On (the Lack of) Code Confidentiality in Trusted Execution Environments

Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, Srdjan Čapkun
Department of Computer Science
ETH Zurich
{name.surname}@inf.ethz.ch

***Abstract*—Trusted Execution Environments (TEEs) have been proposed as a solution to protect code confidentiality in scenarios where computation is outsourced to an untrusted operator. We study the resilience of such solutions to side-channel attacks in two commonly deployed scenarios: when the confidential code is a native binary that is shipped and executed within a TEE and when the confidential code is an intermediate representation (IR) executed on top of a runtime within a TEE. We show that executing IR code such as WASM bytecode on a runtime executing in a TEE leaks most IR instructions with high accuracy and therefore reveals the confidential code. Contrary to IR execution, native execution is much less susceptible to leakage and largely resists even the most powerful side-channel attacks. We evaluate native execution leakage in Intel SGX and AMD SEV and experimentally demonstrate end-to-end instruction extraction on Intel SGX, with WASM bytecode as IR executed within two popular WASM runtimes: `WAMR` and `wasmi`. Our experiments show that IR code leakage from such systems is practical and therefore question the security claims of several commercial solutions which rely on TEEs+WASM for code confidentiality.**

1. Introduction

The trend of outsourcing data storage and computation has given rise to concerns about the confidentiality of not only data but also of *code* executing on remote (typically cloud) services. To address these broad concerns, confidential computing, based on Trusted Execution Environments (TEEs) such as Intel SGX [1] and AMD SEV [2], has been deployed in today's commercial cloud [3, 4, 5].

TEEs allow the client to deliver their confidential code and data into a protected CPU enclave, which then isolates it from the OS and hypervisor that are running on the same machine and, more generally, from the untrusted Service Provider (SP). This is typically achieved via attestation – the client first sends the public part of its code to the SP (e.g., a VM), attests that this code is running within an enclave, establishes a secure channel (typically TLS) to the enclave, and then uses the secure channel to deliver confidential code and data to the enclave. Once the confidential code is delivered to the enclave, it can be executed in isolation. Recent years have seen an emergence of several designs that generally follow this approach, use different TEEs, and offer various trade-offs, both as academic proposals [6, 7, 8, 9, 10, 11, 12, 13] and commercial solutions [14, 15, 16, 17].

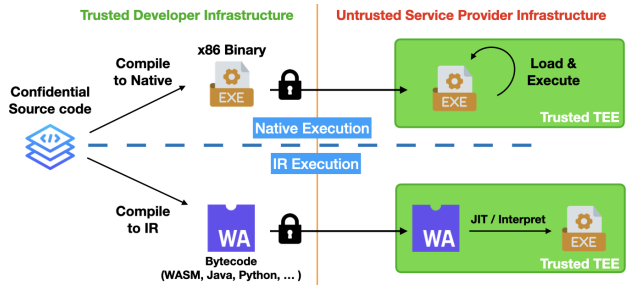

Figure 1. The two main approaches providing code confidentiality with TEEs: native execution (above the dashed line) and IR execution (below the dashed line).

One of the core ways in which these solutions diverge is the format in which the confidential code is delivered to the enclave. They typically follow one of two approaches: *native execution* and *IR execution*, where IR stands for Intermediate Representation. We illustrate these approaches in Fig. 1. In native execution, the developer compiles the confidential code to a *native* binary (i.e., x86) and then, after initializing a remote enclave, sends the binary to it. In IR execution, the developer compiles the confidential code to bytecode (e.g., WebAssembly or Java) or directly sends the source code to the enclave. Whereas in the case of native execution, the enclave can simply copy the instructions from the received binary to memory and execute them, in the case of IR execution, it needs first to convert the received IR into native code. This is done by a Virtual Machine (VM)-like environment in which either a just-in-time (JIT) compiler first converts the IR code to native or an interpreter directly executes it. A number of academic and commercial systems now support either native or IR execution within TEEs. WebAssembly (WASM) runtimes are particularly well supported [14, 15, 16, 10, 12] because WASM requires a small runtime resulting in a small TCB. Moreover, more than 40 programming languages can currently be compiled to WASM, with support for more underway [18].

However, although several [6, 7, 8, 9, 11, 13, 14, 15, 17] of these systems claim to support code confidentiality for native or IR execution, so far, these claims have not been evaluated in the open literature.

Our Paper. We perform the first analysis of confidential code leakage in native and IR execution of modern TEEs. We *quantify* the inherent leakage between these two execution modes, to evaluate whether the choice of the format in which code is supplied to the enclave is a factor that system designers of confidential code solutions should consider. As this problem has not been previously investigated in the literature, an essential step of this analysis is to define how to measure (partial) code leakage. This aspect is not trivial and plays a crucial role in interpreting our results and the conclusions that can be drawn from them. To reason about a deployment mode such as native execution and IR execution irrespective of the deployed binary, we investigate single instruction leakage: we estimate how much the attacker can learn about the distribution of each (IR or native) instruction, and then we compose this across the whole instruction set. Intuitively, if for every possible binary the attacker is able to recover more instructions from IR execution than from native execution then the former is more leaky than the latter.

However, this analysis can be influenced by prior information about the confidential binary, e.g., by knowing which compiler was used, the attacker might be able to exclude certain combinations of instructions. Since prior information needs to be gathered from the developer infrastructure (left side of the vertical orange line in Fig. 1), it is outside of the control and visibility of the adversary, and depends on specific attack scenarios. We decouple the effect of such prior information by fixing it to zero: we assume that the attacker does not know how the confidential binary was created, and only utilizes the public information supplied to the enclave – i.e., the public code that is loaded and attested to before the enclave receives the confidential code. This type of analysis is referred throughout the paper as *"open world"* analysis where no prior information about the target binary is known. In contrast, in a more powerful *"closed world"* analysis the attacker has a higher degree of prior information, e.g., tries to fingerprint specific versions of libraries that they know were included in the target software.

We perform an open-world code leakage evaluation on native execution from Intel SGX and AMD SEV TEEs (x86 ISA) and IR execution with WASM runtimes. In our evaluation, we single-step the enclaves by controlling interrupts and recording various side-channel measurements for each instruction. This allows building a trace of the execution of the victim enclave at the instruction granularity in an attempt to identify individual IR instructions or instruction sequences.

Findings. Our results show that native execution is largely robust to even the most sophisticated side-channel attacks and leaks limited information about individual instructions. This result is based on the fact that in *code leakage* attacks the attacker can differentiate and classify two instructions only if they differ in their metadata. However, by computing the metadata of x86 instructions across several microarchitectures, we find the information present insufficient to perform any meaningful instruction recovery. Thus, even an ideal attacker able to accurately observe the microarchitectural state would have a high uncertainty about which instruction was executed. As we discuss in the paper, this does not mean that a native execution system cannot be attacked, it only means that if the attacker has no prior information about the instruction distribution a code leakage attack is not feasible.

IR execution, however, has shown to be highly vulnerable to our side-channel analysis on two tested WASM runtimes: `WAMR` [19], an interpreter developed by a consortium of companies under the Bytecode Alliance, and `wasmi` [20], an independently developed lightweight interpreter. We successfully leaked more than 45% of the secret instructions with 100% confidence from a synthetic C program running various math and cryptographic functions and from a chess engine written in Rust [21]. Collectively, we successfully extracted over 1 billion WASM instructions from both code bases, albeit not all with 100% confidence. This is possible because we are able to leak around 80% of the instructions in the WASM instruction set architecture (ISA) with 100% confidence. This level of confidence is obtained from just observing one run of the victim enclave.

These results are consistent with the expected side-channel leakage. Each IR instruction is represented by several native instructions. To identify an IR instruction, the attacker can therefore rely on a much longer side channel trace than when it tries to identify an individual native instruction. Therefore, it is clear that IR execution is generally more vulnerable to code leakage than native execution. Our results show that in the case of IR execution, such leakage is also practical, which raises questions about the security guarantees of any IR execution in TEEs.

Contributions. We summarize our contributions as follows:

- To our knowledge, this is the first study to investigate and bring forth the challenges in providing *code* confidentiality in TEEs.
- We generalize system designs aiming to provide code confidentiality in TEEs into two – native execution and IR execution– and develop a methodology to quantify and compare their code leakage.
- We analyze instruction leakage in both systems on various microarchitectures supporting TEEs from Intel and AMD. Our evaluation reveals that native execution leaks significantly less than IR execution. We also show that IR execution greatly amplifies any leakage from native execution and allows us to extract most of the confidential instructions *from a single execution*.
- To demonstrate the practicality of these findings in IR execution, we develop an end-to-end instruction extraction attack against `WAMR` and `wasmi`, two WASM runtime running on Intel SGX. We responsibly disclosed our findings to the affected vendors (cf. Appendix A).

2. System and Attacker Model

We consider a setting in which computation is outsourced while needing to safeguard the confidentiality of

the code. Two main parties are involved in this setting:

- A *Confidential Algorithm Owner* (CAO) that wants to offload computation to the cloud while keeping their code confidential; and
- A *Service Provider* (SP) that provides support for Trusted Execution Environments (TEEs).

While the SP TEEs provide memory confidentiality at runtime, the CAO cannot simply create an *enclave* (a TEE instance) containing the confidential code, ship it to the SP, and expect it to remain confidential: on both Intel SGX and AMD SEV, the initial state of the enclave is visible by the untrusted operating system (OS) and/or the hypervisor. Academic [6, 7, 8, 9, 10, 11, 12, 13] and industrial [14, 15, 16, 17] solutions address this problem by supplying the confidential code to the enclave only after the enclave has been initialized and attested. The confidential part of the code is, therefore, only communicated to the enclave after the attestation and the creation of the secure channel between the CAO and the enclave.

Typically, two approaches are employed to supply and execute confidential code in an enclave: *native execution* and *IR execution*. Each can be further broken down into three stages: (i) compile, (ii) attest, and (iii) deploy and execute.

(i) Compile. In this stage, the CAO compiles its confidential source code for the TEE. Native execution approaches [6, 7, 8, 9, 17, 16, 13] require the CAO to compile to a native format (we focus on x86 object binaries). In IR execution [14, 15, 17, 16, 10, 11, 12], code generally gets compiled to an *intermediate representation* (IR) chosen as a compilation target, e.g., WebAssembly (WASM) bytecode and Java bytecode. In some IR execution systems, the compilation step is skipped as the TEE directly interprets the source code, e.g., the enclave directly receives a confidential Javascript, Python, or Go source code.

(ii) Attest. In this stage, the CAO deploys an initial, non-confidential code with the SP and attests that this code is initialized in the enclave. Attestation ensures that the initial enclave has been deployed in a legitimate TEE and that its integrity is guaranteed. This initial enclave code is often provided by the chosen framework or SP [17, 14, 16]. As part of attestation, the CAO bootstraps a secure channel (e.g., TLS) with the enclave. On this secure channel, the CAO sends either the confidential code to the enclave or a key to decrypt a confidential code image already contained in the initial enclave.

(iii) Deploy and Execute. After the attestation, the CAO instructs the initial enclave to execute the confidential code. In native execution, this is straightforward – the enclave simply jumps [1] to the entry point of the x86 confidential code, which was stored in its memory as a result of the previous stage. In IR execution, the initial enclave contains an interpreter (e.g., WASM or Python), potentially with a just-in-time (JIT) compiler; the confidential instructions get interpreted, and, if a JIT compiler is available, some parts get compiled to native (x86) to speed up the execution.

1. Potentially, after a dynamic linking step.

2.1. Attacker Model

The goal of the attacker is to leak the instructions and, therefore, the confidential code that is executing in the TEE. We assume that the attacker is either the Service Provider (SP) or has privileged access to the server in which the confidential code is executing, i.e., the attacker controls the supervisor software, that is, the hypervisor (on a system with AMD SEV) and/or the operating system (for Intel SGX). This is a standard attacker model for TEEs [22, 23]. The attacker can see the non-confidential, initial enclave code as this code is provided in plaintext to the OS and hypervisor to load the enclave; typically, this code is public. Most of the analysis that we perform is done assuming that the attacker has no prior information about the confidential code: meaning theoretically any instruction combination is possible even ones that no compiler might ever produce. This is referred to as the *open world* analysis. This is done to make sure that different levels of prior information about native execution or IR execution do not bias the results. In line with the open world setup, we assume that the attacker has no control over when the confidential algorithm is executed and which secret inputs are given to it. We deem this assumption to be the most realistic, as it implies that some basic hardening has been implemented. This assumption impacts the side channels available to the attacker, as for instance, in this setting, it is difficult to i) restart an enclave a large number of times to average out noise, and ii) correlate the instructions across multiple runs – as different inputs might lead to different code paths being executed.

Since the attacker has control over supervisor software on the system, they are able to: manipulate interrupts, observe changes to paging management structures (such as page table entries), and other information available to the OS, such as the last branch record (LBR). These capabilities[2] allow the attacker to single-step the TEE execution (through interrupts), see whether memory read and writes are executed (through the page tables), the approximate location (down to the cacheline) of memory read and writes, which code-page is being executed, whether some types of jumps were executed, and the execution time of interrupted instructions. We refer to an attacker with these capabilities as the *state-of-the-art* (SotA) attacker. Note that despite causing a high-level of interrupts the SotA attacker is usually quite hard to detect, as discussed more in detail in Section 9.

Throughout the paper, unless otherwise specified, we employ an open world SotA attacker. However, when necessary to establish upper bounds on code leakage, we use a stronger open world attacker model, which we refer to as the *ideal attacker*. As the ideal attacker is specific to the system for which we want to estimate an upper bound, we will only introduce it when needed in the following sections. When specified, we also study a closed world attacker, particularly to get an intuition about the usefulness of partial code leakage.

2. As demonstrated in the literature against SGX [24, 25, 26, 27, 28, 29] and AMD SEV [30]. Also see Section 9.

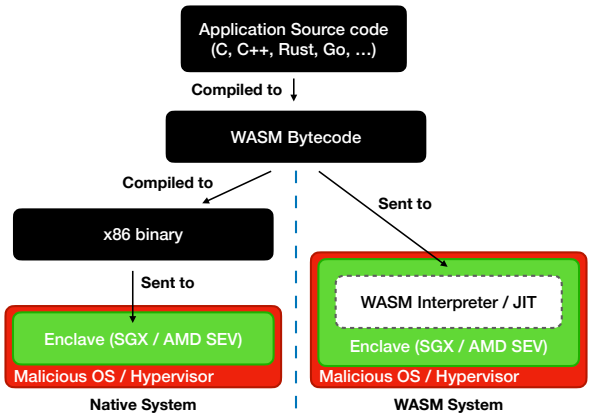

Figure 2. The two approaches to code confidentiality in TEEs. The native execution enclave (left) gets the source code compiled to x86; the IR execution enclave (right) gets as input WASM bytecode. Both systems operate in an environment with a malicious OS and are tasked with executing the same source code.

3. Leakage Analysis Overview

To compare the leakage in native and IR execution, we instantiate them in two systems, the *native system* and the *WASM system*, illustrated in Fig. 2. The *native system* accepts and executes confidential instructions in x86 (native) binary format. The *WASM system* implements IR execution by accepting as input WASM bytecode instructions. The WASM system enclave can then either interpret the bytecode or process it with a JIT compiler before execution. Collectively, we refer to interpreters and JIT compilers as *translators*. We choose WebAssembly (WASM) to evaluate intermediate representation (IR) leakage due to its widespread adoption, ample language support (more than 40 languages can be compiled to WASM bytecode [18]), and the existence of multiple stable and lightweight runtimes. Further, it can easily be compiled into native code, making the comparison between the two systems easier and more rigorous. The enclaves in the two systems get the instructions in different formats from the *same source program*. We compile the source code to WASM bytecode and then the bytecode to x86 outside the enclave (cf. Fig. 2). The native system is given the final x86 binary, while the WASM system is given the intermediate WASM bytecode. Thus, the two systems are tasked with executing the very same program, allowing us to attribute any possible differences in leakage to the system running the instructions.

There are two fundamental differences between the native system and WASM system that influence their susceptibility to side channels: (i) translators often execute more low-level instructions than equivalent native binaries, and (ii) the instruction set architectures (ISAs) of native instructions are usually considerably bigger than the ISAs used for interpreted languages. Combining these two observations, our hypothesis is that the WASM system is potentially leakier than the native system due to having longer (and thus more unique) patterns of execution traces and having fewer possible instructions in the ISA that generate these traces. In the following, we expand on these differences.

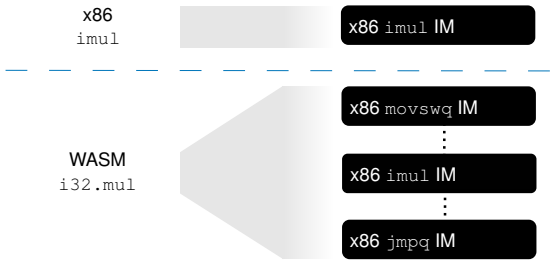

Figure 3. Sample trace collection during the execution of an x86 `imul` and one of its WASM equivalents, `i32.mul`. The x86 instruction generates a single Instruction Measurement (IM), while the WASM instruction generates 9 IMs due to executing 9 underlying x86 instructions.

Number of Executed Native Instructions. Translators of high-level languages with powerful semantics execute multiple native instructions for each high-level instruction. These translators thus *amplify* the amount of information an attacker can collect during the execution of interpreted code, compared to attacking a native system. For example, Fig. 3 shows the difference in collected traces by an attacker when profiling one x86 instruction versus one of its equivalents in a WASM interpreter.

All translators, from high-performance JIT-based to interpreters, must perform two steps to execute a binary: first, they have to parse the code and, second, execute it. Parsing usually involves looping over each instruction of the input code, decoding it, and preparing it for execution (e.g., with a `switch-case` statement, as shown in Listing 1). As the underlying architecture does not provide single complex instructions to perform these operations, multiple native instructions are executed while parsing a single WASM instruction. Additionally, since different WASM instructions require different actions by the parser, the amplified instructions will differ based on which WASM instruction is being parsed. Effectively, this creates an exploitable control-flow dependency. Similar issues arise during execution. For instance, the WASM `add` instruction adds the last two values from the WASM stack and then writes the result back to the stack. An interpreter needs first to read these values and then write the result back, generally using multiple native instructions for this task. In contrast, on x86, it is possible to perform all these operations with a single `add`. In summary, the WASM system enclave executes several (and different) x86 instructions for each WASM instruction *both during parsing and execution*.

While Listing 1 shows the implementation of the `WAMR` [19] loader, other WASM projects we inspected (Wasmtime [31], Wasmer [32], WasmEdge [33], Wasm3 [34], and `wasmi` [20]) have similar implementations. In fact, we remark that the amplification described above with the related control-flow dependency on input instructions is likely to be found in any interpreter or compiler available today. However, different

```
1 static bool
2 wasm_loader_prepare_bytecode(...) {
3   ...
4   while (p < p_end) {
5     opcode = *p++;
6     emit_label(opcode);
7
8     switch (opcode) {
9       ...
10      case WASM_OP_NOP:
11        skip_label();
12        break;
13
14      case WASM_OP_IF:
15        PRESERVE_LOCAL_FOR_BLOCK();
16        POP_I32();
17        ...
```

Listing 1. Excerpt of the main loop of the Bytecode alliance `WAMR` translator [19] (commit `b554a9d`) responsible for loading a WASM binary. `opcode` (line 5) is the opcode of the current WASM instruction being parsed. This listing shows how a control-flow dependency on the `opcode` usually manifests (line 8) in WASM interpreters and compilers, and how different instructions exhibit different amplification factors. For instance, `WASM_OP_IF` (line 14) requires multiple operations to be translated, amplifying the information available to the attacker compared to the equivalent functionality in x86 (usually a single instruction).

implementations will exhibit different amplification factors, as, compared to each other, they might employ a different number of x86 instructions to parse and emulate high-level instructions. This aspect is crucial as it affects the exploitability of the high-level instructions.

Difference in ISAs. The WASM Instruction Set Architecture (ISA) is significantly smaller than the x86 ISA (between ≈6x and ≈14x, depending on the x86 microarchitecture). Since the attacker knows that the enclave accepts only valid instructions, they have fewer instructions to guess from in the WASM system than in the native system. To give a concrete example of why this helps the attacker, consider the `add` instruction in x86 and WASM. In the WASM case, it can only add the two most recent values in the stack, while in x86, many variations are possible, e.g., adding from different locations in memory, from registers, or even vectors. Assuming an attacker that can only leak the opcode (i.e., an `add`), this reveals more information in the WASM system than in the native system.

4. Methodology

In our study, we single-step the enclave to collect information about each executed native instruction. We refer to the information collected for each native instruction as *instruction measurement* (IM). Given the side channels available in our attacker model, each IM contains the following information about an executed instruction: the execution time, the set of accessed code pages, the set of accessed data pages, and for each of the data pages, whether the access was a memory read or write. A series of IMs forms an *execution trace* containing all the information available to the attacker. Note that the trace contains as many IMs as the x86 instructions measured. Thus, in the native system, the attacker collects one IM per confidential x86 instruction. On the other hand, in the WASM system, multiple IMs are collected for each confidential WASM instruction. Finally, we can only measure instructions if they are executed; hence the execution trace only contains IMs related to the executed branches and no information about non-executed code paths.

Features. It is worth noting that not all the information in an IM can be directly used to infer which instruction was executed. This is due to two reasons: first, the measurement might be too noisy, and second, it might be only related to an instruction's inputs and not to its operand. For instance, the side channel used to measure the execution time is subject to noise, and it is, therefore, generally hard to discriminate instructions based on this measurement: a memory read (`mov`) and an addition from memory (`add`) are two very different instructions (in terms of a program's logic) that produce similar timing distributions [35]. Thus, based on the timing information alone, an attacker would not be able to distinguish between the two. With respect to the second reason, knowing the data page that was accessed does not generally contain any information about the instruction type – the relevant piece of information about the instruction is that a memory access was made, not where it was made. On the other hand, knowing whether the stack was accessed does reveal information about the executed instruction type because some instructions only operate on the stack and not on other segments of memory.

Therefore, instead of using the raw numbers contained in IMs, we collect four features: the execution latency (with a resolution of 10 cycles), the type of memory access (read/write or no access), whether the instruction accessed the stack (yes or no), and whether the instruction modified the control-flow (yes or no). We arbitrarily choose a 10-cycle resolution for the attacker to over-approximate the best current attacker capabilities. To the best of our knowledge, even the most advanced attacks that leverage instruction timings show significant noise and are not even close to a resolution of 10 cycles for current TEEs [36, 25]. More details on related attacks can be found in Section 8. Note also that the IM does not include cache access information, despite being within the capabilities of a SotA today. We decided to exclude this information from the IM because the relevant features from this measurement (whether memory was accessed) can already be inferred from the page monitoring controlled channel, which is easier to measure and deterministic. This highlights the difference between recovering instructions compared to data: for data inference, precise memory accesses are important, while for instruction inference, we need to extract *metadata* about the instruction.

Candidate Sets. To be able to quantitatively compare code leakage, we introduce the notion of candidate sets. The attacker forms a candidate set for each instruction they are trying to recover. Let us assume that from the IM, the attacker can deduce that the underlying confidential x86 instruction made a memory read from the stack, e.g., because the IM contains a memory read from a page assigned to the

```
1      ...
2  .loop:
3      mov    -4(%rbp), %ecx   # %ecx = z
4      imul   %eax, %ecx       # %ecx = z * x
5      mov    %ecx, -4(%rbp)   # z = %ecx
6      inc    %eax             # x += 1
7      cmp    -8(%rbp), %eax   # x < y?
8      jl     .loop            # loop if true
9  .out:
10     ...
```

Figure 4. A simple assembly program with a loop that, on each iteration, computes $z = z * x$. The loop iterates y times. The variable x is stored on `%eax`, y on `-8(%rbp)`, and z on `%ecx`.

TABLE 1. ATTACKER VIEW OF THE ASM IN FIG. 4; $y = 2$.

Instruction	Cycles	Memory	Stack Access	Is CF?	Candidate set size*
mov	0 – 10	R	✓	✗	545
imul	0 – 10	-	✗	✗	581
mov	0 – 10	W	✓	✗	86
inc	0 – 10	-	✗	✗	581
cmp	0 – 10	R	✓	✗	545
jmp	0 – 10	-	✗	✓	23
mov	0 – 10	R	✓	✗	545
imul	0 – 10	-	✗	✗	581
mov	0 – 10	W	✓	✗	86
inc	0 – 10	-	✗	✗	581
cmp	0 – 10	R	✓	✗	545
jmp	0 – 10	-	✗	✓	23

Data collected in the Skylake microarchitecture.
*Candidate sets contain only semantically different instructions.

application's stack. Then the candidate set for that IM will contain instructions such as `pop`, `mov`, and `add`, as they can all read from the stack. On the other hand, it will not contain a `push`, as this instruction always *writes* to the stack. More formally, an instruction belongs to the candidate set of an IM if and only if there exists a version of that instruction that would produce a set of observations that matches the IM. The candidate set is useful in that it tells us that the instruction underlying an IM can only be among the ones contained in that IM candidate set. Therefore, if the set only contains one instruction, then the attacker has recovered a target instruction. In general, we can say that the smaller the candidate sets, the more information the attacker collected (i.e., the lower the entropy). The candidate set allows us to compare the leakage in the two systems in the sense that if one system tends to produce smaller candidate sets than the other, then we can say that it is leakier – and by how much. The ISA used in the target system (x86 or WASM) helps to form an initial candidate set. Since the target system can only execute valid instructions, the candidate set of an instruction with an "empty" IM contains all of the instructions of the system's ISA.

Finally, where indicated, we report numbers for *semantically different* instructions in the candidate sets. We define semantic equivalent instructions as instructions that perform the same task but differ only in the input operand size or type (e.g., signed or unsigned). For instance, in WASM, `i32.add` is equivalent to `i64.add`, while in x86, `movq` is equivalent to `mov`. Semantically different instructions are then instructions that are not semantically equivalent. We perform this simplification because we note that generally if a candidate set contains only semantically equivalent instructions, it can be misleading to report a higher number of instructions in it.

5. Leakage Analysis

We now explain how to leverage IMs to build candidate sets for instructions in the native and WASM systems in an open world setting (cf. Section 2.1), and use such candidate sets to measure how much of the confidential code leaks. For both systems, we proceed as follows:

- First, we analyze a simple program: a small loop where each iteration computes the multiplicative product of two numbers, as shown in Fig. 4. It is composed of 6 assembly instructions, where the two numbers are multiplied in line 4. We compile this program to x86 for the native system and to WASM for the WASM system.
- Second, we discuss the IMs obtained from its execution and analyze the candidate set sizes for each instruction.
- Finally, we estimate the leakage of the system by computing candidate set sizes for all instructions in each ISA.

In the following, we first analyze the baseline native system. We start our analysis with the SotA attacker with practical capabilities (e.g., timing resolution of 10 cycles). We then expand the attacker capabilities to account for future attacks with single-cycle accuracy, functional units occupied over time, and more. We use such an unrealistically strong attacker to determine an upper bound to leakage in the native system (Section 5.2). Finally, we analyze the WASM system under the SotA attacker (Section 5.3).

5.1. Leakage in the Native System

We compiled the sample binary from WASM bytecode to x86 and profiled its execution to gather its IMs: Table 1 shows the collected features when the loop is executed twice, and the number of *candidate instructions* on Skylake CPUs that can produce the same IMs. We observe that, despite combining the information from several side channels, the attacker rarely gets a candidate set with fewer than 100 instructions.

In fact, this is not the case just in the example binary of Fig. 4, but it is a consequence of the candidate sets that can be built with the employed side channels. As we will outline in the following, IMs of many instructions belong to the same candidate set, thus making them indistinguishable from each other.

Full ISA. To place all instructions of the ISA into their candidate sets, we need to collect IMs for all instructions (and their variations) in the x86 ISA available in SGX and

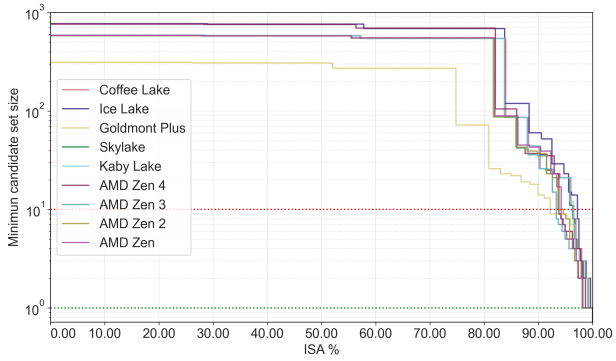

Figure 5. Instruction candidate set size distribution of semantically different SGX and SEV instructions for various 64-bit x86 microarchitectures under the SotA attacker. The plot shows the minimum candidate set size that contains at least x percent of the ISA available in the TEE (SEV for AMD and SGX for Intel). This assumes the best resolution available to the SotA attacker w.r.t. execution time is 10 cycles. The dotted red line is set at $y = 10$, and it indicates that $> 90\%$ of the ISA instructions have a candidate set size greater than 10.

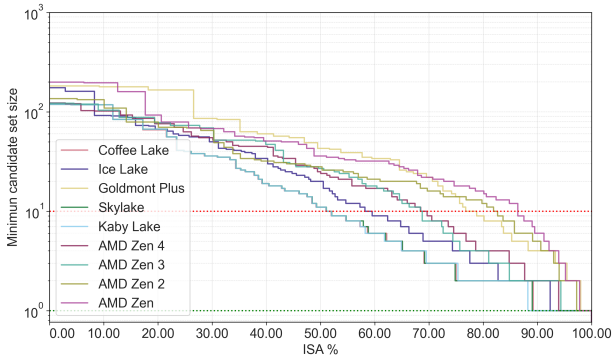

Figure 6. Instruction candidate set size distribution of semantically different SGX and SEV instructions for various 64-bit x86 microarchitectures under the ideal attacker. The dotted green line is at $y = 1$. Given where it intersects the various microarchitectures' ISA, it indicates that more than 90% of the instructions cannot be recovered even by the ideal attacker.

SEV. Further, we would also have to do this for different microarchitectures, as these support different extensions of the x86 ISA and thus change the set of available instructions. Instead of generating programs to execute all possible instructions on different microarchitectures, we adapted and reused the results of a dataset collected as part of an x86 benchmarking suite for the x86 ISA [35]. Particularly, we inferred from the dataset to which candidate set among the ones introduced above every instruction belongs. The dataset had to be adapted to account for the fact that some instructions are illegal in SGX or that others are intercepted by the hypervisor on SEV. We describe these caveats in Appendix B.

By computing all possible candidate sets, we can check how many instructions of the ISA have a candidate set size below a certain threshold, with the idea that the smaller the overall candidate set sizes are, the leakier a system is. Observe that matching IMs to candidate sets and thus candidate instructions is the exact problem an attacker aiming to leak confidential code tries to solve. We report the cumulative distribution of the sizes of the candidate sets in Fig. 5. What can be observed from the figure is that around 80% of the instructions of the ISA belong to a candidate set containing more than 100 instructions. Note that for SEV, 1.48% of the instructions in the ISA belong to a candidate set of size 1 and can, therefore be leaked to the attacker. This is because in SEV, some instructions, such as `CPUID`, are intercepted by the hypervisor and are, therefore leaked to the attacker (not through side channels, but through a system interface). There are a few other instructions with a candidate set size < 10, but they are limited to less than 8% for all analyzed microarchitectures. Thus, the open-world SotA attacker is practically never able to resolve any instruction of the x86 ISA *based on the evaluated side-channels information alone*.

5.2. Ideal Attacker

For the native system, we also explore different strengths of attacker models, for instance, showing how the candidate set sizes change based on different levels of cycle accuracy available to the attacker. We present these results in Appendix C and discuss in Section 8 how these resolutions map to known attacks. Here instead, we study what we believe to be the extreme in terms of attacker strength, which we refer to as the *ideal* attacker. The ideal attacker has the capability of benchmarking instructions – like done in [35]. Note that [35] is a general method to benchmark instructions outside the enclave and hence uses capabilities currently blocked by SGX and SEV, such as reading performance counters and injecting instructions around target instructions. These capabilities also allow the attacker to observe the utilization of individual functional units and obtain cycle-accurate execution time for each instruction. We assume that the other security properties of SGX and SEV otherwise hold, e.g., the ideal attacker cannot read the enclave memory. To the best of our knowledge, the data on single instructions collected in [35] is the most detailed and comprehensive dataset about the performance of current x86 processors to date. Since current attacks do not even get close to the resolution and wealth of information available in [35], the ideal attacker is currently far from realistic. Crucially, however, these capabilities allow the attacker to see all the *metadata* about instructions known today. Meaning that if instructions do not differ for the ideal attacker it is because their implementation uses the same hardware resources in the same order. For instance, an x86 `add` and an `sub` in the microarchitectures we studied use the same resources with the same timings and power consumption. This analysis, therefore, allows us to establish an upper bound of leakage (microarchitecture dependent) that can be achieved by employing (next-gen) side-channel information about x86 instructions. If a *new type* of instruction metadata is discovered in the future, this conclusion might need to be revised.

To build the candidate sets for the ideal attacker, we construct the IM using the data in [35] as follows: cycle-accurate execution time, functional units (FUs) occupied over time, the byte length of the instruction, whether data addresses are accessed, and the type of data access (read or write). Regarding the FUs, for each instruction, we let the attacker perfectly see the order in which they are used and which other FUs could be used by the instruction. Using these very detailed IMs, we create the candidate sets by grouping together all x86 instructions for which the information in the IM is exactly the same. Finally, we remove duplicate entries that are semantically similar, e.g., `mov` and `movq`. The resulting cumulative distribution of the candidate set sizes is depicted in Fig. 6.

While the resulting candidate set sizes are significantly smaller than for the SotA attacker, around 50% of the ISA still belongs to a candidate set of at least size 10 for all analyzed microarchitectures. On the other hand, up to 10% of instructions are uniquely identifiable with a candidate set of size 1 on both SGX and SEV. Based on these results, the ideal attacker might be able to extract some instructions, but the majority of the ISA still remains ambiguous and cannot easily be leaked. Therefore, we conclude that the studied microarchitectures are implemented in such a way that not enough information is available to reconstruct most confidential x86 instructions from the IMs side channels. Note that this conclusion applies in the open world setting – that is, if no prior information about the binary executing in the enclave is given. An attacker with more prior information might be able to extract more instructions even in the native execution system. Prior information might include information about the distributions of compiler-emitted instructions or even the distribution of instructions about key code segments (such as function epilogs). At present, the weight that prior information plays in this leakage has not been evaluated, and we leave this to future work.

5.3. Leakage in the WASM System

We again first consider the loop of Fig. 4, compiled to WASM in the open world setting. However, while in the native system the binary only gets executed, we note that WASM translators (AOT, JIT, and pure interpreters) generally have two phases: *loading* and *interpretation*. During loading, the WASM binary is parsed, and each instruction is decoded into some internal and implementation-specific format. The second phase encompasses the execution of the loaded WASM binary.

We choose to analyze the `WAMR` [19] and `wasmi` [20] interpreters because they combine aspects of both a JIT compiler and a pure interpreter. During the loading phase, `WAMR` and `wasmi` parse the WASM instructions and eliminate instructions whose results can be statically determined. For instance, the `WAMR` loader optimizes away instructions that load constant parameters by pre-placing their constants into the WASM stack before execution. This optimization speeds up the interpreter, as only a subset of instructions needs to be executed later. This pre-processing of instructions makes

TABLE 2. ATTACKER VIEW OF THE LOOP IN FIG. 4 WHEN COMPILED TO WASM; $y = 1$. WE REPORT THE NUMBER OF IMS RECORDED AND THE CANDIDATE SET SIZE BOTH WHEN LOADING (AS DONE IN THE FIRST JIT PHASE) AND INTERPRETING EACH WASM INSTRUCTION.

Instruction	# of IM per instruction				Candidate set size*			
	JIT Loader		Interpreter†		JIT Loader		Interpreter†	
	`WR`	`wi`	`WR`	`wi`	`WR`	`wi`	`WR`	`wi`
`loop`	66	122	-	-	1	3 (1)	-	-
`get.local`	63	55	-	29	1	1	-	1
`get.local`	62	55	-	29	1	1	-	1
`get.local`	63	55	-	29	1	1	-	1
`i32.mul`	33	27	9	29	4	21 (14)	6	1
`i32.store`	91	29	14	43	1	11 (5)	1	1
`get.local`	63	55	-	29	1	1	-	1
`i32.load`	91	29	14	44	1	11 (5)	1	2 (1)
`set.local`	80	56	-	26	1	1	-	1
`get.local`	63	55	-	29	1	1	-	1
`i32.load`	91	29	14	44	1	11 (5)	1	2 (1)
`set.local`	80	56	-	26	1	1	-	1
`get.local`	62	56	-	29	1	1	-	1
`i32.const`	55	29	-	25	1	9 (4)	-	2 (1)
`i32.add`	33	27	9	29	4	70 (33)	6	2
`local.tee`	97	56	7	26	1	1	2	1
`get.local`	62	55	-	29	1	1	-	1
`i32.lt_s`	33	27	11	30	4	70 (33)	6	10 (6)
`br_if`	35	181	14	35	1	1	1	1
`end`	67	46	-	-	1	1	-	-

`WR`= `WAMR` (same version as Listing 1); `wi`= `wasmi` (commit `e87021b`).
*Candidate sets include all instructions. If the candidate set can be simplified, semantically different instruction counts are reported within parenthesis.
†Some instructions are simplified by the JIT loader and are thus not present in the interpreter trace. Non-**bold** instructions are skipped by the `WAMR` interpreter. *Italics* instructions are skipped by the `wasmi` interpreter. With multiple loop iterations ($y > 1$) only non-skipped instructions repeat.

the loading phase of `WAMR` and `wasmi` akin to a JIT compiler. Multiple native x86 instructions are executed for each WASM instruction during both phases. Thus, each WASM instruction of the loop of the sample program lets us collect multiple IMs: we report them in Table 2. In WASM, the loop is composed of 20 instructions. The `WAMR` loader simplifies 12 of the WASM instructions in the loading phase, leaving 8 instructions (in bold in the table) to be executed in the interpreter phase. The `wasmi` loader simplifies only two of those (in italics in the table). In total, for `WAMR` (`wasmi`) we recorded 1290 (1100) IMs in the loading phase of the loop and 184 (1120) IMs in the interpreter phase; with two loop iterations. Between loading and interpreting the loop, the WASM system presents an increase in instructions executed between 123x (`WAMR`) and 185x (`wasmi`) compared to when the same code is executed in the native system.

Our goal is now to understand how *unique* each trace of IMs for each of these WASM instructions is. For this, we profiled each WASM instruction (see Section 6 for more details) and obtained their traces of IMs. With this profiling, we build candidate sets for the WASM system, considering the information obtained from multiple IMs to differentiate instructions. Table 2 reports the candidate set sizes for the instructions in the loop. For several instructions, the attacker gets candidate set sizes of size 1, thus perfectly recovering the instruction, which was not possible in the native system.

However, even in WASM, some instructions are very similar to each other, e.g., instructions that require few

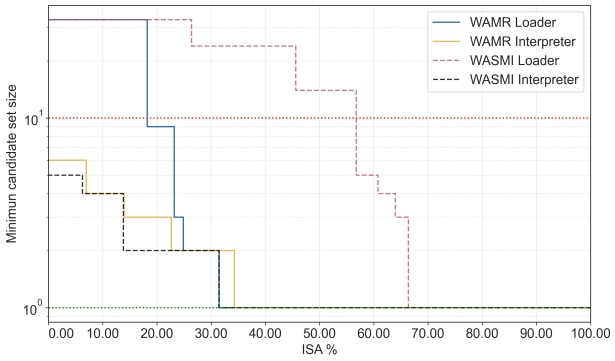

Figure 7. Candidate set size distribution of WASM instructions in the `WAMR` and `wasmi` translators under the SotA attacker. Only semantically different instructions are included in the candidate sets. The green dotted line is at $y = 1$, where candidate sets of that size offer no confidentiality.

x86 instructions to execute tend to still be challenging to classify accurately. For instance, in the `WAMR` interpreter, the `i32.add` and `i32.sub` instructions are both implemented with 9 x86 instructions and differ for a single one: `i32.add` uses an x86 `add`, wherein `i32.sub` has an x86 `sub`. Since the side channels available to the SotA attacker cannot distinguish between these two instructions, `i32.add` and `i32.sub` end up in the same candidate set[3]. We can also observe this in Table 2: instructions with a small number of IMs tend to have bigger candidate set sizes.

In summary, the WASM system leaks more instructions of the example loop compared to the native system, with 85% in `WAMR` and 80% (90% semantically different) in `wasmi` of its instructions being fully leaked, compared to 0% in the native system.

Full ISA. Similarly to the native system, we compute all possible candidate sets of the WASM system: if the candidate sets tend to be small for a large percentage of the WASM ISA, then the system itself cannot provide code confidentiality, as this attack will likely extend to different WASM binaries besides our sample program.

For this, we obtained the IMs of each WASM instruction while profiling a WASM test suite [37]. The test suite we used is developed to comply with the WASM standard and ensures we reach a good coverage for all the 172 core WASM instructions. We depict the distribution of the WASM instructions' candidate sets that we obtained for `WAMR` and `wasmi` in Fig. 7. As can be seen, for `WAMR` almost 80% of the ISA has a candidate set size ≤ 2, both in the loading and interpreting phase (which, in an actual attack, can be combined). The `wasmi` loader tends to be less leaky (only about 30% of the ISA is leaked perfectly), but its interpreter exhibits a similar leakage as the `WAMR` one. Comparing this to the native system, where even the ideal attacker could, at best, recover 10% of the ISA instructions, it is clear that the WASM system is leakier than the native system. Finally, not only is the WASM system leakier, but the results also highlight that a SotA attacker can practically break code confidentiality for at least 70% of the WASM ISA in both `WAMR` and `wasmi`.

3. Interestingly, the attacker can still distinguish these two instructions because they differ in multiple instructions in the loading phase.

6. IR Instruction Leakage in Practice

Despite the extent of the leakage of the WASM system, building an end-to-end instruction extraction attack is far from trivial. In this section, we describe what challenges emerge in building the attack and how we overcame them.

To introduce the tasks and challenges faced by the attacker, let us analyze an example IM trace that the attacker has extracted by single-stepping the enclave. Assume that we are given a trace of 184 IMs from the `WAMR` interpreter (like in the example used in Section 5.3). Recall that each IM contains the side-channel measurement of *one* executed x86 instruction. Say the attacker knows that the first 9 IMs relate to the first WASM instruction, and the next 14 IMs relate to the second WASM instruction that was parsed by the interpreter. Now the attacker can take the first 9 IMs and try to match them to one of the known WASM candidate sets. Then the same can be done with the next 14 IMs, and so on. While this approach is straightforward, it assumes that the attacker knows a priori how to *segment* a trace such that each segment contains the correct number of x86 instructions that the underlying (and unknown) WASM instruction used when executing. Note that each WASM instruction generally needs a different number of x86 instructions to execute, and additionally each instruction might need a variable number of x86 instructions depending on its input operands (because more or less processing is needed based on the input value). This creates a chicken and egg problem: if the attacker knew which confidential WASM instructions were executed, then they could use this information to make an educated guess on how to segment the trace of IMs. Each segment would then be classified into a WASM instruction using the candidate set information. However, the WASM instructions are not known beforehand – that is the information that the attacker is trying to extract in the first place.

To break this cyclic dependency, in an attack preparation phase, we extract more information than just the possible candidate sets. Particularly, we also collect: i) WASM instruction prefix and suffix markers, and ii) possible number of x86 instructions per WASM instruction. We leverage this extra information to segment the trace.

With this in mind, the attack is split in two phases: i) a preparation phase, which we name the *Profiling Phase*, and, ii) the *Attack Phase*. We depict them and how they interact with each other in Fig. 8. In the profiling phase, the attacker builds information about the WASM system, particularly they single-step the enclave in debug mode, feeding it with known WASM instructions to build the candidate set information and a database of patterns for each WASM instruction (this database is used to segment the trace). The profiling phase needs to be done once for each TEE and WASM runtime used (e.g., once for `WAMR` v1.1 in SGX, then again for `WAMR` v1.2 in SGX, etc). We detail this phase below in Section 6.1. In the attack phase,

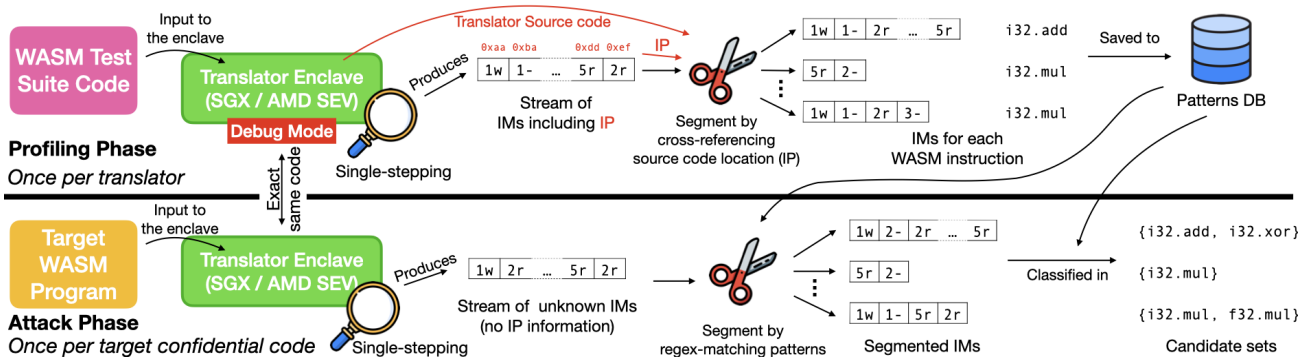

Figure 8. Overview of the end-to-end attack steps.

the attacker then single-steps the victim enclave executing in production mode while the confidential WASM program is being interpreted. In this phase, the attacker obtains a single trace of IMs that need to be segmented correctly before matching each segment with the previously profiled candidate sets. We describe this phase in Section 6.2.

6.1. Profiling Phase

In this phase, the attacker's goal is to profile the target translator and generate patterns of traces for each WASM instruction. We use these patterns to build candidate sets and to segment the IMs that will be collected during the attack phase. To fulfill this goal, the attacker collects IMs of known enclaves using the methodology described in Section 4. Additionally, in this phase, the attacker can control (and knows) the code and inputs of the enclave, and can run the enclave in debug mode. Debug mode allows the attacker to save the Instruction Pointer (IP) together with the IM.

As mentioned above, a different dataset needs to be built for each specific translator that will be used as public code (cf. Section 2) in the victim enclave. Meaning that at the start of the profiling phase, the attacker instantiates the specific translator of the victim, and profiles it from an attacker controlled enclave. To get as much information as possible, the attacker wants to execute a program that uses every WASM instruction and feeds diverse classes of inputs to each WASM instruction[4]. We use the official WASM test suite [37] for this purpose. This suite is maintained by the WebAssembly Working Group and is normally used to test the adherence of new compilers and interpreters to the WASM specification.

When we single-step the WASM test suite, multiple WASM instructions are executed for each test. Just like in a normal attack, we thus get a long trace of IMs for each test. However, segmenting the trace in this phase is feasible because: i) we know what WASM instructions are executed; ii) we have the IP information included in each IM. Hence, we can use the IP to automatically reference the translator source code, and trivially segment the IM trace.

In the attack phase, the attacker does not have access to the IP. Nevertheless, we empirically found that the following two pieces of information within IMs are sufficient to segment any trace perfectly for the tested translators: the code page number that was accessed; and whether the x86 instruction performed a memory read, a memory write, or no memory access. Given that this information is also sufficient for the rest of the attack, we represent each IM with a string composed of two parts: (i) the code page number; and (ii) the memory access type. For instance, `1r` represents the IM of an x86 instruction that was executed from page number `1` and made a memory read. Similarly, `1w` and `1-` refer to a memory write and to no memory access, respectively, from an instruction executed on page `1`. To represent a collected WASM segment, we concatenate the symbols for the various IMs belonging to that WASM instruction (according to the IP information). We refer to this string as the *pattern* for a particular WASM instruction. The main role of WASM instruction patterns is to leverage regular expressions to segment the instruction trace and to create candidate sets.

6.1.1. Profiling the translators. To leverage the IP information to segment a trace, we observe that certain marker IPs always appear in between two WASM instructions. These markers can then be used to segment in the profiling phase. We discuss these markers for both loading and interpreting in the following.

WAMR loader and wasmi. During its loading phase, `WAMR` loops through each instruction, as shown in Listing 1. `wasmi` works similarly both in the loader and interpreter (see `wasmi` code links [20]), with the exception that a function call is made on each loop iteration for the loader. Since each WASM instruction is loaded in one loop iteration, the beginning and the end of the loop is a good candidate to choose as segmentation marker. By manually inspecting the binary of the `WAMR` loader and the `wasmi` interpreter, we found the addresses of the first instruction of this loop and the first instruction outside of the loop. For the `wasmi` loader we use the address of the function entry point and return address, since it is a more precise

4. Feeding different input values is important because it allows to reach good coverage. The same WASM instruction might be executed by a different set of x86 instructions depending on what inputs are given to it, as we discuss in Section 6.2.

measuring point. Segmenting the WAMR loader and wasmi interpreter (and loader) execution trace is straightforward with knowledge of the IP: whenever we encounter the IP of the first loop instruction (function entry point) in the IM trace, we start a new segment. When we encounter the IP of the first instruction outside of the loop (function return address), we stop the segment generation. All the IM collected that do not belong to the loop (function) are assigned to a special null segment. This gives us a pattern for each loop iteration (function call). Then, since we know what WASM instructions are executed in the test suite, we can assign the correct WASM instruction to each segment. The quickest way to obtain the list of WASM instructions being executed, is to run a second, modified, version of the translator that print out which instruction is being parsed in each loop iteration (function call). The modified translator is only used to produce the list of executed WASM instructions and not to collect IMs, since then this translator would not exactly match the version used by the victim.

WAMR Interpreter. In the interpreter phase, the WAMR control flow is more involved than in the WAMR loading phase and wasmi. The interpreter executes one instruction, then fetches the pointer of the next instruction from memory and directly jumps to it – without performing any loop or function call. Crucially, every jump to the next WASM instruction is implemented as an indirect jump (e.g., jmp *rax). Thus, to automatically segment the execution trace of the interpreter, we look for indirect jumps in the execution trace. Since we have the IP for each IM, we can check on the WAMR interpreter source code whether the instruction at that IP is an indirect jump. Whenever we encounter an indirect jump, we start a new segment and add the previous one to our database[5]. Similarly to the loader, we need to label the segments: we again modified the WAMR interpreter to print the instruction label every time it starts interpreting a new instruction. As metioned above, this modified version is only used to produce the WASM instruction labels that are assigned to the segments.

The same profiling process can be used on other translators: fundamentally what the attacker needs is a way to find instructions boundaries based on the IP (either by manual inspection or automatically) and a way to map each segment to (known) IR (e.g., WASM) instructions labels.

6.1.2. Fused Instructions Handling. So far, the way we described the patterns for WASM instructions does not properly account for *fused instructions* from the CPU: separate x86 instructions that the CPU executes as one. The problem is that we observed that fused instruction pairs can sometimes still execute un-fused. Unfortunately, this leads to the pattern of WASM instructions being non-deterministic. We report an explanation of this phenomenon in Appendix D.

Theoretically, we could collect every possible variation of one WASM instruction, repeating a trace collection many times until we get all possible patterns. However, this approach is infeasible in practice for two reasons. First, since when interrupted the CPU non-deterministically fuses instructions, collecting all possible patterns requires many repetitions and is not guaranteed to terminate. Second, the number of different traces needed to be collected grows exponentially with the number of possible fused instruction pairs. Just 10 fused instructions pairs in a trace require 1024 patterns to be collected and stored.

We addressed this issue by detecting which IM could be related to fused instructions and then saving only the fused version of the pattern. This is done by cross-referencing the x86 instructions in the source code of the translators with the IP recorded for the IM. Then, alongside the pattern, we save an array of positions that could potentially be "unfused". This representation is not only compact (we need to save only one version of the pattern) but also allows us to match any combination of unfused instructions in the pattern efficiently.

5. This approach only works if indirect jumps are used only at the boundary between two instructions, as is the case in the WAMR interpreter.

6.2. Attack Phase

Trace Segmentation. In the attack phase, the adversary targets a production enclave which is given as input a confidential algorithm and single-steps it to obtain an execution trace. To segment the trace, as the IP is not available in this phase, the attacker needs to use the library of segments collected during the profiling phase. By representing the full trace as a string (in the same way as we reduced each segment to a pattern string while profiling), we can reduce the segmentation task to the well-known string-matching problem. Segmenting the trace then proceeds as follows. We try to find which of the known patterns match the beginning of the "trace string". We usually obtain multiple matches as some of the known patterns overlap. We then assume that any of the matched patterns is correct and remove it from the "trace string", this gives us the first tentative segment. If this segment is correct, we should be able to match other patterns at the start of the remaining "trace string". If nothing matches, we backtrack and choose one of the previously found valid patterns. We repeat this process until the whole execution trace is segmented – meaning an assignment to a segment is found for each IM. We will discuss the performance of this algorithm in Section 7.

Creating and Matching Patterns. The approach described above assumes that we can collect *every* possible pattern for each WASM instruction. Whilst the WASM test suite achieves wide coverage, some patterns needed to fully segment unseen binaries are still missing. In particular, while *linear* WASM instructions (instructions that have no loops or branching) exhibit only a single pattern, it is challenging to build every pattern for instructions with loops and branches. For instance, the WASM clz instruction is implemented in the WAMR interpreter with a loop that iterates once for every leading zero present in the input integer.

For cases of instructions with complex control flow, we leverage the observation that, generally, their "prefix" x86 instructions and their "suffix" x86 instructions will be the

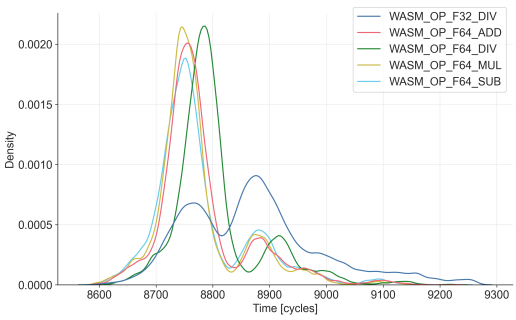

Figure 9. Timing distribution (N=11527) of the 5th x86 instruction of each of the five listed WASM instructions (`WAMR` interpreter phase). The two division ops seem to be following a different distribution than the others.

same, no matter how complex the internal control flow is. Thus, when we encounter more than one pattern for the same instruction, we automatically try to create a general pattern – a single pattern that will capture several variations of the instruction control-flow. We do this by arranging the characters of the string representation in a tree where each node of the tree is one *token* (code page number and access type). We then add multiple patterns to the same tree and extract the common prefixes from it. Particularly, after the tree is assembled, we traverse it and collect every pattern found up to 2-3 splits of the tree. We found this heuristic to be quite accurate in practice. We use the same process to find common prefixes and suffixes.

Between matching for common pre- and suffixes and accounting for variable numbers of instructions due to fused instructions, we found that the most convenient way to synthesize patterns was through regular expressions (*regexes*). This allowed us to use existing matching engines and rapidly prototype different matching configurations. We automatically generated regexes for each possible segment while also keeping the regexes' complexity within bounds.

Segment Classification. As discussed above, trace segmentation and segment classification are inherently linked tasks. Given a correct segmentation, we already get a possible list of candidate WASM instructions for each segment "for free": those are the instructions whose known patterns matched the segment. In fact, this is how we generated the candidate sets for WASM that we discussed in Section 5.3. Recall that segments are generated only using the code pages and the memory access type. Remarkably, this information alone is so accurate to not only segment the trace but also to correctly classify up to 80% of the WASM ISA.

Candidate Sets Pruning. We investigate whether we can further reduce the candidate set size for the remaining 20% of the WASM ISA where there is more than one candidate. In particular, IMs also contain the time spent executing individual x86 instructions, a feature we have not used so far in our attack, as it is not fully deterministic. We explore the potential of using time measurements to further prune candidate sets by using a concrete candidate set obtained from the `WAMR` interpreter, consisting of the following WASM instructions: `F32_DIV`, `F64_ADD`, `F64_DIV`, `F64_MUL`, and `F64_SUB`. Manual inspection of the interpreter's binary reveals that all of these WASM instructions are expanded into 9 x86 instructions. However, among these 9, only the fifth x86 instruction differs between the various WASM instructions. Therefore, any potential timing difference should be visible only in the 5th instruction[6]. The distribution of the recorded timings of the 5th x86 instruction is depicted in Fig. 9. While the timing distributions mostly overlap, they still exhibit some differences between them.

To demonstrate the significance of these timing differences, we developed a basic classical machine learning model that tries to classify between the aforementioned five WASM instructions using only the timing data. A simple random forest classifier [38] achieves around 45% accuracy, significantly outperforming a random guess (which has 20% accuracy). More details, including a confusion matrix, are given in Appendix C.

In summary, the candidate sets that we presented in Fig. 7 could be improved by including timing information. However, the attacker would have to record multiple executions for the same confidential algorithm to establish some confidence in the results. On the other hand, the information used when segmenting is deterministic, so the attacker only needs one execution of the confidential code to build the candidate sets that were presented in Fig. 7, and we thus deem the deterministic pipeline to be sufficient in practice.

7. Evaluation

We evaluated the methods and algorithms presented in Section 6 by using an Intel SGX enclave running the `WAMR` [19] runtime at commit version `b554a9d` and an enclave with a `wasmi` runtime at commit version `2ad9e4c`. To collect the patterns for each WASM instruction, we single-stepped the two translators while they executed the WASM test suite [37] (commit `e87021b`). We run only tests that do not test for exceptions, as we are interested only in correct programs, although it would be straightforward to also include these tests. We focus on SGX due to the availability of better tooling in this platform compared to SEV. While the time needed to execute the attack will differ between the two TEEs, the leakage numbers will be identical. We discuss why this is the case in Section 9.

Pattern Generation. Overall, we profiled 21073 tests for `WAMR` and 26315 for `wasmi`. Since `wasmi` does not fully support the same extensions of the WASM ISA as `WAMR`, not all the test suite test were compatible with it. To reach an adequate coverage for `wasmi` we complemented the test suite with the tests that are included in the `wasmi` repository. Note that we single-step the tests with the enclave in debug mode, as we need the IP to produce the segmentation patterns as discussed in Section 6.1. When we are profiling the `WAMR` loader, we only single-step the loader function (`wasm_loader_prepare_bytecode`).

6. We observed that surrounding instructions are also affected and exhibit timing differences, albeit smaller ones.

When we are profiling the WAMR interpreter, we single-step only the interpreter's main function (wasm_interp_call_func_bytecode). For the wasmi loader we used the Compiler::compile_instruction function, and for the interpreter the Interpreter::do_run_function function. By monitoring the program counter after the trace collection, we observed that we can very reliably single-step the enclave through interrupts, as no instruction was skipped for any of the tests in the test suite. Hence we run each test only once. In our machine (with an Intel i9-9900KS CPU), 36 hours for the WAMR and wasmi interpreter. It took about 24 hours for the WAMR loader, while we were able to shorten the time for the wasmi loader to 3 hours by executing only non-redundant tests. In total, we found 1576 (153) unique patterns for the WAMR (wasmi) loader and 345 (177) unique patterns for the WAMR (wasmi) interpreter. Using the methods described in Section 6.2, we then created 137 (57) regular expressions for the WAMR (wasmi) loader patterns and 133 (128) for the WAMR (wasmi) interpreter.

Open world attack. The SotA adversary can be instantiated in practice, and thus we performed our evaluation with real-world experiments. To test the generality and usefulness of the patterns, we used them to classify single WASM instructions in three synthetic programs. One of the programs is written in C and computes various cryptographic functions. The other two are written in Rust. One is part of a chess engine [21], while the other computes the hash of its inputs. We compiled these programs to WASM and then gave them as input to an initial enclave running WAMR. We single-stepped this enclave in production mode (i.e., without getting the IP information). Not all possible instruction patterns of these programs were present in the test suite. We verified this by naively trying to match the patterns we collected from the test suite and found that some parts of the trace could not be segmented. However, we were able to fully segment the trace using generalized regex patterns (cf. Section 6.2). The C code, the Rust chess code, and the hash code respectively executed $474M$, $431M$, and $62k$ WASM instructions. When loading the code, they parsed $9k$, $38k$, and $49k$ WASM instructions, respectively. Single-stepping the interpreter phase took around 10 hours for both the chess engine and the C code and a couple of seconds for the hash engine. Single-stepping the loading phase completed in less than 5 min. Roughly the same amount of time was required to segment the traces.

From the WAMR loading phase information, we perfectly recover 46%, 49%, and 50% of all the instructions in the C code, the Rust chess engine, and the Rust hash code, respectively. At least 65% of the instructions belong to a candidate set of size ≤ 3 in these three programs. Only looking at the WAMR interpreter phase, we recover around 28% of instructions with perfect information. These percentages are obtained from a single execution trace and without taking into consideration the execution time of the instructions.

Closed world attack. An application of the recovered WASM instruction traces is using it to classify which program or library is executing in the TEE among a fixed known set. For instance, this allows checking if a vulnerable version of a library is present in the confidential code supplied to the enclave. This is a useful building block for other attacks or could be used to check license agreements violations. Knowing that the confidential code belongs to a specific set of programs is an example of prior information (cf. Section 2.1) and therefore forms a closed world analysis. A closed world setting might occur in different ways, to name two: the attacker might know that the victim is linking against a set of (public) pre-compiled binaries; or the exact compiler version and compilation flags used by the victim.

IR execution is particularly vulnerable to this classification task compared to native execution. This is because in native execution the attacker can only measure the instructions from the executed code paths. This implies that the attacker would need to either know the input of the enclave or have a trace for every possible code path of the target function/library, which is being checked for presence in the enclave. On IR execution, on the other hand, the loading phase is particularly well suited to match known segments of code. This is because, generally, instructions are parsed sequentially and in the same order across executions, no matter what other inputs are provided to the enclave. Additionally, functions are also parsed independently in the WAMR loader, allowing the attacker to even check for individual matching function signatures of a library.

We note that smaller functions are generally harder to classify than larger ones (where it might be sufficient to just match with 100% confidence a couple of marker instructions in them). We thus tested several small functions by trying to match their presence in a larger library. We took a part of the Go Ethereum implementation [39] that supports compilation to WASM and used it as a test library. We copied the implementation of 10 individual arithmetic functions (responsible for handling big number operations) of this project and used them in smaller programs. These smaller programs simply contain a main function that calls the copied library functions. We then collected a trace of the loading phase of these small programs and segmented their WASM instructions. Finally, we tried to match the traces of the individual functions into a trace of the loading phase of the library in WAMR. We were able to perfectly match the smaller functions in the trace of the library, thus demonstrating that closed-world function classification is practical in the WASM system and can be used to classify which confidential WASM program is running in the enclave.

8. Related Work

In this section, we discuss which side-channel attacks on TEEs we build upon and how they influence the information we assume the attacker gets access to (c.f., Section 4). Note that generally, these side channels are developed to leak data from enclaves given the knowledge of the source code. However, in our setting, we need to adapt them to work without any prior knowledge of the source code.

Stack and Memory Access. Page table-based attacks on Intel SGX exploit the untrusted OS role in managing the

page tables for enclaves [22]. The page faulting mechanism can be abused [40] to notify the attacker through page faults of enclave code and data accesses. Similarly, the access and dirty bits of the page table entries can be used to monitor read and writes [26, 27] accesses performed by the enclave. Monitoring these bits while single-stepping gives the attacker a per-instruction resolution of these values. Moreover, the attacker can also detect control-flow changes if the instruction jumps/branches to another page. Note that these attacks are completely deterministic and noise-free.

Microarchitectural Structures. Additional information can be extracted from the numerous microarchitectural details made available to the OS. While performance counters are not updated in enclave mode, their values, as measured from an attacker-controlled program, can still be influenced by the enclave execution. It is also worth mentioning that the last branch record (LBR), given knowledge of the location and target of jumps in an enclave, can be used to test for branching conditions [28]. It is feasible to extract the LBR given the knowledge of the code, but it is challenging to employ this side channel in our setting given that we do not know a priori the address of the jumping instructions in the confidential code.

Instruction Timing. Instruction timing is considerably noisier than any previously described attack. To estimate the best resolution available to the attacker, we describe how related work leaks data from enclaves despite the noisy measurements. Nemesis [25] observed that while single-stepping via interrupts, the interrupt delivery time is dependent on the instruction executed by the enclave. Usually, the attacks that leverage these timing measurements [36, 25, 41] perform multiple thousands of measurements for a single instruction to reduce the noise. We note that repeating measurements is not trivial and either requires the attacker's capability of re-running the enclave arbitrarily [25] or specific instructions before the measurement to launch a microarchitectural replay attack [41]. Even with the ability to repeat measurements, these attacks have a resolution of $40 - 100$ cycles.

Port Contention. The final source of information we consider is related to monitoring CPU port contention. Several attacks have demonstrated that this is a practical side-channel attack [42, 43]. However, they require repeated experiments to extract a signal from their noisy measurements. Nevertheless, we assume complete knowledge of the exact functional units used in the ideal attacker in Section 5.

Summary. We chose to give the SotA attacker an even better timing resolution than what is currently feasible by allowing them a 10 cycles resolution from a single run. Note that we also study an ideal attacker which, among other things, is cycle accurate and can perfectly monitor the CPU port utilization. As discussed in Section 7, despite these capabilities, both attacker models leak very little information from the native system. On the other hand, using only controlled-channel information is enough in the WASM system to leak the vast majority of the ISA, highlighting the magnitude of the leakage amplification between the two systems.

9. Discussion

Applicability to other TEEs. To execute the attack we need the following primitives (cf. Section 6.1): i) single-stepping each instruction, ii) getting page-level memory information for each instruction, including which code page was accessed and whether the instruction performed a read, write, or no access. As long as an attack makes this information available on the target TEE, and if the same translator code is being targeted, the leakage will be *exactly* the same as depicted in Fig. 7. In fact, the translator can even (and will) slightly differ: the WASI interface is generally implemented differently on AMD SEV compared to SGX. Crucially, however, the code we targeted to compute the candidate sets does not include the WASI interface, so the leakage numbers provided are not affected by this difference.

Concurrently an attack framework has been developed for AMD SEV, dubbed SEV-Step [30], that satisfies all of the requirements mentioned above. With SEV-Step it has been demonstrated that the guest VM OS events can be suppressed and filtered out to target only a victim application running on the guest. Thus, as long as the same `WAMR` and `wasmi` code as SGX is used in SEV, the results on leakage presented in Section 6 *apply 1:1* to SEV as well. What changes between the two systems is how long it takes to run the end-to-end attack (due to hardware differences). Thus, the duration of the attack presented in Section 7 will look different on SEV.

Finally, other frameworks have recently emerged for other TEEs allowing the attacker to gather similar side-channel information. For instance, TDX-Step [44] for Intel TDX, and Load-Step [45] and Cachegrab [46] for ARM Trustzone.

Detectability of the attack. As discussed in Section 7, single-stepping the enclave execution significantly delays the normal execution of the confidential code. While this might seem a good way to detect that such an attack is taking place, our attack on WASM requires a single victim run. Since the cloud provider is malicious (cf. Section 2.1), they can drop network packets after the confidential code has been loaded. To the victim this would appear as if the guest VM/SGX enclave is unreachable (which can happen even without an attack). Thus detection alone is not a viable defense, and rather the approaches that we discuss next should be considered in a production deployment.

Defenses. For SGX, an effective defense against our attack is AEX-Notify [47] which prevents us from reliably single-stepping the enclave and hence building a complete trace of IM. Without a complete trace we cannot create segments, and without segments we cannot classify WASM instructions. We recommend any project that aims to achieve code confidentiality in SGX to apply this defense. To the best of our knowledge, no similar defense yet exists for SEV.

10. Conclusions

We studied two different approaches commonly used for deploying confidential code into TEEs – deploying native binaries and intermediate representation (IR) – against state-of-the-art side-channel attacks. We developed a novel methodology to analyze the side-channel leakage of these approaches. We experimentally validated our methodology on nine modern microarchitectures and showed that IR-based confidential code deployments amplify any leakage found in native execution deployments. We showed that in an open-world setting (without prior knowledge of the confidential instructions) native execution results in limited leakage even against an ideal attacker, while next to no code confidentiality against a state-of-the-art attacker was present when using WASM as an IR. While IR execution is already unsafe even without adding prior knowledge capabilities, we leave to future work the task of investigating whether native execution still holds enough confidentiality guarantees in this setting.

Code availability. The source code developed for this paper is available at the following link: https://github.com/dn0sar/TEE-WASM-Code-Extraction

References

[1] I. Corporation. *Intel Software Guard Extensions*. https://software.intel.com/en-us/sgx.

[2] Advanced Micro Devices Inc. *AMD Secure Encrypted Virtualization (SEV)*. https://developer.amd.com/sev/. Accessed: January 2020.

[3] *Confidential Computing concepts — Confidential VM*. https://cloud.google.com/compute/confidential-vm/docs/about-cvm. Accessed: August 2022.

[4] *Microsoft Docs: Build with SGX enclaves - Azure Virtual Machines*. https://docs.microsoft.com/en-us/azure/confidential-computing/confidential-computing-enclaves. Accessed: August 2022.

[5] *Azure Confidential VM options on AMD*. https://docs.microsoft.com/en-us/azure/confidential-computing/virtual-machine-solutions-amd. Accessed: August 2022.

[6] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. "VC3: Trustworthy Data Analytics in the Cloud Using SGX". *2015 IEEE Symposium on Security and Privacy*.

[7] E. Bauman, H. Wang, M. Zhang, and Z. Lin. "SGXElide: Enabling Enclave Code Secrecy via Self-Modification". *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO 2018)*.

[8] T. Lazard, J. Götzfried, T. Müller, G. Santinelli, and V. Lefebvre. "TEEshift: Protecting Code Confidentiality by Selectively Shifting Functions into TEEs". *Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18)*.

[9] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee. "OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX". *Proceedings 2019 Network and Distributed System Security Symposium (NDSS '19)*.

[10] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni. "Twine: An Embedded Trusted Runtime for WebAssembly". *2021 IEEE 37th International Conference on Data Engineering (ICDE)*.

[11] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza. "TrustJS: Trusted Client-Side Execution of JavaScript". *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*.

[12] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. "AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting". *Proceedings of the 20th International Middleware Conference (Middleware '19)*.

[13] A. Baumann, M. Peinado, and G. Hunt. "Shielding Applications from an Untrusted Cloud with Haven". *ACM Trans. Comput. Syst.* (2015).

[14] *Enarx: WebAssembly + Confidential Computing*. https://enarx.dev.

[15] M. Brossard, G. Bryant, B. El Gaabouri, X. Fan, A. Ferreira, E. Grimley-Evans, C. Haster, E. Johnson, D. Miller, F. Mo, D. P. Mulligan, N. Spinale, E. van Hensbergen, H. J. M. Vincent, and S. Xiong. *Private delegated computations using strong isolation*. Technical report. 2022.

[16] *Edgeless Systems: Confidential computing at scale for everyone*. https://www.edgeless.systems.

[17] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. "SCONE: Secure Linux Containers with Intel SGX". *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[18] *Awesome WebAssembly Languages*. https://github.com/appcypher/awesome-wasm-langs. Accessed: August 2022.

[19] *Wasmer - The Universal WebAssembly Runtime*. https://github.com/bytecodealliance/wasm-micro-runtime. Target loader code leading to leakage amplification: https://github.com/bytecodealliance/wasm-micro-runtime/blob/b554a9d05d89bb4ef28068b4ae4d0ee6c99bc9db/core/iwasm/interpreter/wasm_loader.c#L6137 . Target interpreter code leading to leakage amplification: https://github.com/bytecodealliance/wasm-micro-runtime/blob/b554a9d05d89bb4ef28068b4ae4d0ee6c99bc9db/core/iwasm/interpreter/wasm_interp_fast.c#L996.

[20] *Wasmi - WebAssembly (Wasm) Interpreter*. https://docs.rs/wasmi/latest/wasmi/. Target loader code leading to leakage amplification: https://github.com/paritytech/wasmi/blob/01423af0caebcd201542d2f5333ba037e85c419f/crates/wasmi/src/engine/executor.rs#L186.

[21] *Perft Test Benchmarks for crates.io/chess/, crates.io/shakmaty/*. https://crates.io/crates/chess. Version: 3.1.1.

[22] V. Costan and S. Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086.

[23] Advanced Micro Devices Inc. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed: May 2022.

[24] J. Van Bulck, F. Piessens, and R. Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". *Proceedings of the 2nd Workshop on System Software for Trusted Execution (SysTEX'17)*.

[25] J. Van Bulck, F. Piessens, and R. Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.

[26] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. "Telling Your Secrets without Page Faults: Stealthy

Page Table-Based Attacks on Enclaved Execution". *26th USENIX Security Symposium (USENIX Security '17)*.

[27] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*.

[28] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". *26th USENIX Security Symposium (USENIX Security '17)*.

[29] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. "CopyCat: Controlled Instruction-Level Attacks on Enclaves". *29th USENIX Security Symposium (USENIX Security '20)*.

[30] L. Wilke, J. Wichelmann, A. Rabich, and T. Eisenbarth. *SEV-Step: A Single-Stepping Framework for AMD-SEV*. 2023. arXiv: 2307.14757 [cs.CR].

[31] *Wasmtime: A standalone runtime for WebAssembly*. https://github.com/bytecodealliance/wasmtime. Target loader code leading to leakage amplification: https://github.com/bytecodealliance/wasmtime/blob/1bf0c8c220e37c91cbe0946d944bdb6f0c13e35c/cranelift/wasm/src/code_translator.rs#L118.

[32] *Wasmer - The Universal WebAssembly Runtime*. https://wasmer.io/. Target loader code leading to leakage amplification: https://github.com/wasmerio/wasmer/blob/dce55432e64f5d4d6b2b21b523aba2e8b1149b4a/lib/compiler-llvm/src/translator/code.rs#L1416.

[33] *Wasm Edge Runtime*. https://wasmedge.org/. Target loader code leading to leakage amplification: https://github.com/WasmEdge/WasmEdge/blob/9f5408746a76dc345c75f4780a357951992567ce/lib/executor/engine/engine.cpp#L88.

[34] *Wasm3 - A fast WebAssembly interpreter, and the most universal WASM runtime*. https://wasmedge.org/. Target loader code leading to leakage amplification: https://github.com/wasm3/wasm3/blob/49290f19ff48c7aa59630a3f6006a9305f13ee02/source/m3_compile.c#L2552, Jump table definition: https://github.com/wasm3/wasm3/blob/49290f19ff48c7aa59630a3f6006a9305f13ee02/source/m3_compile.c#L2257.

[35] A. Abel and J. Reineke. "uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures". *ASPLOS (ASPLOS '19)*.

[36] I. Puddu, M. Schneider, M. Haller, and S. Capkun. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". *30th USENIX Security Symposium (USENIX Security 21)*.

[37] *Official WebAssembly test suite*. https://github.com/WebAssembly/spec/tree/main/test.

[38] T. K. Ho. "Random Decision Forests". *Proceedings of 3rd International Conference on Document Analysis and Recognition*. IEEE.

[39] *Official Go implementation of the Ethereum protocol*. https://github.com/ethereum/go-ethereum. Version 1.10.24.

[40] Y. Xu, W. Cui, and M. Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". *2015 IEEE Symposium on Security and Privacy*. IEEE.

[41] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher. "MicroScope: Enabling Microarchitectural Replay Attacks". *IEEE Micro* (2020).

[42] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri. "Port Contention for Fun and Profit". *2019 IEEE Symposium on Security and Privacy (SP)*.

[43] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi. "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures". *Proceedings 2020 Network and Distributed System Security Symposium (NDSS)*.

[44] Intel Corporation. *Intel Trust Domain Extension Research and Assurance*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/tdx-security-research-and-assurance.html. Version 1.0, 24 Apr 2023.

[45] Z. Kou, W. He, S. Sinha, and W. Zhang. "Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict". *2021 58th ACM/IEEE Design Automation Conference (DAC)*.

[46] K. Ryan. "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone". *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*.

[47] S. Constable, J. V. Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein. "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves". *32nd USENIX Security Symposium (USENIX Security 23)*.

[48] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*.

[49] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual. Volume 2: System Programming*.

Appendix A. Responsible Disclosure

On 2 November 2022, we disclosed our findings to the following companies promising code confidentiality in TEEs: Veracruz (ARM) [15], Edgeless [16], Enarx [14], and Scone [17]. Edgeless acknowledged receiving our report but did not take any further steps. Enarx responded that they are researching mitigations, while Scone told us they are working on mitigating the reported issues. Veracruz responded that side channels are out of scope in their attacker model. Nonetheless, they are working on clarifying their documentation about the risks related to code confidentiality in TEEs.

Appendix B. x86 ISA Instruction Count

We focus only on the 64-bit version of the x86 architecture when creating candidate sets. In building the candidate sets for the microarchitectures supporting SGX and SEV, we need to account for the fact that some instructions are handled differently in these environments. Particularly, in SGX, some of the instructions are illegal and thus will never be called on bug-free enclaves. On SEV, all instructions are allowed to execute; however, they will cause a hypervisor intercept, thus leaking to the attacker which instruction was executed. In the case of SGX, we never include illegal instructions in a candidate set, while in the case of SEV, we place the intercepted instructions in candidate sets of

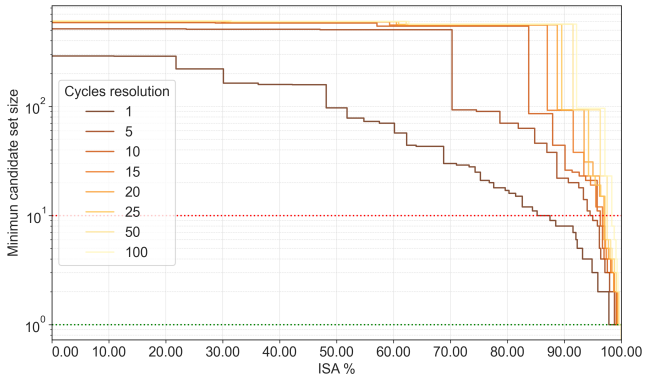

Figure 10. Candidate set sizes' distributions on the Skylake microarchitecture for a SotA attacker in the native system with varying cycle accuracy thresholds.

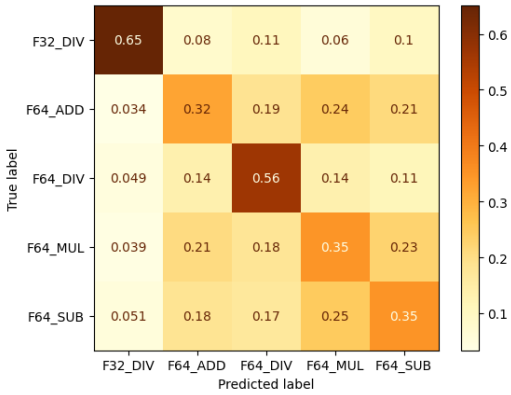

Figure 11. Confusion matrix of a simple random forest classifier for five WASM instructions. The classifier is pretty confident about the two divisions but cannot distinguish the other 3 instructions.

size 1. Next, we detail what instructions exactly end up in this special classification for the two TEEs.

SGX. We used the information from the Intel SDM Manual [48] Volume 3D Table 35-1 to find the criteria for instructions not allowed in SGX. To summarize, instructions with a privilege level lower than 3 and instructions that perform I/O operations or that could access the segment register are considered Illegal. Note that an instruction could have an illegal version and a legal version. For instance, the `mov` instruction can write to the segment registers, and that version of the instruction is illegal.

SEV. Instructions that cause a hypervisor intercept on SEV are reported in "Table 15-7. Instruction Intercepts" of the AMD64 Architecture Programmer's Manual [49]. Note that there might be other conditions that cause intercepts, which might leak information to the attacker, but we only consider the instructions on that table in our calculation. Finally, the dataset we used for the Zen microarchitecture was actually obtained from information collected from a Zen+ CPU from uops.info [35]. The Zen+ and Zen microarchitectures support the exact same x86 instructions; however, the Zen+ does not provide support for SEV.

Appendix C. Analysis of SotA Attacker Cycle Accuracy

To give an idea of the relationship between the strength of the SotA attacker's instruction cycle resolution and the native system information leakage, we show in Fig. 10 how the candidate set sizes change with different thresholds for the attacker's cycle resolution. Furthermore, we report the confusion matrix from a random forest classifier (as mentioned in Section 6.2) in Fig. 11.

Appendix D. Unfused instructions

When single-stepping with interrupts, fused instruction pairs are generally stepped through atomically – thus, we will only encounter one IM in the execution trace instead of two. This phenomenon has also been documented in previous work [36, 29]. However, while previous work observed deterministic instruction fusion [29], we observed that for the same pair of instructions in the program (at the same virtual address), it could happen that the instructions sometimes execute unfused. This is the case even when the same input data is given to the program and both with and without hyperthreading enabled. We hypothesize that this behavior is due to the precise timing at which the interrupt is delivered in relation to the stage of the execution of the to-be-fused instruction pair. However, the timing at which the interrupt is delivered cannot be controlled to such precision, and therefore the behavior randomly occurs, albeit somewhat infrequently. Note that we collect significantly larger instruction traces compared to [29] (e.g., some traces we obtain have more than 1 billion instructions) and hence have a higher likelihood of observing this behavior compared to [29].

Appendix E. Utility and applicability of the attack

Our study considered an attacker with the goal of recovering the ISA instructions of the confidential algorithm, i.e., the opcodes. These results can be used in different ways: we now discuss some possible practical attacks that leverage such data.

Reverse-Engineering Algorithms. A reverse engineer that wants to understand what the confidential algorithm does can leverage our results on semantically equivalent instructions (see Section 4) to further reduce the number of candidate instructions and reconstruct the logic of the algorithm. Note that our attacker only leaks the instructions but not

their *operands*. However, in a language like WASM, this is irrelevant for most instructions since their operands are implicit. For instance, an addition in WASM implicitly operates on the last two values present on the stack. Thus leaking that an addition was performed is enough to also leak the operands in this case. Note, however, that even in WASM, some instructions take constant values as parameters. These instructions can move values around on the stack based on their operand. We leave the task of leaking the operands for these instructions as future work.

Applicability to Other Languages. For our evaluation, we chose WASM as the language to instantiate the particular IR execution that we studied (cf. Section 3). However, some code confidentiality designs in TEE (e.g., Scone [17]) also support different interpreted languages, e.g., Python and NodeJS. The methods we introduced in this paper can easily be applied to analyze how much the translators of these other languages amplify the instruction leakage. As far as we are aware, their translators are not designed to provide code confidentiality, so we expect them to exhibit similar levels of leakage.

Appendix F. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

The authors study to what extent confidentiality protections for code executing within trusted execution environments (TEEs) are preserved in the face of side-channel attacks that leverage single-step execution. The authors analyze the potential leakage for both native and intermediate representation (IR) instructions and develop a practical attack for WebAssembly (WASM) IR, demonstrating that IR code is significantly more vulnerable to side-channel attacks than native code and quantifying the leakage.

F.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

F.3. Reasons for Acceptance

1) Important to understand the resilience of TEEs to side-channel attacks against code confidentiality
2) Development and execution of a practical attack against the WAMR and wasmi runtimes on Intel SGX
3) Useful study that rigorously quantifies the level of leakage for both native and IR (WASM) code