlight. With predictions that in the near future tens of billions of devices will be added to the Internet of Things (IoT), new challenges arise, including the development and deployment of software for a wide variety of devices, architectures and instruction set architectures. Therefore, it also generates new challenges for the side-channel security aspect in JIT-based scenarios.

Brennan *et al.* [26] discussed that JIT compiler mechanisms can be induced to generate timing side-channels if the input distribution to a program is non-uniform. They verified that the Java Virtual Machine (JVM) can be primed to favor certain paths, resulting in optimizations that reduce their execution time. Thus, this can introduce timing side channels even in programs traditionally considered "balanced". Cleemput *et al.* [1], discussed about side-channel when generating JIT code with invariable latency paths and proposes a profile-based JIT protection by applying compiler transformations to regions with leakages. Wu *et al.* [60] discussed about static analysis and transformation-based methods for eliminating cache-timing side channels, which operates in LLVM IR. Several static analyses are performed to identify the sensitive variables and timing leaks associated to the same. Renner *et al.* [2] discussed the timing side-channels in JavaScript runtime systems and pointed that runtime components such as garbage collectors and JIT compilers can trivially introduce timing leaks in constant-time implementations and proposed changes to the WebAssembly language. Other works and counter-measures were also proposed aiming to analyse and maintain the constant-time execution property [21, 44, 61, 62, 63].

However, none of these works studied and analyzed the potential for multiple-target Cross-ISA DBT to change the leakage from a program, adding or removing it, during emulation of a binary. They just perform the analysis over High Level Languages Virtual Machines (HLL-VM). HLL-VMs are built into a virtual-ISA (bytecode) which is designed to support virtualization and portability. It usually includes metadata information to allows type-safe code verification, interoperability, and performance, giving more advantages to optimize code at runtime. For instance, with type information the Java Virtual Machine (JVM) can keep track of exactly what object types a method is being called during runtime. For Object-Oriented Languages which provides polymorphism, the infor-

mation may allow the inline of virtual methods and then perform even more optimizations such as escape analysis, register allocation, constant folding, etc.

However, conventional ISAs are not designed for being emulated and may addresses several issues, such as:

- **Maintaining Precise Exceptions**: Many instructions need the precise CPU state to handle certain exceptions. For this, DBT engines usually insert pieces of code (in the translated code) to update the emulated PC, before potential exception-throw instructions.

- **Different Instruction Set Features**: Cross-ISA DBTs must handle translation for architectures with different register banks. This may lead to a problem when the number of guest registers is higher than the number of host registers, which causes several memory spills. Although, many ISAs use condition codes, which may be difficult to emulate. This is different for HLL virtual-ISA which is usually stack-oriented (not register oriented) and condition codes are avoided.

- **Instruction Discovery During Indirect Jumps**: Indirect Branches may jump to different locations, which are usually avoided in HLLs. Also, the binary program may mix code with data.

- **Self-modifying and Self-referencing code**: DBT engines must also handle codes that alter its own instructions while executing. Usually, this is costly and is done by flushing the entire Translated Code Cache to assure correctness.

In this section, we analyzed the potential leakage impacts when emulating cryptographic routines on Cross-ISA emulation, using the two aforementioned DBTs: QEMU and HQEMU.

3.1 Materials and Methods

In order to verify the feasibility of timing side-channels of constant/non-constant translated cryptographic implementations, we used a timing leakage detection model based on the Dudect Tool, presented by Reparaz *et al.* [22] and two dynamic binary translators: QEMU and HQEMU.

Both the generation of all inputs, the time collection, and the statistical test were implemented together with the cryptographic algorithm and compiled into one unique executable binary, allowing us to factor out the time for starting the emulator and the time to construct all random inputs. We executed 12 algorithms that are used on cryptographic code, between constant-time and non constant-time implementations. Non constant-time implementations include timing leakage through common timing channels introduced in Section 2.2, while the other performs counter-measures to achieve constant-time execution. We also executed these implementations under different RFTs, aiming to verify the differences that could be generated by selecting different versions of the guest code.

The source code of the experiments was implemented in C language and then compiled to four instruction sets: x86, x86-64, ARM32 and AArch64. These implementations are

all divided into four main parts: (1) a routine which generates all random inputs and allocates the time vectors, (2) a loop which iterates over each of the inputs calling the cryptographic routine, (3) the cryptographic routine and (4) the data-processing routine. We only measured the execution time of the third part, the cryptographic routine. For this, we added a special instruction to get the execution cycle before and after the routine execution. In x86 we used the `rdtsc` instruction[1] and in ARM, we use the standard library `clock_gettime` function. After the loop consumes all the inputs, the data-processing routine is called to processes the time-vectors and apply the statistical test in a cumulative online fashion. All these steps can be repeated indefinitely and, after each execution, the statistical test updates its assumptions about the algorithm.

The Welch's t-test statistical test outputs two numbers, the t-value and the p-value. The first shows the magnitude of the difference between the execution time of the two classes (time-vectors) and the second describes the exponential tendency for the t-value, defining a confidence interval for the test. A t-value higher than 4 is usually considered a strong evidence that the two classes have different execution times and the implementation is not constant. In the results, we used the t-value in a categorical form and algorithms with output values higher than 4 are classified as not constant. It is important to note that a presence of leakage is necessary, but not sufficient for a timing side-channel attack to work.

All experiments were executed in two different systems. The first one contains a GNU/Linux Ubuntu 16.04.4 LTS (kernel 4.13.0) operating system and a 4-core 64-bit processor Intel(R) Core(TM) i7-7700 CPU 3.60GHz with 32GB of RAM. The second one contains a GNU Debian 8.10 (kernel 4.4.23) operating system and a 4-core 64-bit processor ARM Cortex-A53 CPU, up to 1.2GHz per core and 1GB of RAM.

Experiments with x86 and x86-64 ISAs were compiled using gcc-7.1 and emulated using QEMU and HQEMU. Experiments with ARM32 and AArch64 architectures were cross-compiled using `arm-linux-gnueabi-gcc` (version 5.4) and `aarch64-linux-gnu-gcc` (version 5.4) and also emulated using QEMU (version 2.12.1) and HQEMU (version 2.5.2). Finally, the native binaries (the ones not emulated), are referenced during the result section as `nat-aarch64` for AArch64 system and `nat-x86-64` for the x86-64 system. All implementations were compiled with the O3 optimization set.

3.1.1 Algorithm implementations

In order to perform the desired experiments in a valid manner, we select 12 routines that are widely used in cryptographic constructions. These are described below.

String Comparison

This is a simple byte-to-byte string comparison given two 128-bit strings as input.

The non constant-time implementation used is the C's library function `memcmp`, as showed in Listing 2.1, that performs an early-termination when the first different byte is

[1] Read timestamp counter (`rdtsc`)

encountered. The constant-time version uses logical XOR and OR operations, walking trough the entire string, as showed in Listing 2.2.

Conditional Selection

Given two input variables and a bit flag, the conditional selection routine selects between the two inputs depending on the value of the flag bit.

The first version is the same showed on Listing 2.4, which is equivalent to a selection using the ternary C's operator. The compiled binaries for this version were generated using conditional movement instructions for all used architectures and did not present side-channels on their native execution, being considered constant-time. The second version changes the type of the Boolean variable `bit` to an integer type (also in `ct_select_u32` functions on Listing 2.4), preventing the compiler to optimize the same to conditional instructions and generating bitwise and arithmetic instructions instead, in a constant-time version.

In this particular case, two equivalent constant-time implementations were used, but with different Assembly code (one uses conditional codes and the another one not).

mbedTLS Get Zeros Padding

The `get_zeros_padding` is a function extracted from mbedTLS library [64], a well-known TLS/SSL library implemented in C and designed to fit on small embedded devices. The function takes a buffer of secret data, whose length is public, and computes the number of elements in the buffer before the all `0x00` suffix.

The first tested version is the original mbedTLS implementation, showed in Listing 3.1. This version scans the input buffer from the tail end until it finds the first non-zero element, exposing the padding length trough timing channels. To safely express memory access computations the mbedTLS developers rewrote the buggy function to iterate over all elements of the buffer, showed in Listing 3.2. This is the second version that we tested, as a constant-time approach from the previous one.

```
1 static int get_zeros_padding( unsigned char *input, size_t input_len,
    size_t *data_len ) {
2     unsigned char *p = input + input_len - 1;
3     ...
4     while( *p == 0x00 && p > input )
5         --p;
6
7     *data_len = *p == 0x00 ? 0 : p - input + 1;
8     return 0;
9 }
```

Listing 3.1: Non constant-time padding length function.

```
1 static int get_zeros_padding( unsigned char *input, size_t input_len,
    size_t *data_len ) {
2     ...
3     for( i = input_len; i > 0; i-- ) {
4         prev_done = done;
```

```
5         done |= ( input[i-1] != 0 );
6         *data_len |= i * ( done != prev_done );
7     }
8     return 0;
9 }
```

Listing 3.2: Constant-time `get_zeros_padding` function.

BigDigits Compare

BigDigits [65] is an open-source multi-precision arithmetic library that implements common functionalities including some cryptographic routines and constant-time operations. The big integer structure is also designed using a byte pointer, a variable storing signal and another storing the size.

The routine `mpCompare` compares two given big integer numbers returning a value for the relation -1, 0 and 1, for less than, equals and greater than, respectively. The comparison executes in a byte-to-byte fashion, similar to the `memset` function, in non constant-time, leaking information through timing channels. The equivalent constant-time routine `mpCompare_ct` is showed in Listing 3.3, which traverses the entire digit and uses a mask to detect inequalities. The binary is often generated using conditional instructions for the comparisons in the lines 6-8 of Listing 3.3.

```
1 /** Returns sign of (a - b) as 0, +1 or -1 (constant-time) */
2 int mpCompare_ct(const DIGIT_T a[], const DIGIT_T b[], size_t ndigits) {
3   /* All these vars are either 0 or 1 */
4   unsigned int gt = 0, lt = 0, c, mask = 1; /* Set to zero once first
     inequality found */
5   while (ndigits--) {
6     gt |= (a[ndigits] > b[ndigits]) & mask;
7     lt |= (a[ndigits] < b[ndigits]) & mask;
8     c = (gt | lt);
9     mask &= (c-1);  /* Unchanged if c==0 or mask==0, else mask=0 */
10  }
11
12  return (int)gt - (int)lt; /* EQ=0 GT=+1 LT=-1 */
13 }
```

Listing 3.3: Constant-Time mpCompare_ct function.

AES

The Advanced Encryption Standard [47] (AES) is a widely used symmetric-key cipher based on the substitution–permutation network principle [48] in which data is encrypted in blocks of 128 bits.

A non constant-time version of AES consists in a tabled implementation, that performs S-Box look-ups depending on a secret value. As shown in Section 2.2, this implementation leaks its behaviour trough timing due to different cache-performance.

A constant-time implementation is obtained trough a bitsliced version, which removes S-boxes and their respectively look-ups.

Curve25519

Elliptic curve cryptography (ECC) is based on a generalized discrete logarithm problem and is widely used for key exchange and digital signatures. Compared to other public-key schemes, ECC provides the same security level as other discrete logarithm systems over a prime field, but with shorter operands [66]. Curve25519 [67] was proposed by Bernstein and is commonly used for the computation of a secret key with the Elliptic Curve Diffie-Hellmann protocol. The equation of the elliptic curve is given by $y^2 = x^3 + 486662x^2 + x$ over the prime field (with the prime $p = 2^{255} - 19$). The curve was designed to be resistant to side-channel attacks, avoiding all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings. For scalar multiplication, Curve25519 uses a specific implementation of the Montgomery ladder allowing the execution of the exact same instructions whether the key bit value is 0 or 1.

Despite the aforementioned efforts to make it constant-time, Kauffmann *et. al.* [68] showed that a dependence of the computation time on the input value can be revealed by comparing the timings obtained by computing multiple times the scalar multiplication with the same key and with different keys. The timing leakage comes from a run-time library routine: `llmul`, which performs the multiplication of two 64-bit integers. A timing difference is introduced when the 32 most significant bits of both operands of the multiplication are all zero. For the non constant-time application, we tested the Curve25519 with a similar run-time library multiplication code, that imposes an early exit when both operands are zero.

3.1.2 Test Parameters

The execution time is measured in cycles which could lead to huge variation in the tests. By running several experiments we realized that the number of samples needed to perform the Welch t-test on Virtual Machines is higher than the native execution. In Multi-threaded DBT engines, the execution time for simple measurements could also be somehow affect by the JIT Compiler. To mitigate this problem, we perform a warm-up phase on the DBT engines, aiming to generate all needed JIT code, before we start to take measurements.

For the tested algorithms we used different values for N based on their execution time and leakage detection speed. The values used are displayed in Table 3.1.

3.2 Experimental Results

The Tables 3.2 to 3.9 exhibit the output values of the leakage detection test on the presented emulators and architectures.

Concerning with the emulation, in HQEMU we performed the experiments using Basic Block RFT (which performs a directly block translation from QEMU TCG to LLVM IR, from every QEMU translation block), NET RFT and NETPlus-E-R RFT. Tables 3.2, 3.4, 3.6, and 3.8 show the leakage results when executing the algorithms on the Intel(R)

Algorithm	**Samples Per Test (N)**
Memcmp	100000
Const. String Cmp.	100000
Conditional Select	10000
Bool. Cond. Select	10000
GetZerosPad	100000
GetZerosPad (const)	100000
BigDig Cmp	100000
BigDig Cmp(const)	100000
AES32	500000
AES bitsliced	500000
Curve25519	8000
Curve25519 (const)	8000

Table 3.1: Number of samples used for each algorithm to perform the Welch t-test.

Core(TM) i7-7700 system (host) with QEMU and HQEMU, using NETPlus-E-R, NET and block selection mode respectively. Tables 3.3, 3.5, 3.7, and 3.9 present the leakage results when executing the algorithms on the AArch64 system (host) with QEMU and HQEMU, using NETPlus-E-R, NET and block selection mode, respectively.

Lines of the table represent the execution of the given algorithm implementation and the columns represent the guest architecture. The results are interpreted categorically, therefore "LEAK" fields shows that the null-hyphothesis was rejected and "CONST" fields shows the opposite. Also, the second and third columns, which are replicated along all tables, stands for the results of the test running natively, without emulation. The 32-bit native executions were omitted, because they offers the same results as 64-bit native executions. The next Section discusses these results.

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	**LEAK**	CONST	CONST	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	**LEAK**	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	**LEAK**	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.2: QEMU leakage results on Intel System.

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	**LEAK**	CONST	CONST	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	**LEAK**	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	**LEAK**	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.3: QEMU leakage results on AArch64 System.

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	CONST	CONST	**LEAK**	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	CONST	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	CONST	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.4: HQEMU leakage results on Intel System Using Block Translation (Basic Block RFT).

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	CONST	CONST	**LEAK**	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	CONST	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	CONST	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.5: HQEMU leakage results on AArch64 System Using Block Translation (Basic Block RFT).

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	CONST	CONST	**LEAK**	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	CONST	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	CONST	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.6: HQEMU leakage results on Intel System Using NET RFT.

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	CONST	CONST	**LEAK**	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	CONST	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	CONST	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.7: HQEMU leakage results on AArch64 System Using NET RFT.

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	CONST	CONST	**LEAK**	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	CONST	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	CONST	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.8: HQEMU leakage results on Intel System Using NETPlus-E-R RFT.

Algorithm	**Nat-aarch64**	**Nat-x86-64**	**ARM32**	**aarch64**	**x86**	**x86-64**
Memcmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Const. String Cmp.	CONST	CONST	CONST	CONST	CONST	CONST
Conditional Select	CONST	CONST	CONST	CONST	CONST	CONST
Bool. Cond. Select	CONST	CONST	CONST	CONST	**LEAK**	CONST
GetZerosPad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
GetZerosPad (const)	CONST	CONST	CONST	CONST	CONST	CONST
BigDig Cmp	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
BigDig Cmp(const)	CONST	CONST	CONST	CONST	CONST	CONST
AES32	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
AES bitsliced	CONST	CONST	CONST	CONST	CONST	CONST
Curve25519 bad	LEAK	LEAK	LEAK	LEAK	LEAK	LEAK
Curve25519	CONST	CONST	CONST	CONST	CONST	CONST

Table 3.9: HQEMU leakage results on AArch64 System Using NETPlus-E-R RFT.

3.3 Discussion

We executed our leakage detection model natively (not emulating) on both systems. These executions showed a well-defined behaviour regarding to the leakage detection. Constant-time algorithm implementations don't reject the null-hypothesis (having the t-value below 4.0) while non constant-time implementations do the opposite, exhibiting a significant timing difference and validating the correctness of the used model during a emulation. This is showed in the Tables, on the second and third columns.

Concerning the leakage during emulation, we verified that the emulator performs a high-quality translation preserving the major algorithm characteristics (in regarding of timing channels) for most part of the algorithms, behaving as expected for the most part of the algorithms. We also noted that non constant-time applications always leaks faster on native execution than emulated one, i.e. it needs less samples to reject the null-hypothesis. This is expected since emulated algorithms have a higher execution time and variation due to handling emulation mechanisms, such as State Mapping, block transition and IBTC, leading to a necessity of more samples for the statistical test. Although the presence of a leakage is not sufficient condition to carry out an attack, it is a necessary condition and then, we also verify its feasibility on emulated code.

The results show that some implementations behave differently (the yellow ones in the Tables) than the native execution when they are emulated, as the case of the emulating ARM32 Boolean Conditional Selection in QEMU, listed in Tables 3.2 and 3.3. In fact, some DBTs emulation mechanisms can compromise constant-time implementations exposing significant vulnerabilities on translated code. In the rest of the section, we analyse these results and discuss the dynamic binary constructions used in the emulators that compromises the constant-time property. Afterwards, we discuss the solutions for these issues.

3.3.1 The Boolean Conditional Selection Case

QEMU DBT

The Boolean Conditional Selection code uses conditional instructions to perform the selection to execute in constant-time in most of modern processors. Our results verify this property in native executions. However, the emulation of the same code with QEMU presents a significant time variation when emulating code compiled for the ARM32 architecture, showed in Tables 3.2 and 3.3. By further investigating, we discovered that the leakage comes from the way that QEMU translates conditional codes from ARM32 architecture.

The 32-bit ARM architecture takes a great advantage of conditional codes and its present in almost all instructions. To assure a simple and portable translator for ARM32, QEMU transforms instructions conditional codes (from guest code) into forward jumps (in its intermediate representation) to the next instruction, that is performed after evaluating the corresponding flags. This introduces a measurable and significant timing behaviour.

The translation for this algorithm is illustrated on Figure 3.1. The native ARM32 binary uses a move on equal instruction (`moveq`) to select between the two incoming parameter values, which comes in `r0` and `r1` registers depending on the value of `r2`. This original branchless version results in a straightforward Control-Flow Graph, showed as a single basic block in leftmost part of the Figure. Then, the QEMU front-end disassembles the block and maps each instruction to a corresponding set of TCG Operations (which composes QEMU Intermediate Representation), as showed in the middle block of Figure 3.1 (after TCG IR optimizations). The conditional movement instruction, `moveq`, is translated to a set of instructions, including a branch instruction (`brcond_i32` TCG opcode) to skip the computation if the condition code is evaluated to false. Finally, the QEMU back-end performs the code generation to the host architecture, which simply maps each of the TCG Opcode to a corresponding set guest instructions directly. The QEMU TCG Opcode `brcond_i32` and its condition code is directly mapped to a conditional jump instruction in both x86-64 and AArch64 back-ends implementations. The translation results in the CFGs displayed by rightmost blocks of the Figure 3.1, for x86-64 host (above) and AArch64 (below) and the red blocks (and arrows) represent the basic blocks inserted in the host generated code to handle the condition part.

Due to the instruction's conditional codes, both translations (to x86-64 and to AArch64) end up with a different CFG from the original, compromising its constant-time property. The inserted branches cause a timing leakage on the implementation which is quickly verified by the detection model, with about 890000 samples for x86-64 system (giving a t-value of 4.15) and about 940000 samples (giving a t-value of 4.07) for AArch64 system. By exploiting this timing leakage, that consists in a branch depending on a secret value, an attacker could determine which of the elements was indeed selected, compromising applications that depend on the constant-time swap.

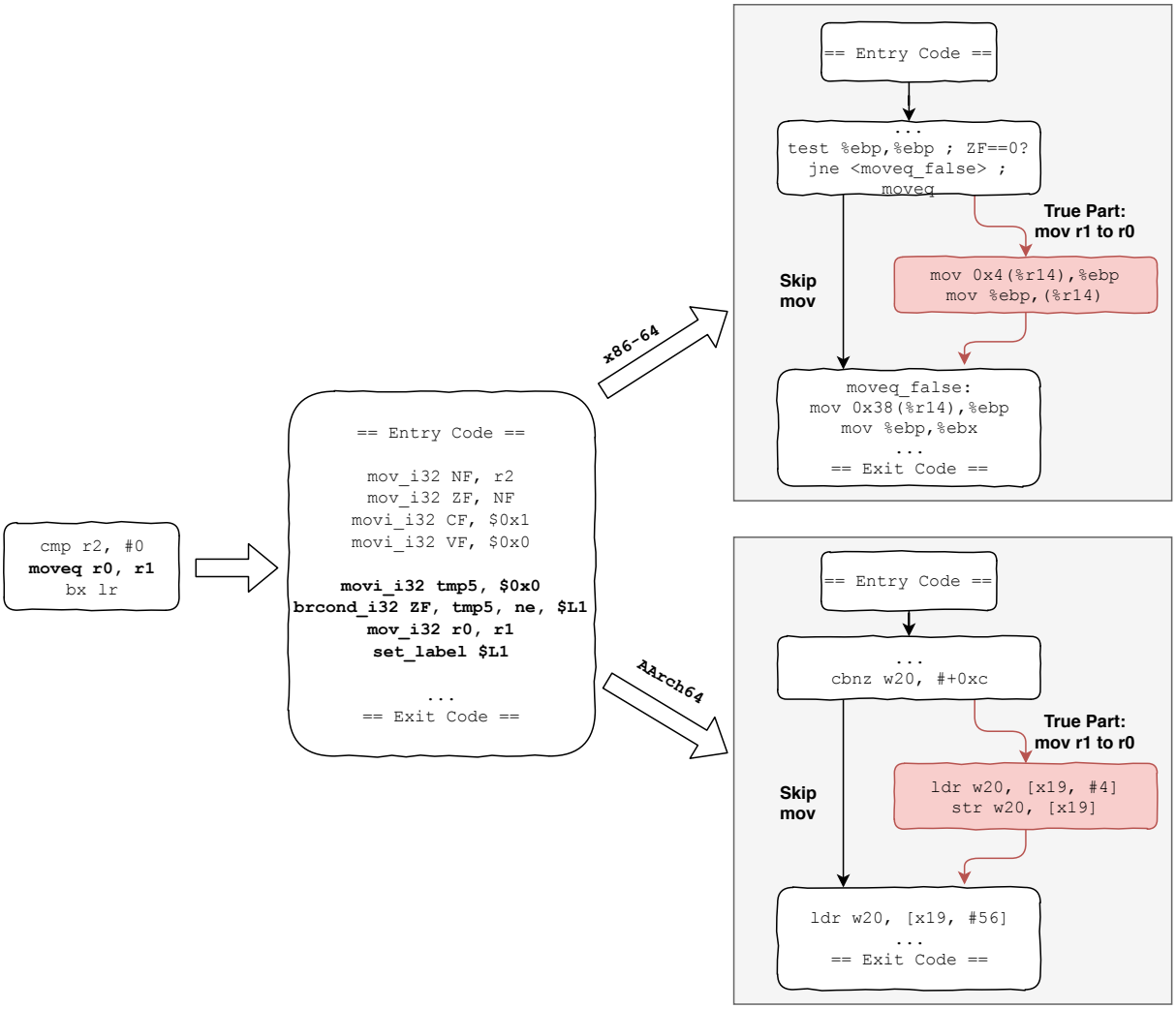

Figure 3.1: Boolean Conditional Select Code Generation in QEMU DBT.

HQEMU DBT

HQEMU generates code using LLVM. To achieve this, it selects the hot blocks using NETPlus-E-R RFT and converts the QEMU intermediate language to LLVM IR. When converting the QEMU Translation Block code to LLVM IR, the branch introduced in QEMU with the `brcond_i32` from the condition flags is also converted to a branch instruction in LLVM IR. However, after LLVM optimization passes (such as SimplifyCFG pass, that aims to remove branches, i.e. PHI nodes from LLVM IR) the forward branch is simplified and reduced to a predicated LLVM `select` instruction since the same can have several advantages. The LLVM IR `select` instruction is used to choose one value based on a condition, without branching. The most common way to implement predicative instructions is to have an arithmetic and/or data access operations followed by a select instruction.

Besides the `select` instruction being able to be translated to a host's conditional instruction (predicated instructions), it also has the advantage of being easier to vectorize (in fact, one of the usual phases of vectorization is the if-conversion pass, the process of

converting control-flow dependencies, a conditional branch, to data-flow dependencies, a select). In this way, the ARM32 conditional instructions for this implementation are also translated to a predicated instruction for the host system and no timing leakages were found, as shown in Table 3.5.

However, the LLVM `select` instruction does not always get translated (lowered) to host's predicated instructions and so it cannot rely in the same to generate codes without side-channels. Similarly to QEMU, HQEMU also leads to a timing leakage as shown in Table 3.4, however, differently from QEMU, the leakage occurs when translating from the x86 architecture. as illustrated in Figure3.2. The x86 `cmovne` conditional instruction in the guest block (showed in the leftmost part of the Figure 3.2) is disassembled and mapped to a conditional TCG Opcode (`movcond_i32`) that is posteriorly mapped to the LLVM `select` instruction.

The translation from architectures with different bit widths (e.g. 32-bit to 64-bit) usually requires an address space translation for memory accesses. Since HQEMU is a process running under a operating system, it has his own virtual memory. When the guest code must perform a memory access, QEMU/HQEMU translates Guest Address (that is address used in the guest binary) to his Host Physical Address (HPA). When generating code, QEMU usually sets up the initial page from his data HPA in the segment register (`gs`, in Intel Processors) and accesses are done with this displacement. For LLVM to perform the address calculation and a memory access to store the desired result may be costly using predicated instruction. Thus, LLVM selects a different set of instructions and a forward branch is inserted, as showed in the rightmost part of the Figure3.2 for x86-64 (above) and AArch64 (below). Thus, as QEMU, HQEMU also compromises the CFG by inserting a significant timing leakage through branches. This leakage can be quickly verified by the detection model, with about 20000 samples, 445x less samples than the leakage on QEMU emulating ARM32.

Also, by using different RFTs to select code, we noted that all of them presented the same leakage point as using NETPlus-E-R, as shown in Tables 3.6, 3.7, 3.8 and 3.9. Consequently all of them showed to be executed in a non constant-time. Thus, even generating larger regions rather than translating basic blocks, LLVM optimizations and transformations still opt to translate the `select` instruction into a forward branch.

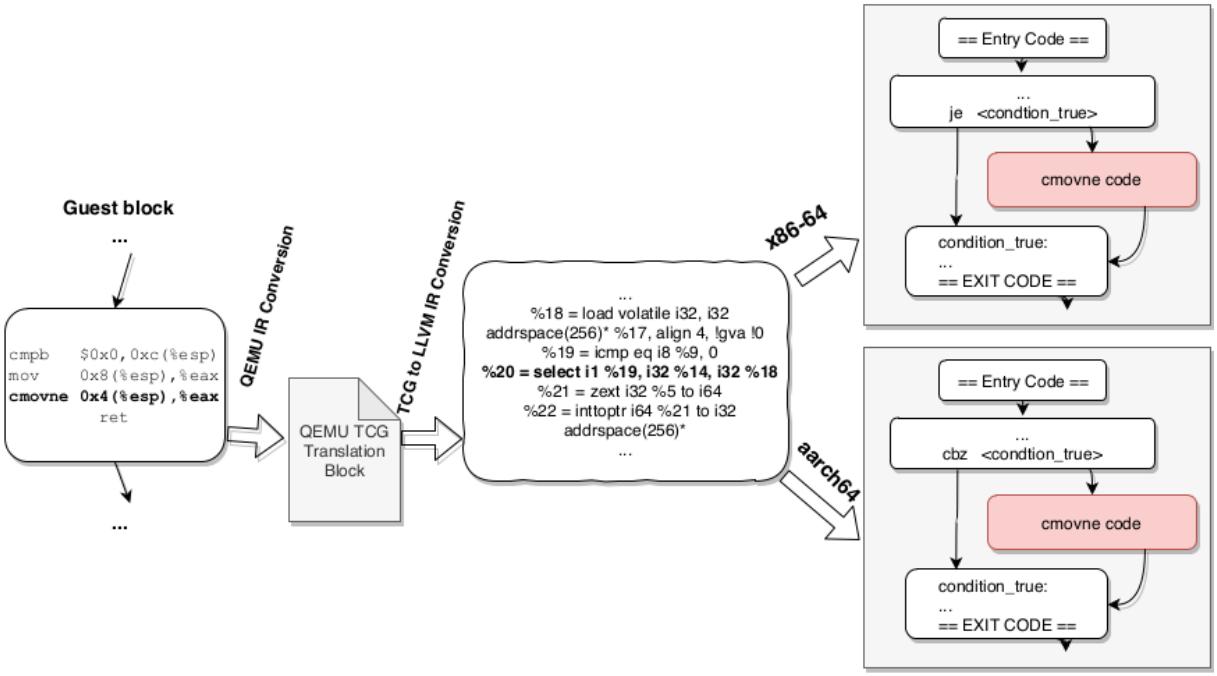

Figure 3.2: Boolean Conditional Select Code Generation in HQEMU DBT (with NETPlus-E-R).

3.3.2 The mbedTLS `get_zeros_padding` Case

The constant-time version of the mbedTLS function called `get_zeros_padding` also presented a significant timing leak on QEMU translating from ARM32, as indicated in Tables 3.2 and 3.3. The major problem also comes from the translation of instructions with conditional codes, similarly to the Boolean conditional select case. In this case however, the ARM32 binary is generated containing two conditional instructions with contrasting conditions: `moveq` and `orrne`, showed on Figure 3.3 in the leftmost part. Also, the two conditional operations depends on the status flags generated by the same compare instruction.

The QEMU's ARM32 front-end generates branches for each conditional instruction on the guest binary, even if subsequent instructions share the same conditional code. Because of the two previously instructions, QEMU generates a slightly different CFG for the translated code regarding the original ARM32 binary CFG, compromising it (showed Figure 3.3 in the rightmost part). This case may be a little worse than the other since two branches that operate with secret values are inserted on the translated code, both x86-64 and AArch64 hosts. This demonstrates that the leakages are inserted in a instruction granularity, that is, for each conditional instruction a branch depending on a secret value is inserted and, hence an attacker could exploit the secret-material resulting after each conditional operations with suitable pos-processing analysis. By exploring this timing leakage, the number of zeros in the pad could be inferred.

Using the leakage test in QEMU, this leakage was detected with about 960000 samples, with a t-value of 4.02, on Intel and 820000 samples, with a t-value of 4.09.

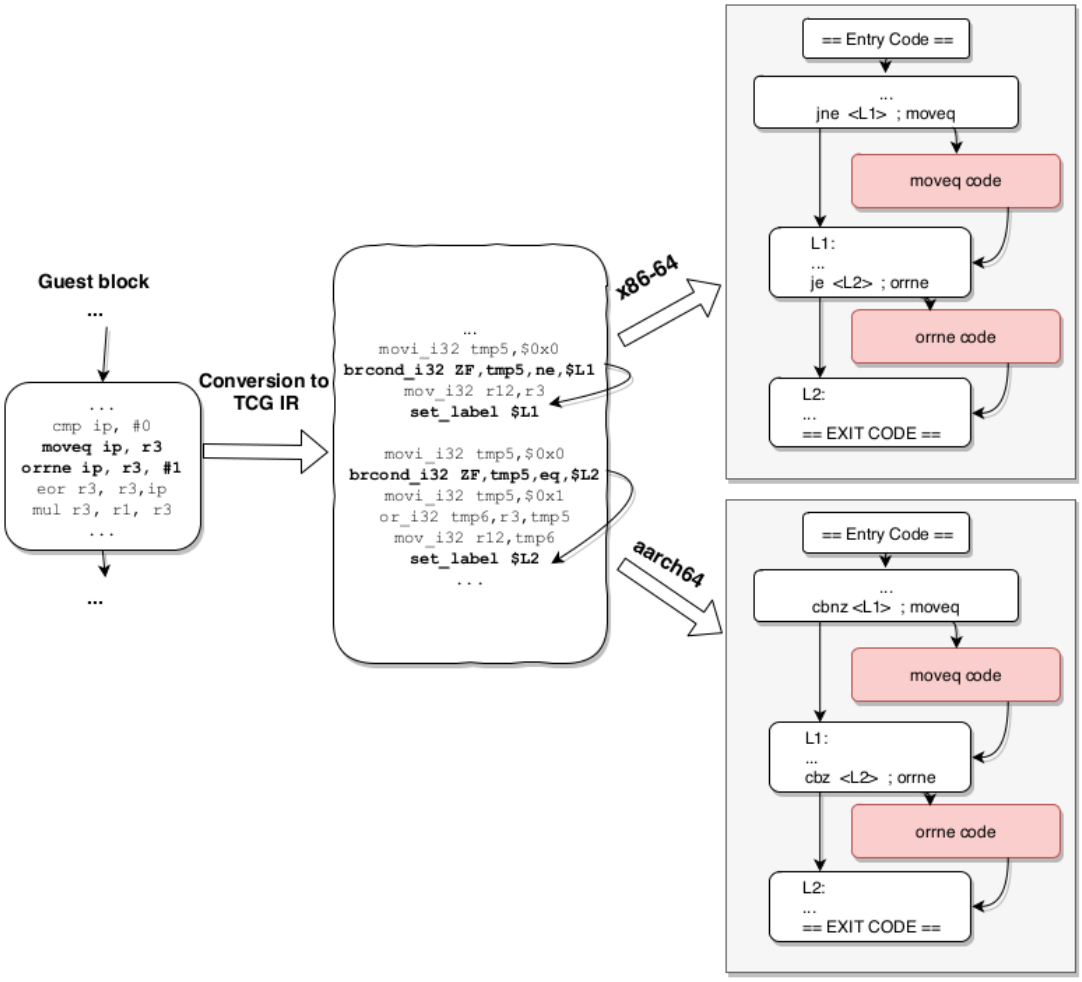

Figure 3.3: Get Zeros Padding Code Generation in QEMU DBT.

3.3.3 The BigDigits Compare Case

The BigDigits Compare implementation also uses conditional instructions in its constant-time implementation. Similarly to the previous cases, it also presents a timing leakage, that is inserted by emulating condition codes in QEMU, as shown in Tables 3.2 and 3.3.

The comparison algorithm is generated with 4 conditional operations on ARM32: `movcs`, `movls`, `andcc`, and `andhi`, as showed in the leftmost part of Figure 3.4. The QEMU TCG then disassembles the block and maps the conditional instructions to 4 branch instructions (`brcond_i32`) in the QEMU intermediate language (Figure 3.4 in the middle). The arrows from `brcond_i32` instructions in TCG IR denotes the skipped conditional instruction – forward branch). All the TCG branch instructions are translated to conditional branches on generated code. The compromised CFG for the translated code is showed in the rightmost part of Figure 3.4, for both x86-64 (above) and AArch64 (below) host architectures. This leakage causes a severe impact on the execution time. Indeed, this leakage is detected with about 450000 samples in the Intel system (with a t-value of 4.38) and about 580000 samples in the AArch64 system (with a t-value of 4.27).

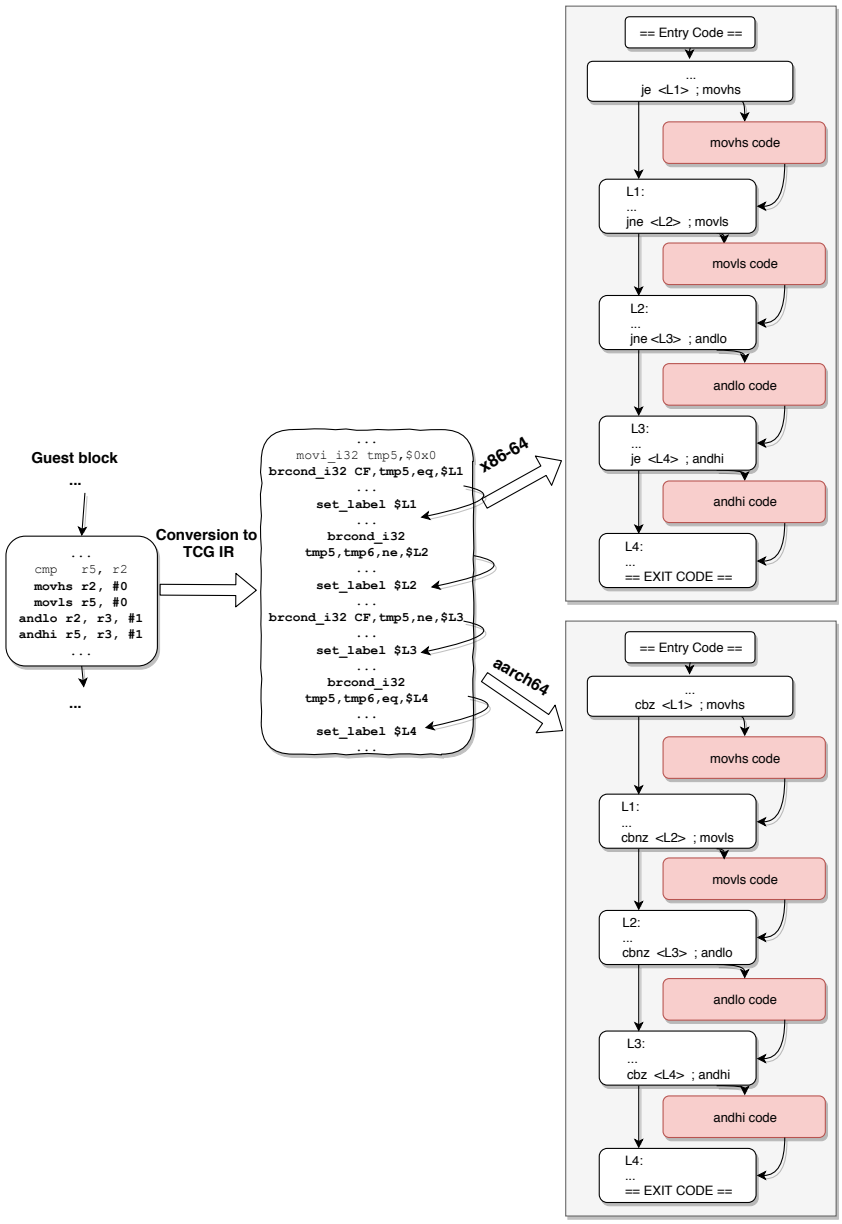

Figure 3.4: Constant-time BigDigits Comparison Code Generation in QEMU DBT.

3.3.4 Other cases

By observing the results, we note that all RFTs behave identically. See the Boolean Conditional Select Case (x86) for instance, even forming different regions, LLVM still optimizes the LLVM `select` instruction by lowering to a conditional branch in both architectures: x86-64 and AArch64. In fact, LLVM `select` instruction can always be lowered to a conditional branch, but LLVM aims to generate a code where branches are executed unconditionally, and all the parameters used in the branches are assigned

conditionally, since it doesn't suffer for branch mis-prediction penalty. Thus, LLVM tries to lower the `select` instruction to a conditional movement instruction on the host code, when:

- Overall cycles needed to execute all the operations of all the branches must be less than the overall possible branch misprediction penalty.
- All the conditions must use the result of a single Assembly comparison operation, or at least two conditions must use the result of an Assembly comparison operation.

LLVM offers no restrictions on the exact number of parameters of the operations within the branches. Although the number of general-purpose registers is restricted, memory variables can also be used, as the penalty of the memory access is insignificant when compared to the penalty of a single mis-prediction. In this way, we cannot expect LLVM to generate a constant-time code for implementations that results in a LLVM `select` instruction.

The QEMU TCG instead, provides the `movcond_i32` operation in its intermediate language. By examining the QEMU back-ends and our experiments, we noted that this operation is always lowered to conditional (predicated) instructions on the investigated host platforms, due the QEMU simplicity and the lack of analysis which could determine whether `movcond_i32` is better translated to a branch or not.

3.4 Counter-measure

In order to assure a constant-time execution, there are approaches to remove or mitigate the aforementioned problems. Molnar *et al.* [69] first introduce methods for detecting control-flow side channel attacks. They model control-flow side channels with a program counter transcript in which the value of the program counter at each step is leaked to an adversary. The proposal rely on hardware to guarantee a one-to-one mapping between the flow of control in a program's execution and all observable behavior, and to rely on source-to-source software transformations to remove any control-flow dependency on cryptographic keys. If control flow is made independent of a key, and if the observable behavior only depends on control flow (i.e. on the trace of program counter values, but not on the values being computed during the program execution), no information about the key can be derived through side channels. Nonetheless, Coppens *et al.* [44] expose some flaws in the source-to-source transformation approach proposed, since it doesn't handle conditional function calls, and conditional loads or stores

Latter, Cleemput *et al.* [21] studied some of counter-measures approaches based on compiler transformations and their effectiveness on x86 processors. However, most of their approaches affect the overall performance by more than 8-fold. In their posterior study [1], the authors extended the enforcement of invariable latency paths with a profile-based JIT protection by applying a selective if-conversion[2] and transformations to regions with leakages.

[2]If-conversion is a transformation which converts control dependencies into data dependencies, i.e. tries to convert conditional branches into predicated instructions.

None of these works studied and analyzed the potential for multiple-target DBT to change the leakage from a program, adding or removing it, during emulation of a binary. Nonetheless these counter-measures can be ported to a DBT scenario. Since Dynamic Binary Translators such as QEMU introduces a branch that depends on a secret-value, when emulating the status Flags, it can be mitigated stripping the branch out.

3.4.1 QEMU

On QEMU, the `brcond_i32` operation in its intermediate representation is always translated to a host branch instructions. The simplest way to eliminate the forward branch generated from a guest's conditional instruction, is transforming the `brcond_i32` to a `movcond_i32` operation. The `movcond_i32` operation, conditionally select a desired value depending on a predicated value (flags, in this case) and is always translated to a conditional instruction on the host platforms (for the four investigated architectures). Since the QEMU already generates the code to be used when the evaluation is true, the result of the same can be saved to a QEMU temporary and, latter can be selected by generating the `movcond_i32` operation, in QEMU front-end. The value of the temporary is selected if the desired condition is true or it just discard otherwise, maintaining the older value.

We implemented this transformation in the QEMU ARM front-end, in the function `disas_arm_insn`[3], which is responsible to disassembles the guest basic block and to maps it to the corresponding QEMU TCG operations. The results of the transformation are displayed in Table 3.10. The table is similar to the another ones, but we filtered only the algorithms which QEMU inserts a timing side-channel for the ARM32 guest. The x86-64 and AArch64 showed in the table are the host architectures, used for translating from the ARM32 guest. The yellow columns (columns 2 and 4 of the table) stands for the results before applying the transformation (labeled as "no transform") to generate code for x86-64 and AArch64 systems and the green ones (columns 3 and 5) stands for the results after applying the aforementioned transformation, also in both systems.

Algorithm	**x86-64 (no transform)**	**x86-64 (w/ transform)**	**AArch64 (no transform)**	**AArch64 (w/ transform)**
Bool. Cond. Select	LEAK	CONST	LEAK	CONST
GetZerosPad (const)	LEAK	CONST	LEAK	CONST
BigDig Cmp(const)	LEAK	CONST	LEAK	CONST

Table 3.10: Counter-measure transformation applied on QEMU translating from ARM32.

It is possible to see that the transformation removes the timing leakage points inserted by the QEMU translator. By stripping out the `brcond_i32` and replacing to a `movcond_i32` all the tested algorithms remained with their t-test values below 4, denoting a constant-time execution. By using the `movcond_i32` operation, branches that were inserted to skip the conditional instructions when the flags are evaluated to false were all translated to predicated instructions on the host platforms and presented a constant-time execution.

[3] `https://github.com/qemu/qemu/blob/master/target/arm/translate.c#L9168`

3.4.2 HQEMU

In the LLVM, however, we cannot rely on the `select` instruction for its intermediate language, since it's not always lowered to a conditional instruction on the host system. Also, using the `movcond_i32` instruction from the QEMU intermediate representation does not resolve the problem, since the same gets translated by HQEMU to a `select` instruction in the LLVM IR. Instead, LLVM provides an if-conversion transformation Pass, that performs the if-conversion efficiently, converting conditional branches to conditional instructions (predicated instructions). As this Pass is architecture-dependent, this must be attached to a Machine Function Pass, and cannot operate on LLVM IR directly. This means that changes in LLVM back-ends must be performed and compiled.

3.5 Summary

Chapter 3 explored some timing side-channels presented in Section 2.2 in a cross-ISA DBT scenario, which has slight different challenges on the translation mechanism than High Level Languages Virtual Machines. Using a statistical method as a black-box evaluator to detect such leakages, we demonstrated that Dynamic Binary Translators can insert timing channels on cryptographic implementations compromising the constant-time property. We exposed the timing channels on two widely known DBTs: QEMU and HQEMU and investigated the root cause of the problem, which comes when translating condition codes from ARM32 guest binaries. Finally, we developed a solution for the aforementioned problem and mitigate the timing channels introduced in QEMU DBT, restoring the constant-time property of the application.

Chapter 4

Conclusions

Timing side-channel attacks are an important issue for cryptographic algorithms. If the execution time of an implementation depends on secret information, an adversary may recover the latter through measuring the former. Different approaches have emerged recently to exploit information leakage on cryptographic implementations and to protect them against these attacks. However, little has been said about Cross-ISA emulation and its impact on timing attacks.

In this work, we analyzed the impact of Cross-ISA emulating a binary with and without timing leakage using QEMU and HQEMU, two popular multiple-target dynamic binary translators. Differently from High Level Languages Virtual Machines, Dynamic Binary Translators also address different challenges as Complex Instruction Set Emulation, which can lead to different emulation mechanisms. By using valid statistical methods to evaluate timing leakages we asserted the feasibility of timing side-channels in dynamic generated code.

In summary, the results show that dynamic binary translators maintain the constant-time property in several implementations and also, leak information through timing-channels when emulating non constant-time implementations. However, translating conditional instructions may lead to a break in the constant-time property. Even if conditional instructions assures a constant-time execution in modern systems, dynamic binary translators such as QEMU, may modify the Control Flow Graph from a binary, inserting non constant-time constructions, compromising the aforementioned property. This leakage compromises several constant-time implementations, including cryptographic codes used in popular known libraries, such as mbedTLS, and can be verified with about 20000 samples of execution times. We investigated these issues, showed the root cause and implemented a counter-measure that mitigate the same. The compiler transformation implemented is similar to the if-conversion transformation and was also successfully demonstrated in other works which aims to mitigate timing side-channels [1, 18, 21] due its efficiency. For this work, the transformation was able to convert branches controlled with secret data to conditional instructions on the host platform, mitigating the timing channels inserted by the emulation mechanisms and maintaining the constant-time property of the application.

Although side-channel analysis was also performed in High Level Languages Just-in-Time scenarios [1, 44], this is the first work of its kind in analyzing a multiple-target

dynamic binary translator impact on side-channel leakage. Cross-ISA emulation diverges from HLL emulation because imposes other challenges on the translation system such as: instruction discovery during indirect jumps, status register emulation, precise exceptions, among others. Given the growing importance of emulators in an internet of things (IOT), fog and cloud computing world with multiple Instruction Set Architectures, this problem will become even more important.

Bibliography

[1] J. Van Cleemput, B. De Sutter, and K. De Bosschere, "Adaptive compiler strategies for mitigating timing side channel attacks," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[2] S. C. John Renner and D. Stefan, "Constant-time webassembly," *Technical Report*, 2018.

[3] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the aes," in *Cryptographers' Track at the RSA Conference*, pp. 271–286, Springer, 2007.

[4] M. Neve, J.-P. Seifert, and Z. Wang, "A refined look at bernstein's aes side-channel analysis," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pp. 369–369, ACM, 2006.

[5] D. J. Bernstein, "Cache-timing attacks on aes," *Technical Report*, 2005.

[6] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual International Cryptology Conference*, pp. 104–113, Springer, 1996.

[7] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*, pp. 388–397, Springer, 1999.

[8] S. Mangard, "A simple power-analysis (spa) attack on implementations of the aes key expansion," in *International Conference on Information Security and Cryptology*, pp. 343–358, Springer, 2002.

[9] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack.," in *USENIX Security Symposium*, vol. 1, pp. 22–25, 2014.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: Sgx cache attacks are practical," *arXiv preprint arXiv:1702.07521*, p. 33, 2017.

[11] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 591–604, IEEE, 2015.

[12] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 299–319, Springer, 2014.

[13] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pp. 312–320, ACM, 2007.

[14] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*, pp. 225–242, Springer, 2007.

[15] Y.-J. Kang, N. Bruce, S. Park, and H. Lee, "A study on information security attack based side-channel attacks," in *ICACT*, pp. 61–65, IEEE, 2016.

[16] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 3, p. 17, 2009.

[17] E. Käsper and P. Schwabe, "Faster and timing-attack resistant aes-gcm," in *Cryptographic Hardware and Embedded Systems-CHES 2009*, pp. 1–17, Springer, 2009.

[18] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 69–76, IEEE, 2017.

[19] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity.," in *NDSS*, pp. 8–11, 2015.

[20] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud.," in *USENIX Security symposium*, pp. 189–204, 2012.

[21] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 23, 2012.

[22] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1697–1702, IEEE, 2017.

[23] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations.," in *USENIX Security Symposium*, pp. 53–70, 2016.

[24] O. O. Napoli, V. M. do Rosario, D. F. Aranha, and E. Borin, "Evaluation of timing side-channel leakage on a multiple-target dynamic binary translator," 2018.

[25] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[26] T. Brennan, N. Rosner, and T. Bultan, "Jit leaks: Inducing timing side channels through just-in-time compilation,"

[27] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores," in *CGO*, pp. 104–113, ACM, 2012.

[28] F. Bellard, "Qemu, a fast and portable dynamic translator.," in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, p. 46, 2005.

[29] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux Journal*, vol. 1996, no. 29es, p. 7, 1996.

[30] K. Woods and G. Brown, "Assisted emulation for legacy executables.," *IJDC*, vol. 5, no. 1, pp. 160–171, 2010.

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[32] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pp. 23–34, IEEE, 2007.

[33] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 177–192, USENIX Association, 2008.

[34] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," in *ACM SIGPLAN Notices*, vol. 46, pp. 74–85, ACM, 2011.

[35] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: Less is more," *ACM SIGOPS*, vol. 34, no. 5, pp. 202–211, 2000.

[36] C. Wang, B. Zheng, H.-S. Kim, M. Breternitz Jr, and Y. Wu, "Two-pass mret trace selection for dynamic optimization," Apr. 6 2010. US Patent 7,694,281.

[37] D. Davis and K. Hazelwood, "Improving region selection through loop completion," in *ASPLOS*, vol. 4, pp. 7–3, 2011.

[38] H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani, "Improving the performance of trace-based systems by false loop filtering," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 405–418, 2011.

[39] R. Hookway, "Digital fx! 32: Running 32-bit x86 applications on alpha nt," in *compcon*, p. 37, IEEE, 1997.

[40] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The transmeta code morphing/spl trade/software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 15–24, IEEE, 2003.

[41] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *European Symposium on Research in Computer Security*, pp. 355–371, Springer, 2011.

[42] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[43] A. Antyipin, A. Góbi, and T. Kozsik, "Low level conditional move optimization," *Acta Cybernetica*, vol. 21, no. 1, pp. 5–20, 2013.

[44] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Security and Privacy, 2009 30th IEEE Symposium on*, pp. 45–60, IEEE, 2009.

[45] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution.," in *USENIX Security Symposium*, pp. 431–446, 2015.

[46] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *European Symposium on Research in Computer Security*, pp. 97–110, Springer, 1998.

[47] T. Jamil, "The rijndael algorithm," *IEEE potentials*, vol. 23, no. 2, pp. 36–38, 2004.

[48] F. Ayoub, "The design of complete encryption networks using cryptographically equivalent permutations," *Computers & Security*, vol. 2, no. 3, pp. 261–267, 1983.

[49] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 201–215, Springer, 2006.

[50] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 605–622, IEEE, 2015.

[51] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 305–316, ACM, 2012.

[52] E. Biham, "A fast new des implementation in software," in *International Workshop on Fast Software Encryption*, pp. 260–272, Springer, 1997.

[53] D. Page, "Defending against cache-based side-channel attacks," *Information Security Technical Report*, vol. 8, no. 1, pp. 30–44, 2003.

[54] S. Gianvecchio and H. Wang, "An entropy-based approach to detecting covert timing channels," *TDSC*, vol. 8, no. 6, pp. 785–797, 2011.

[55] J. Chen and G. Venkataramani, "An algorithm for detecting contention-based covert timing channels on shared hardware," in *HASP*, p. 1, ACM, 2014.

[56] G. Becker, J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, *et al.*, "Test vector leakage assessment (tvla) methodology in practice," in *International Cryptographic Module Conference*, vol. 1001, p. 13, 2013.

[57] B. L. Welch, "The generalization of student's' problem when several different population variances are involved," *Biometrika*, vol. 34, no. 1/2, pp. 28–35, 1947.

[58] F.-X. Standaert, "How (not) to use welch's t-test in side-channel security evaluations.," *IACR Cryptology ePrint Archive*, vol. 2017, p. 138, 2017.

[59] T. Sakai, "Two sample t-tests for ir evaluation: Student or welch?," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pp. 1045–1048, ACM, 2016.

[60] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 15–26, ACM, 2018.

[61] C. Sung, B. Paulsen, and C. Wang, "Canal: A cache timing analysis framework via llvm transformation," *arXiv preprint arXiv:1807.03329*, 2018.

[62] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 377–388, ACM, 2018.

[63] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1406–1418, ACM, 2015.

[64] A. Holdings, "Arm mbedtls."

[65] D. Ireland, "Bigdigits multiple-precision arithmetic source code," 2016.

[66] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

[67] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *International Workshop on Public Key Cryptography*, pp. 207–228, Springer, 2006.

[68] T. Kaufmann, H. Pelletier, S. Vaudenay, and K. Villegas, "When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015," in *International Conference on Cryptology and Network Security*, pp. 573–582, Springer, 2016.

[69] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *International Conference on Information Security and Cryptology*, pp. 156–168, Springer, 2005.