Guardian: Symbolic Validation of Orderliness in SGX Enclaves

Pedro Antonino
pedro@tbtl.com
The Blockhouse Technology Limited
Oxford, UK

Wojciech Aleksander Wołoszyn
wojciech@tbtl.com
The Blockhouse Technology Limited
Mathematical Institute, University of Oxford
St Hilda's College
Oxford, UK

A. W. Roscoe
awroscoe@gmail.com
The Blockhouse Technology Limited
University College Oxford Blockchain Research Centre
Department of Computer Science, University of Oxford
Oxford, UK

ABSTRACT

Modern processors can offer hardware primitives that allow a process to run in isolation. These primitives implement a trusted execution environment (TEE) in which a program can run such that the integrity and confidentiality of its execution are guaranteed. Intel's Software Guard eXtensions (SGX) is an example of such primitives and its isolated processes are called enclaves. These guarantees, however, can be easily thwarted if the enclave has not been properly designed. Its interface with the untrusted software stack is a perhaps the largest attack surface that adversaries can exploit; unintended interactions with untrusted code can expose the enclave to memory corruption attacks, for instance.

In this paper, we propose a notion of an orderly enclave which splits its behaviour into the following execution phases: *entry*, *secure*, *ocall*, and *exit*. Each of them imposes a set of restrictions that enforce a particular policy of access to untrusted memory and, in some cases, sanitisation conditions. A violation of these policies and conditions might indicate an undesired interaction with untrusted data/code or a lack of sanitisation, both of which can be harnessed to perpetrate attacks against the enclave. We also introduce Guardian: a tool that uses symbolic execution to carry out the validation of an enclave against our notion of an orderly enclave; in this process, it also looks for some other typical attack primitives. We discuss how our approach can prevent and flag enclave vulnerabilities that have been identified in the literature. Moreover, we have evaluated how our approach fares in the analysis of some enclave samples. In this process, Guardian identified some security issues previously undetected in some of these samples that were acknowledged and fixed by the corresponding maintainers.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**; **Software security engineering**.

CCSW '21, November 15, 2021, Virtual Event, Republic of Korea.

ACM ISBN 978-1-4503-8653-1/21/11…$15.00
https://doi.org/10.1145/3474123.3486755

KEYWORDS

SGX; enclave; orderliness; TEE; Trusted Execution Environment, symbolic execution; memory corruption

ACM Reference Format:
Pedro Antonino, Wojciech Aleksander Wołoszyn, and A. W. Roscoe. 2021. Guardian: Symbolic Validation of Orderliness in SGX Enclaves. In *Proceedings of the 2021 Cloud Computing Security Workshop (CCSW '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3474123.3486755

1 INTRODUCTION

Some modern processors offer hardware primitives that implement trusted execution environments (TEEs) [32]. When (part of) a process is run inside a TEE, its execution's integrity and confidentiality are preserved. The use of a TEE essentially splits computing resources into two categories: trusted and untrusted. While trusted resources are managed by the TEE and, therefore, enjoy its protection and guarantees, untrusted resources lie outside its trust boundary. Intel's TEE implementation, called Software Guard eXtensions (SGX) [15, 20], allows a process to have a protected memory range that is called an *enclave*. Enclaves are isolated from other applications, the operating system, hypervisors, and even other enclaves. The protection offered by TEEs, however, can be thwarted by a careless enclave design and implementation. The interface between enclave and the untrusted software stack, especially, has to be very carefully designed to prevent attacks. There is little point in running a program inside a TEE if the untrusted software stack can manipulate its execution and breach its guarantees. As enclaves are being rapidly adopted [3, 13, 40, 43, 63], frameworks to analyse and improve on their security are increasingly needed.

We propose the notion of an *orderly enclave* which splits the execution of enclaves into phases: *entry*, *secure*, *ocall*, and *exit*, each of which imposes a set of restrictions on the behaviour of the enclave in the form of untrusted-memory-access policies and sanitisation conditions. For instance, in the *entry* phase—which corresponds to the point when the untrusted host application has started the enclave and handed control over to it—we require CPU registers, which are shared between the untrusted hosted application and enclave, to be sanitised before the *secure* phase starts to prevent untrusted register values from being used in *secure* computations. Violations to these policies and conditions represent flaws in the design or implementation of the enclave, or even underlying *vulnerabilities* which might be used to breach enclave's guarantees. Our *novel* concept of orderliness focus on the analysis of the *new* *enclave-specific attack surface* emerging from the interaction between trusted and untrusted code and data. Therefore, it operates in a significantly different way than similar analysis tools which simply examine enclaves looking for memory-corruption vulnerabilities that are typically found in traditional programs.

We also propose *Guardian*: an open-source tool that uses symbolic execution to validate whether an enclave binary is orderly—it also tries to detect typical memory-corruption vulnerabilities in the process. Symbolic execution approaches are very useful for the analysis of programs, especially when targeting intermediate languages or bytecode [1, 8, 30, 33, 38, 48]. They analyse the execution of a program based on symbolic representatives of its variables. This symbolic exploration tends to cover a large number of code paths, making it an effective strategy for finding bugs and vulnerabilities in programs. We use *angr* [48]—a symbolic execution engine that focuses on usability—as a backend for our tool. The developer of the enclave can use our tool in an iterative process where it validates design and implementation, fixing violations until no more of them are found.

Our contributions are as follows. The definition of an orderly enclave which captures a helpful discipline for the developer with respect to interactions between enclave and the untrusted software stack. Unlike similar approaches that merely apply traditional techniques to the analysis of enclave code, we explores the new enclave-specific attack surface emerging from the interactions between trusted and untrusted worlds. We also give a practical overview of how our framework should be applied to typical Intel SGX SDK enclaves, detailing how the SDK meets the restrictions imposed by each execution phase of an orderly enclave. Furthermore, we propose an open-source tool that uses symbolic execution to validate an enclave binary against our notion of an orderly enclave, while also identifying common memory-corruption vulnerabilities in the process. Our definition and tool was inspired by [56]. While that paper manually identifies a number of enclave vulnerabilities, our notion of an orderly enclave and Guardian *automatically* identifies such vulnerabilities or their effects. We present a practical evaluation of our framework where we show how it fares in analysing some real enclaves. When analysing these enclaves, including one maintained by Intel, Guardian uncovered *previously-unknown security issues* which were promptly acknowledged and fixed by the corresponding maintainers; these issues were reported as *violations of our orderly enclave property* - specifically of the access policies they impose - as opposed to typical memory-corruption vulnerabilities. These results demonstrate that our framework, and especially our definition of orderliness, and tool complement similar state-of-the-art approaches.

Outline. Section 2 examines related work, and Section 3 introduces the concepts related to Intel SGX, symbolic execution, and *angr*. Section 4 presents our framework, discusses how it identifies some typical enclave vulnerabilities, and presents an evaluation on how it fares when applied to some real enclaves. We present our concluding remarks in Section 5.

2 RELATED WORK

The identification of enclave vulnerabilities has been the focus of many recent papers. In this section, we focus on the work that we believe to be most closely related to ours.

Lee at al. present the first memory-corruption attack against Intel SGX enclaves in [28]. The *Dark-ROP* method finds attack gadgets in the enclave's code, which are later explored to execute malicious code that can both exfiltrate enclave's derived keys and generate attacker-controlled reports. Biodo et al. [6] improve on this initial attack technique by proposing two new attacks that can be carried out under weaker adversarial assumptions; it works for enclaves with a randomised memory layout (using, for instance, SGX-Shield [45]) and do not require kernel privileges. These papers highlight the need for tools such as Guardian to detect vulnerable memory manipulations.

Bulck at al. [56] manually analyse runtime libraries for several TEE implementations and how poorly designed interfaces between trusted and untrusted code can lead to vulnerabilities. It considers Intel SGX, Keystone [27], and Sanctus [35], as far as TEE implementations, and runtime libraries including Intel SGX SDK and Open Enclave SDK [37]. It identifies a number of exploitable vulnerabilities that can be used to perpetrate powerful attacks against enclaves. That work [56] inspired our paper. While that paper identifies a number of vulnerabilities manually, Guardian implements machine-testable conditions described in our notion of an *orderly enclave*, giving rise to a framework that automatically looks for such vulnerabilities.

TeeRex [14] is a tool that relies on symbolic execution, using *angr*, to find memory corruption vulnerabilities in Intel SGX enclaves. TeeRex is arguably the work that is closest to ours. Both approaches use symbolic execution to look for vulnerabilities, but they are based on fundamentally different ideas. While TeeRex focuses on finding typical exploit primitives within enclaves, we look for violations of policies to access untrusted memory and sanitisation conditions, both of which are part of our *orderly enclave* definition. Moreover, we analyse the trusted execution spanning from runtime library to user code, whereas TeeRex focuses on user-defined code. It has been shown that runtime-library code can also be flawed [56]. Anecdotally, the binaries analysed by TeeRex in [14] are built with an Intel SDK version that exhibits register sanitisation violations, typically afflicting runtime-library code. Our framework is able to identify such vulnerabilities whereas TeeRex is not. Our approach requires a more detailed input from the user in the form of annotations of specific program points, whereas TeeRex only needs addresses of some specific enclave functions. Both Guardian and TeeRex have heuristics to automatically identify these addresses but TeeRex does not use debug symbols and so it can be applied to closed source binaries, unlike Guardian. In Section 4.4, we present violations to our orderly-enclave definitions that were detected by our framework in some practical enclaves but which went undetected when analysed by TeeRex. They have been acknowledged and fixed by the corresponding enclave maintainers.

Formalisation and verification of isolated processes have been studied in recent papers [50–52]. Sinha et al. [51] propose *Moat*: an approach to formally verify confidentiality for Intel SGX enclaves.

It formalise x86 instructions using simple micro-architectural instructions lifted by BAP [7] which are encoded in the Boogie language [29]. The notion of enclave confidentiality proposed is a hyper-property constraining secret-dependant enclave writes to untrusted memory; enclave secrets are passed as an input to Moat. The Boogie verifier [5] is then used to check this hyper-property. Moat scalability issues are acknowleged in a follow-up paper [50] which improves on Moat by proposing a methodology to implement and verify isolated applications, such as enclaves, designed following a specific confidentiality policy: the application can only write to untrusted memory via an encrypted channel. This methodology decomposes application into two modules: user code and a small runtime library, which helps to also decompose the confidentiality verification and its scalability. This work also proposes a modular verifier to carry out part of this verification. Subramanyan et al. [52] propose the notion of a trusted abstract platform and of the secure remote execution property. They attempt to generically and formally capture the concept of a TEE and what means to securely execute within it, respectively. The confidentiality properties analysed by these works are candidates to be translated into future versions of our framework, possibly leading to a fine-grained confidentiality analysis.

SGXBounds [26] is a framework that instruments enclaves with extra boundary metadata about different memory segments (heap, stack, global objects) so that out-of-boundary violations can be identified and handled at runtime. It adapts ideas of memory-safety-hardening paradigms for traditional programs to SGX enclaves [36, 46]. Even though both this work and our own are concerned with some form of boundary checking, they differ radically in intent and approach. While SGXBounds detects typical memory-corruption attacks that take place within the enclave, such as buffer-overflows involving the trusted stack and heap, our framework tries to identify vulnerabilities in the new attack surface arising from the interactions between trusted and untrusted world.

Many side-channel attacks against enclaves have been identified in recent papers [2, 11, 12, 17, 34, 41, 55, 57–60]; SGX has not been designed to prevent them [20]. There are attacks using various kinds of side-channel oracles—power-based, timing-based, cache-based, FPU-based, etc.—to exfiltrate secrets and influence the behaviour of enclaves. Symbolic execution can be harnessed to identify also side-channel oracle primitives. For instance, alignment faults can be harnessed to create a side-chain oracle [56], and by enforcing that the AC flag is cleared upon enclave entry, Guardian can identify whether the enclave's code exhibits the behaviour necessary to create such an oracle.

3 BACKGROUND

This section introduces the concepts about TEEs and symbolic execution necessary to make our paper self-contained.

3.1 Intel SGX

Intel's TEE implementation, called Software Guard eXtensions (SGX) [15, 19], allows an untrusted host process to define a range in its virtual memory where integrity-protected and confidential code and data are hosted; this isolated memory is called an *enclave*. SGX extends Intel's traditional instruction set with privileged instructions[1]—such as ecreate, eadd, eextent, einit and eremove—to create, initialise, and dispose of this protected memory range. The confidentiality, authenticity, and freshness of the memory in this range is guaranteed by a combination of SGX privileged instructions, which protect swapped pages, and Intel SGX's Memory Encryption Engine [18], which protect pages in physical memory. SGX also introduces unprivileged/user instructions[2] that can be used to execute enclave code. User code can enter and leave the enclave's code using instructions eenter and eexit, respectively. Aside from these synchronous enclave transitions, the enclave can also be interrupted and resumed.

Built upon these hardware primitives, Intel offers an SDK to help developers create enclaves using C/C++ [20–22]. The enclave programming model relies on two key concepts: an *ecall* is a mechanism by which untrusted code can invoke enclave code, whereas an *ocall* is used to call untrusted code from the enclave. The ecall abstraction is implemented by three functions: an *untrusted proxy function*, a *trusted bridge function*, and a *trusted user-defined ecall function*. The untrusted proxy function transparently abstracts the ecall mechanism for the untrusted application. It switches into enclave mode and effectively calls the corresponding bridge function. The bridge function sanitises and manipulates inputs and, in turn, calls the user-defined ecall function, which carries out some trusted computation. The ocall mechanism is implemented similarly but the direction is reversed: proxy function is part of the trusted code whereas bridge and user-defined functions are untrusted. As we detail in Section 4.1, the SDK introduces an approach to automatically generate proxies and bridges from annotated function signatures [21] as an attempt to mitigate attacks harnessing improper interactions between trusted and untrusted worlds. Note that while the enclave can read and write to the memory of the host application, the application is unable to directly access the enclave's memory. Alternative TEE implementations include TrustZone [39], SEV [47], Keystone [27], and Sanctus [35].

3.2 Symbolic execution and *angr*

Symbolic execution is a technique that explores a program's state space using symbolic values instead of concrete ones [1, 4, 8–10, 25, 30, 31, 33, 38, 44, 48]—a symbolic value of a given type denotes *any* possible value of that type. For each command executed, this technique creates a corresponding constraint over these values to capture the command's effects. The constraints accumulated at different execution points represent *symbolic states* that compactly capture a (possibly very large) set of concrete states—in fact, they normally capture all concrete states reaching that point. Branching on a symbolic value leads the execution to be split into two states, each of which capturing one of the branches. The branching conditions are tracked into what is called a *path condition*—a conjunction of all conditions for the branches taken up to that point; this constraint is also part of the symbolic state. A constraint solver is used to check symbolic states for satisfiability—unsatisfiable states are unreachable—and to generate concrete values for the program's variables at the corresponding execution point; these

[1] In fact, leaf functions to instruction encls.
[2] Leaf functions to instruction enclu.

concrete values can be very useful in generating concrete inputs to exercise that execution path. Let a and b be integer variables, and $\underline{s_1}, \ldots, \underline{s_5}$ annotations representing symbolic states. For the snippet: $\underline{s_1}$ **if** (b > 0) {$\underline{s_2}$ a := 0; $\underline{s_3}$ **if** (b == 0) {$\underline{s_4}$} } **else** {$\underline{s_5}$ a := 1; $\underline{s_6}$}, a symbolic execution engine would create symbolic states $\underline{s_1} \equiv True$, $\underline{s_2} \equiv b > 0$, $\underline{s_3} \equiv b > 0 \wedge a = 0$, $\underline{s_4} \equiv b > 0 \wedge a = 0 \wedge b = 0$, $\underline{s_5} \equiv (\neg\, b > 0)$, and $\underline{s_6} \equiv (\neg\, b > 0) \wedge a = 1$, for instance. Note that $\underline{s_4}$ represents an unreachable state with its corresponding unsatisfiable constraint.

angr [48] is a symbolic execution engine that targets binaries. It is a powerful tool that offers a number of built-in analyses and instrumentation constructs while being user-friendly. It has been successfully used to analyse real-world binary code [16, 42] and even enclaves [11, 14, 31]. *angr* provides the ability to instrument binaries with breakpoints, which can intercept the execution of specific instructions, and high-level function summaries, which can simulate and add extra behaviour to the binary. These two constructs are fundamental in designing our tool as we detail in Section 4.2.

Symbolic execution is an approach that tends to offer a good compromise between coverage and efficiency. Typically, it is slower but offers better coverage than testing or fuzzing, and it offers worse coverage but is quicker than formal verification approaches. We believe that this balance favours the use of symbolic execution. Our tool is intended to be used before enclaves are distributed; it symbolically executes enclaves to report vulnerabilities (i.e. violations to our notion of an orderly enclave) which need to be addressed by the enclave developer.

4 ANALYSING ORDERLY INTEL SGX ENCLAVES

The guarantees offered by TEEs can be easily thwarted if the design and implementation of an enclave is flawed. Confidentiality is pointless if the untrusted host application can just exfiltrate secrets, and isolated execution useless if the adversary can overly influence it. In this section, we propose a design for *orderly enclaves* that splits enclave execution into several phases, each of which has a specific purpose and expectations for the code being executed there. Building upon this design definition, we also discuss how Guardian analyses binaries, looking for violations of our design and potentially uncovering vulnerabilities.

SGX was designed with a very powerful adversary in mind. The trusted computing base of an SGX enclave consists of its code, the CPU package, and a few privileged containers [15] such as architectural enclaves [21]. All other elements are considered to be under the control of the attacker. So, privileged system software (hypervisor, OS, firmware, etc), and applications—including the enclave's host application—both code and data, are untrusted and assumed to be under the control of an attacker. Even enclaves are isolated from one another. Intel SGX's threat model does not encompass side-channel attacks; the enclave's developer is in charge of protecting its execution against them [20]. Of course, since untrusted code is in charge of loading and executing the enclave code, there is no guarantee that the enclave's code will be executed but only that its execution cannot be tampered with, without detection, or inspected in the clear. For our purposes, we use the following adversary model.

Definition 1. The enclave's execution, code and data, is integrity protected and confidential—hence, assumed to be executed as prescribed—but any non-enclave code or data is under the attacker's control and can be freely manipulated. Therefore, values read from or written to non-enclave memory, or executing non-enclave code should account for arbitrary adversarial influence.

4.1 Orderly enclaves

The execution of an *orderly enclave* spans across the following phases: *entry*, *secure*, *ocall*, and *exit*. Roughly speaking, the *entry* sanitises the inputs of some secure function, *secure* carries out the computation of this secure function, *ocall* is a phase by which the secure function might call into untrusted code—typically to realise some OS-assisted functionality—and *exit* sanitises the outputs of the secure function. We capture these different phases via annotations *TransitionAnnotations* = (*Entry*, *Secure*, *OCall*, *Exit*). The elements in this 4-tuple identify different addresses in the enclave binary denoting transitions between these phases. We discuss these annotations, what each of these phases entails in terms of the enclave's expected behaviour, and how this is achieved by enclaves built with the Intel SGX SDK framework [20, 21] in the following. The SDK code is available at [22]; we discuss and target version 2.12 for Linux, unless we state otherwise.

The pair *Entry* = (*EntryAddress*, *EntrySanitisationDone*) annotates the enclave's entry point and the address at which the CPU registers sanitisation must have been completed, respectively. An orderly enclave starts its execution at *EntryAddress* in phase *entry*, which ends as the *secure* or *exit* phase begins. While the transition to *secure* represents the standard enclave behaviour, transitioning into *exit* can be caused by some failed validation at *entry*, for instance. During *entry*, the enclave code has to set up its state appropriately so that it can securely execute without allowing untrusted resources to overly influence its behaviour. This phase is responsible for: setting up the low-level machine state and securing input arguments over which the enclave will compute.

The enclave has to properly sanitise CPU registers upon entry as they are shared between trusted and untrusted executions. In the Intel SGX SDK, the *trusted runtime system* (*tRTS*) is the SDK runtime library in charge of such sanitisation—the assembly routine enclave_entry performs this task. While registers used to pass arguments to the enclave execution must have their values preserved, the other registers which might impact the execution of the enclave have to be treated carefully. For x86-64 architecture, this routine expects registers *rax*, *rbx*, *rcx*, *rdi*, and *rsi* to store input arguments for the enclave's execution. For instance, *rbx* identifies the enclave thread to be executed, *rdi* the index of the ecall to be executed, and *rsi* a pointer to the marshalling structure containing the parameters of the ecall. Other general purpose registers, such as *rdx* and *r8*-*r15*, are cleared by tRTS so that they do not adversely impact the execution of the enclave. The CPU flags also need to be properly handled. For instance, the work in [56] shows that the Alignment Check (AC) flag and the Direction Flag (DF)—both part of the *rflags* register—can be harnessed to carry out powerful attacks against enclaves. The SDK clears both flags upon entry. Finally, the enclave should use its own private and protected stack for trusted execution. This phase should make sure that the low-level machine state switches from the untrusted stack to the trusted enclave stack. This routine sets up registers *rbp* and *rsp* so that they point to the trusted stack. We capture this low-level sanitisation with predicate *EntrySanitisationValidation* which holds if and only if all the following conditions are true: registers *rdx*, and *r8*-*r15* are zeroed; *rbp* and *rsp* point to the trusted stack; and flags AC and DF are cleared. We do not constrain the values of *rax*, *rbx*, *rcx*, *rdi*, *rsi* as they store arguments for the enclave execution. So, for an orderly enclave, when address *EntrySanitisationDone* is reached, we specify that this predicate must hold.

During *entry*, the enclave should also copy input arguments from untrusted into trusted memory. This copy should prevent, for instance, time-of-check time-of-use attacks whereby untrusted memory can be manipulated to bypass checks executed by the enclave. Note that in the adversary model we propose, the attacker is fully in control of the untrusted memory and could alter some values in memory between the enclave checking and using them. The SDK advocates the use of the *Enclave Definition Language* (EDL) and the *edger8r tool* [20, 21] to carry out this copy. EDL allows developers to annotate pointer parameters with specific copy and validation policies, whereas the edger8r tool generates the code to carry out the copying and validation based on these policies. The in and out EDL annotations are called *pointer direction attributes*: they represent whether the parameter is passed from the calling to the called procedure or the other-way around, respectively. Listing 1 presents a very basic EDL definition with an ecall and an ocall function. In Listing 2, we present abstract templates for prominent functions in the Intel SGX SDK's programming model. While sgx_ecall_function and ocall_function represent a bridge ecall function and a proxy ocall function, respectively, which are generated by edger8r based on our EDL definition, ecall_function is user-defined. Each number in these templates denotes a block of code; they are accompanied by a brief description of their purpose. Function sgx_ecall_function manipulates the parameters and calls the secure computation denoted by ecall_function.[3] The [in] annotation for i generates both a pointer validation in ① that checks whether i points to an **int** that lies in the memory outside of the enclave, and the code to copy the **int** pointed by i into the trusted heap in ②; local variable t_i points to this new trusted **int**. During the *entry* phase, we require an orderly enclave to respect the *EntryPolicy*: the enclave can read from untrusted memory (to carry out argument copies, for instance) but it is not allowed to write to untrusted memory.

After registers and input memory have been properly handled, the *secure* phase can start. It captures the proper trusted computation to be carried out. A binary can have many distinct secure functions defined. For instance, in the SDK paradigm, each ecall function typically describes a different secure computation that the enclave implementation offers. In our framework, we use the annotation $Secure = (SBegin_1, SEnd_1), \ldots, (SBegin_n, SEnd_n)$ to identify these computations where the pair of addresses $(SBegin_i, SEnd_i)$ identifies the beginning and end of the secure computation i. For instance, the address at which ecall_function is called in sgx_ecall_function and

[3] In fact, sgx_ecall_function receives a *marshalling structure* which contains as elements the values of i and retv. For the sake of presentation, we simplify our function definition to use these elements directly. We make a similar simplification for ocall_function.

Listing 1: EDL example.

```
enclave {
  trusted { \\ ecalls definintion block
    public void ecall_function([in] int *i, [out] int* retv);
  };
  untrusted { \\ ocalls definition block
    void ocall_function([in, out] int *j);
  };
};
```

Listing 2: Templates for the main functions of an enclave in the Intel SGX SDK programming model.

```
// ecall bridge function edger8r-generated based on our EDL
sgx_status_t sgx_ecall_function(int* i, int* retv)
{
  ① Argument-pointers-outside-enclave check
  ② Copy from untrusted to trusted memory
  ecall_function(t_i, t_retv);
  ③ Copy from trusted to untrusted memory
}
// user-created ecall function
void ecall_function(int* i, int* retv)
{
  ④ secure commands pre–ocall
  ocall_function(v);
  ⑤ secure command post–ocall
}
// ocall proxy function edger8r-generated based on our EDL
sgx_status_t ocall_function(int* v)
{
  ⑥ Argument-pointers-within-enclave check
  ⑦ Copy from trusted memory to untrusted memory
  sgx_ocall(index, u_v)
  ⑧ Copy from untrusted memory to trusted memory
}
```

the address of the instruction following this call (i.e., the return address for this call) could account for such a pair. During the *secure* phase, the enclave has to follow the *SecurePolicy* which forbids it from reading from and writing to untrusted memory. Performing one of these actions could indicate that the *entry* phase was not performed correctly and there still is some inputs that need copying into trusted memory, or that some bug exists in the enclave code which is leaking information into untrusted memory.

There are subtleties that should be observed about the EDL-edger8r approach. For instance, the edger8r tool does not deep copy parameters, it only shallow copies pointer elements of a data structure. The developer is in charge of writing specific procedures to carry out deep copies if needed. Moreover, the enclave developer might decide to completely bypass the pointer validation and pointed-to memory copy offered by the EDL-edger8r approach by using the annotation [user_check]. In this case, the developer should create its own validation and copy code. Note that, for cases where the EDL-edger8r approach is not sufficient or not used, the *secure* phase will not match exactly the execution of the user-defined ecall function; the code block used for pointer validation and deep copy will be part of this function and the *secure* phase should start after this block. Including such memory manipulations in this phase would constitute a violation of the *SecurePolicy*.

During the *secure* phase, the enclave might need to transition into untrusted code to carry out functionalities that are unavailable within it. Typical examples are OS-assisted functions such as manipulating a file or network socket; system calls are unavailable within the enclave. The SDK's programming model proposes the use of *ocalls* to implement such behaviour. The ocall abstraction is implemented by a *proxy function* in the enclave code that manipulates parameters and call the corresponding ocall function in untrusted code. Since we only consider the enclave's design and code, we are only interested in this proxy function. In Listing 2, ocall_function depicts the template of the proxy function generated by edger8r based on the corresponding EDL definition in Listing 1. Based on the [in,out] EDL annotation for pointer v, edger8r generates: a validation that the **int** pointed by v lies within the enclave in ⑥, the code to copy this **int** into untrusted memory in ⑦ (this new copy is pointed to by u_v), and the code to copy back the **int** from u_v to v in ⑧. The call to sgx_ocall(index, l_v) triggers a series of tRTS functions which are in charge of switching from trusted code into untrusted code and back into enclave code once the untrusted execution is completed, saving and restoring the enclave state in the process. We use the annotation $OCall = (OBegin_1, OEnd_1), \ldots, (OBegin_m, OEnd_m)$ to identify ocall proxy functions where the pair of addresses ($OBegin_i$, $OEnd_i$) denotes the beginning and end of the *ocall* phase, respectively. For instance, the address at which ocall_function is called and the following address in ecall_function could form such a pair. Multiple ocall functions can be called during the secure execution. For each of them, the enclave transitions from the *secure* phase to the *ocall* phase and back. During the *ocall* phase, the enclave can read and write to untrusted memory; *OcallPolicy* allows these actions. These reads and writes are necessary to pass parameters across the trust boundary.

Finally, after the secure computation has been performed, the enclave transitions into its final phase of execution: *exit*. At this stage, the enclave has to both sanitise registers to avoid leaking information about the trusted execution, and copy outputs into untrusted memory. Our framework captures this phase with annotation *Exit* = (*ExitAddress*) where *ExitAddress* marks the end of the enclave execution. This phase starts as soon the *secure* phase ends and finishes when execution reaches *ExitAddress*. When this address is reached, the predicate *ExitSanitation* must hold, where it expects that registers *rcx*, *rdx*, *r8-r15* are cleared, and that stack pointers *rbp* and *rsp* point to untrusted memory. The end of the enclave_entry assembly routine is in charge of carrying out this sanitation on exit. Moreover, the results of the secure computation, or at least some public part of it, have to be copied out to untrusted memory if they are to be accessed by any entity that is not the enclave itself. The SDK also proposes the EDL-edger8r paradigm to effectively carry out this copy. For the annotation [out] for ecall_function parameter retv in Listing 1, edger8r generates the code to both validate that retv points to an **int** outside the enclave, and to copy the **int** pointed by t_retv, possibly modified by the execution of ecall_function, into the address pointed to by retv. This phase enforces *ExitPolicy*: it restricts access to untrusted memory to writes only.

Figure 1: Typical execution with our framework's relevant address annotations and their corresponding Intel SGX SDK enclave commands.

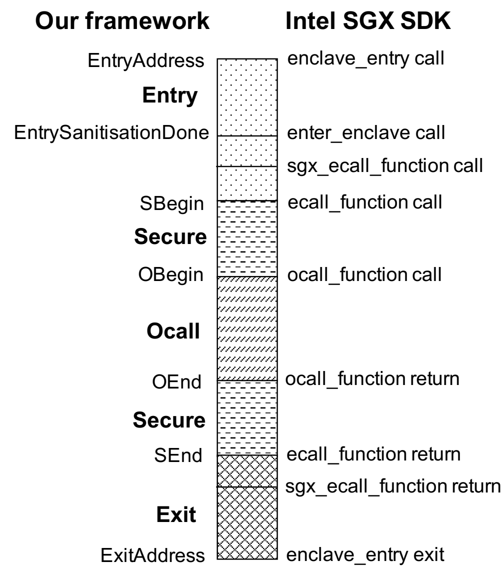

Figure 1 presents a typical execution of an orderly enclave with our address annotations and their corresponding SDK commands. Each phase has its own pattern; we label each of them in bold font. Address annotations and SDK commands are in normal font.

Definition 2. An enclave is *orderly* with respect to a given *TransitionAnnotations* if and only if the following hold:

- *Transition conditions*: the enclave transitions must respect the following rules. It must start at *entry*. From *entry*, it can transition to either *secure* or *exit*. The *EntrySanitisationDone* address must be reached before the enclave enters the *secure* phase. From *secure*, it can transition to either *ocall* or *exit*. From *ocall*, the enclave can only transition back to *secure*.
- *Sanitisation conditions*: at the *EntrySanitisationDone* address, the *EntrySanitisationValidation* predicate must hold, whereas at *ExitAddress*, the *ExitSanitisationValidation* predicate must hold.
- *Untrusted memory access policies*: during each of these phases, the enclave must respect the corresponding policy to access untrusted memory.

Although satisfying the conditions for being orderly does not guarantee security or correctness, they serve as a sensible design guideline created with the goal of preventing security issues arising from the interaction between trusted and untrusted resources. As we demonstrate later, checking for orderliness might be a useful first step into ensuring the correct/secure behaviour of an enclave as it can detect problematic behavioural patterns.

4.2 Enclave analysis with *Guardian*

We introduce Guardian: a tool, built on top of the binary analysis framework *angr* [48], that uses symbolic execution to validate an enclave binary against our orderly definition and to look for some typical memory-corruption vulnerabilities.

In *angr*, one can create a Python function to be executed when some specific address is reached in the symbolic exploration of the binary—these functions are called *SimProcedures* and the process of assigning a function to a specific address is called *hooking*. Moreover, one can extend the state of the symbolic execution with some variables described in what is called a *SimState plugin*. Our tool extends the state of the symbolic execution with a variable that keeps track of the different phases and a flag that denotes whether entry sanitisation has occured. We use SimProcedures to check and implement phase transitions, and they are hooked to the corresponding addresses in *TransitionAnnotations*. For instance, SimProcedure *ToSecure* checks whether the enclave is in either *entry* or *ocall* phases and whether the entry sanitisation flag is set. If these checks succeed, the enclave execution state is switched to the *secure* phase. This procedure is hooked at addresses $OEnd_i$ and $SBegin_i$. A *transition violation* is reported if a transition is being triggered from the wrong phase, whereas a *sanitisation violation* is reported when some sanitisation condition is not met. We also create SimProcedures to simulate some SGX-specific instructions unrecognised by *angr*.

Given the input binary and sizes to the enclave's stack and heap, we determine the enclave's memory layout and instrument data structure global_data_t with the enclave's size, and its heap's size and start offset with respect to the enclave's base address, and the thread_data_t structure with the stack's size and offset with respect to the corresponding Thread Control Structure (TCS); we only consider single-threaded executions. This instrumentation allows our tool to precisely execute memory range checks in the binary. The precise identification of the enclave's memory layout is a key element in our framework. It is used to enforce our policies and stack sanitisation conditions, and to implement some abstractions to account for adversarial behaviour. We use *angr*'s breakpoints to intercept reads and writes to memory and the memory bounds that we calculate to establish whether they satisfy our policies. For instance, a read from an address outside the enclave when in phase *secure* is reported as a violation. Our tool breaks down such violations into: out-of-enclave read, write, or jump violations. Moreover, our tool ensures that reads from a memory address outside the enclave are symbolic, accounting for adversarial behaviour. Furthermore, we also use breakpoints to report reads from, writes to, and jumps to symbolic addresses within the enclave as they can denote memory-corruption vulnerabilities—we call these *symbolic read, write, and jump violations*, respectively.

Guardian was designed to analyse ecalls individually. Together with an enclave binary and its heap and stack sizes, an analyst would manually input *TransitionAnnotations* and the identifier of the specific ecall that they want to examine, and Guardian would carry out the symbolic examination of this enclave checking for orderliness and memory-corruption vulnerabilities. As we mentioned, the inputs for this ecall and any non-enclave memory values are assumed to be attacker controlled and treated as symbolic values by Guardian. To facilitate the use of our tool, however, we propose heuristics to automatically find ecalls and ocalls in a given enclave binary, generating *TransitionAnnotations* in the process. With these heuristics, a Guardian user can simply pass an enclave binary as an input - together with heap and stack sizes - and rely on these heuristics to automatically find ecalls and ocalls and craft *TransitionAnnotations* so that all the ecalls in the binary can be analysed; of course, there is an opportunity for parallelising the analysis of these ecalls which we have not yet explored. Our heuristics use debug symbols to create the associations given in Figure 1. They should correctly annotate SDK enclaves for which the EDL-edger8r combination works without any adjustment such as tailor-made deep copying. It cannot be used to analyse closed source binaries as their symbols are usually stripped.

Guardian does not account for *reentrant ecalls*, namely, ecalls triggered within an ocall execution. Hence, violations caused by this sort of behaviour are not reported by our tool. Furthermore, our tool does not account for implicit enclave exit when a hardware exception occurs, even though the corresponding code has been the source of attack vectors [6]. Capturing these sorts of behaviour would significantly increase the complexity of our analysis so we leave this for future work.

4.3 Detecting enclave vulnerabilities

The main inspiration for our framework is the work in [56]. In that work, a number of vulnerabilities are *manually* identified. Our framework's sanitisation conditions and untrusted memory access policies outline tests that can *automatically* identify those vulnerabilities—or their effects. The work in [14] was published as we were creating on our framework and inspired our detection of typical memory-corruption vulnerabilities. In this section, we discuss how our framework can flag behaviours exhibiting vulnerabilities identified in the literature. We split the vulnerabilities into three categories as follows.

4.3.1 Register sanitisation. As CPU registers are shared between trusted and untrusted execution, they have to be properly sanitised when a transition occurs between these two execution modes. Upon enclave entry, failure to sanitise the the Alignment Check (AC) flag and Direction Flag (DF) in the x86_64 architecture, for instance, can give an attacker the ability to overly influence and probe the behaviour of an enclave. Moreover, failure to clear registers upon exit can cause information about the trusted execution to leak into the untrusted world. Furthermore, failure to properly switch between trusted and untrusted stack can cause the untrusted environment to both exert undue control over the trusted execution and the leaking of confidential information. Runtime libraries should properly set, restore and clear registers to prevent these vulnerabilities.

Our *EntrySanitisation* and *ExitSanitisation* conditions together with our untrusted access policies can identify such vulnerabilities. For instance, an enclave behaviour that does not clear the AC and DF flags would lead to an execution where *EntrySanitisation* does not hold at address *EntrySanitisationDone*. This execution is a witness for this vulnerability that would be identified by our tool.

4.3.2 Memory range checking. Failure to properly establish whether some memory range lies within or outside the enclave might lead to vulnerabilities such as writing confidential data into untrusted memory or reading and branching on an attacker-controlled value. The absence of a range check, an overflow on address calculation, or a misuse of SDK functions sgx_is_within_enclave/sgx_is_outside_enclave have been identified as causes to such failures. These functions check whether a buffer is completely inside and outside trusted memory, respectively; overlapping trusted-untrusted memory buffers fail both checks. Thus, sgx_is_within_enclave returning false does not mean that the memory range passed as an argument lies completely outside the enclave—an assumption wrongly made by some developers in designing real enclaves. Another specific instance of lacking or faulty range checks is null-pointer dereferencing. In C/C++ code, the null-pointer value—a pointer to the zero address—is used to convey that a pointer is uninitialised. When a program tries to dereference such a pointer, the process has typically no memory allocated at that address, causing a segmentation-fault crash. Pointer range validation is usually neglected for this address because of that. In the SGX trust model, however, an attacker is able to manipulate the process' virtual memory and allocate some memory at address zero. So, instead of a crash, the attacker can manipulate the value at this address to force an enclave to possibly execute some undesired behaviour.

The untrusted-memory-access policies enforced by our framework should identify effects of these vulnerabilities. For instance, a faulty range validation in the *entry* phase is likely to lead to some unintended access to untrusted memory in the *secure* phase. Our framework would capture this behaviour as a violation to the *SecurePolicy*. The enclave developer can, then, analyse the trace leading to this violation and find out whether a range check is missing or faulty.

4.3.3 Memory copying across trust boundary. Failure to copy data across the trust boundary might cause the enclave to work on untrusted data, being exposed, for instance, to time-of-check time-of-use attacks. As the attacker controls untrusted memory, the contents of a memory buffer may be changed between checking for some condition and using it in some computation. One cause for such failure is wrongly assuming that the EDL-edger8r approach (deep) copies pointer elements of data structures, or simply overlooking the ability of the attacker to manipulate untrusted memory addresses.

Our policies restricting the access to untrusted memory should capture the effects of such a faulty copy. Moreover, we capture the attacker's power to alter the value at untrusted memory addresses at any point by making all values read from untrusted memory symbolic. Our framework tries to strike a compromise by allowing double fetches of untrusted pointers during *entry* and *ocall* phases, but no fetch is allowed in *secure* and *exit* phases.

4.4 Evaluation

We have analysed 15 binaries ranging across 12 different enclave samples. We discuss some interesting violations found. The relevant results of our analyses are given in Table 1.[4] We report the type of violations that we found for each binary, their number of ecalls, how many exhibited some violation, how many timed out, how many were stopped. We set a 20-minute time budget per ecall analysis. We stop examining an ecall if it reaches a stage where more than 100 branches are being simultaneously explored or 20 violations have been found. The first stop condition is meant to prevent branch explosion whereas the second an excessive number of violations from being reported. We used a machine with Intel i7-9750H CPU, 16GB of RAM, running Ubuntu 18.04.

[4]Guardian is open source and available at https://github.com/blockhousetech/guardian together with the enclave binaries we generated.

Table 1: Relevant evaluation results.

Example	sanitisation	out-of-enclave read	symbolic read	out-of-enclave write	symbolic write	out-of-enclave jump	symbolic jump	#ecalls	#flagged	#timeout	#stopped
tls-cli		✓			✓			8	2	5	1
seal		✓			✓			6	1	0	1
http		✓	✓		✓			3	2	0	1
crypto		✓			✓			7	1	0	3
gmp	✓	✓						7	3	1	4
wolfssl			✓					22	13	0	0
discovery	✓							7	0	0	0

We analysed four SGX architectural enclaves created by Intel: launch, quoting, provisioning, and provisioning certificate enclaves. We used the version 2.12 of the SDK to build the code at [24]. Each of them plays a vital role in the SGX trusted ecosystem. These enclaves offer 11 ecalls altogether. Our analysis could not identify any violations on them. As no violations or timeouts were found, we did not add these result to Table 1. The absence of violations can be explained by the level of scrutiny these enclaves must have gone through given their importance in Intel's trust ecosystem. Also, it highlights the ability of our tool in checking enclaves without reporting false positives. We have manually inspected these enclaves and, to the best of our examination, they seem to indeed abide by our orderly definition.

We analysed the enclaves examined by *TeeRex* in [14]; their binaries can be found at [54]. All these enclaves do not properly sanitise AC/DF flags. So, they can be susceptible to the attacks in [56]. Not only these enclaves, but any enclave built with SDK versions < 2.7.1 will suffer with this vulnerability. For such enclaves, our tool reports an entry sanitisation violation. TeeRex does not check for this sort of sanitisation condition and it does not analyse runtime-library code, and so it is unable to find those. We did not carry out a more thorough analysis of these enclaves as they rely on [user_check]-annotated parameters, and therefore would require manual annotation as opposed to relying on our heuristics. Therefore, these examples are not part of our results table.

Enclaves *tls-cli*, *seal*, *http*, *crypto* use the popular *Rust SGX SDK* framework [61]: it ports the Intel SDK to the memory-safe language Rust—we used the version 2.12 of the Intel SDK to build the code at [53]. *tls-cli* implements a TLS client, *seal* data sealing functions (i.e. data encryption for persistent storage), *http* a http requester, and *crypto* some cryptographic functions. Enclave *gmp* is demo program for the GNU multi precision arithmetic library ported to SGX; we used the version 2.6 of the SDK to build the code at [23]. It offers a few basic mathematical operations and π estimation. The *wolfssl* enclave can be used to create a trusted TLS client/server application based on the wolfssl library—we used the version 2.12 of the SDK to build the code at [62]. Enclave *discovery* is part of a micro-service implementing private contact discovery; we used the version 2.1.3 of the SDK to build the code at [49]. It allows clients to discover which of their contacts are registered for the service without revealing their contacts to the service operator.

For this evaluation, we have added a flag to our tool that indicates whether the enclave under analysis was built using versions 2.1.3 or 2.6, as opposed to version 2.12 of the Intel SDK, which is the version targeted by Guardian. When this flag is set, the sanitisation check is disabled and a slightly different instrumentation of the binary is made—these older SDKs use a different data structures to capture the memory layout of an enclave. Moreover, we had to create a new heuristic to identify ocalls in *Rust SGX SDK* enclaves; we added a new flag to enable it for the corresponding examples.

The *wolfssl* enclave uses a global array variable CTX_TABLE to store some metadata about TLS connections. For a number of ecalls, it receives a **long** ctxId parameter which is used to index-access this table. Since ctxId is user-provided, it is made a symbolic input by our tool and index accessing CTX_TABLE using it triggers a symbolic read violation. By further examining these violations, however, we realised CTX_TABLE's index accesses are limited to indices 0 and 1. This is a violation since our tool reports any read from a symbolic non-single-valued address as a symbolic-read violation. This condition might seem too restrictive, but we rather err on the side of caution, and over- rather than under-report. With a slightly more relaxed condition where a symbolic read is triggered when a symbolic value can take at least three different concrete values, our tool explores *wolfssl* without reporting violations. Based on the history of this enclave's repository, this index restriction has only been introduced recently and our tool could have identified its necessity.

The *symbolic-write* violations found in enclaves *tls-cli*, *seal*, *http*, and *crypto*, and the *symbolic-read* violation found in *http* are caused by the writing and reading of a symbolic address that is calculated based on an *attacker-controlled* memory-buffer length parameter - the use of this parameter is what makes these addresses symbolic. They represent false positives that need to be addressed in future versions of our tool. The support that we currently offer for detecting memory-corruption vulnerabilities is rather primitive as our focus is on checking for orderliness.

The sanity violations detected in the enclave samples *gmp* and *discovery* are due to their use of an old SDK which neglects to appropriately clear the flags register upon entry. Moreover, the *out-of-enclave-read* violations for samples *gmp*, *tls-cli*, *http*, *seal*, and *crypto* are all genuine issues uncovered by Guardian and that have been acknowledged by the sample maintainers. Unlike the false positives mentioned above, these issues were uncovered thanks to violations of our orderliness conditions, attesting the relevance of our definition. We discuss in detail these newly discovered vulnerabilities in the following.

4.4.1 New vulnerabilities discovered by Guardian. We have noticed a recurrent violation pattern across the enclaves we examined. For an ecall pointer parameter annotated with [in], if the corresponding argument for the EDL-edger8r generated bridge ecall function is NULL, the copy-into-trusted-memory code is not triggered and the null-pointer is passed to the ecall function. We use a simplified version of function sgx_tls_client_new of enclave sample *tls-cli*

Listing 3: Simplified version of bridge ecall function sgx_tls_client_new of *tls-cli* enclave sample.

```
static sgx_status_t sgx_tls_client_new(void* pms)
{
  CHECK_REF_POINTER(pms, sizeof(ms_tls_client_new_t));

  ms_tls_client_new_t* ms= SGX_CAST(ms_tls_client_new_t*, pms);
  sgx_status_t status = SGX_SUCCESS;
  char* _tmp_cert = ms->ms_cert;
  size_t _len_cert = ms->ms_cert_len ;
  char* _in_cert = NULL;

  CHECK_UNIQUE_POINTER(_tmp_cert, _len_cert);

① if (_tmp_cert != NULL && _len_cert != 0) {
    _in_cert = (char*)malloc(_len_cert);
    if (_in_cert == NULL) {
      status = SGX_ERROR_OUT_OF_MEMORY;
      goto err;
    }
    if (memcpy_s(_in_cert, _len_cert, _tmp_cert, _len_cert)) {
      status = SGX_ERROR_UNEXPECTED;
      goto err;
    }
    _in_cert[_len_cert - 1] = '\0';
    if (_len_cert != strlen(_in_cert) + 1)
    {
      status = SGX_ERROR_UNEXPECTED;
      goto err;
    }
  }

② ms->ms_retval = tls_client_new(ms->ms_fd, _in_cert);
}
```

presented in Listing 3 to illustrate this pattern; we have omitted all the code of the original function that is not relevant for our explanation to simplify our presentation. This ecall bridge receives a pointer to a string representing a path to a *tls certificate* and this string's length as the elements ms->cert and ms->cert_len of the input marshalling structure ms, respectively; they are stored into local variables _temp_cert and _len_cert, respectively. The **if**-block in ① copies this input string into the *trusted* memory pointed by local variable _in_cert. Note, however, that this block is only executed if _tmp_cert is not NULL, or the string's length is greater than zero. Otherwise, this code block is not executed, and the pointer variable _in_cert pointing to NULL is passed to the ecall tls_client_new as it is called in ②. For traditional programs, manipulating a NULL pointer would probably lead to a crash, causing the program's execution to be aborted - an inconvenient but arguably harmless outcome. In SGX's threat model, however, the attacker is assumed to control any non-enclave address like the NULL address; the attacker has control over the memory range in which the enclave is loaded. Therefore, in this example, the ecall function operates on a string for which value is attacker controlled. Thus, the attacker is able to manipulate the value of this string as the ecall function is executing bypassing all checks that this function carries out until the value is finally used to some supposedly secure calculation, effectively carrying out a time-of-check time-of-use attack. Guardian identifies this vulnerability to this sort of malignant behaviour as it would detect a dereference to this pointer in the *secure* phase; this behaviour triggers an out-of-enclave-read violation. A simple approach to correct such behaviour is to abort the behaviour of an ecall if null pointer is passed to the ecall function, preventing any use of the attacker-controlled memory buffer.

We identified this issue in all *Rust SGX SDK* enclaves in a common ecall t_global_init_ecall. It is used to initialise some enclave metadata information: an identifier and a path. Even though the path parameter is [in]-annotated, it is afflicted by the violation above and so path can be made attacker controlled. Parameters cert - leading to the case that we illustrate with Listing 3 - and hostname of *tls-cli*'s ecall tls_client_new, parameter hostname of *http*'s ecall send_http_request, and parameters str_a and str_a of *sgx-gmp*'s ecalls e_mpz_add, e_mpz_mul, e_mpz_div, e_mpf_div can all be similarly subverted by passing a NULL pointer. In all these cases, as we discussed in our illustrative example, the attacker has the power to change the value of these arguments even after the bridge function has finished processing inputs.

We have contacted the maintainers of these projects who have agreed these violations represented issues - they highlighted potentially harmful unintended behaviours - and have promptly fixed them. [5] Anecdotally, the *gmp* sample, which exhibits this behaviour, has been created and is maintained by Intel engineers. We have tested these enclaves after they have been fixed, and we can confirm that Guardian no longer finds these violations. We point out that these security issues were *uncovered thanks our orderliness definition* rather than by typical memory-corruption behavioural patterns, that is, orderliness conditions were violated leading to the identification of these issues. As some of these enclaves had already been analysed by TeeRex and these vulnerabilities had not been reported, we have evidence that TeeRex was unable (or did not consider) these behaviours vulnerabilities. TeeRex is not publicly available for use, so we cannot test it out against this newly found vulnerabilities.

Based on a fundamentally different philosophy than similar enclave analysis techniques, our tool is capable of identifying subtle violations in a reasonable time frame. The most relevant violations that we identified were not detected via the typical memory-corruption vulnerabilities proposed by TeeRex but via our policies on untrusted memory accesses and sanitisation conditions.

5 CONCLUSIONS

We propose a definition of orderly enclave as a means to capture the intent of enclave developers; we define where phase transitions occur and, consequently, how enclaves should behave at different stages of their execution. Based upon this notion, we created Guardian: an open-source tool that uses symbolic execution to validate enclaves' binaries against this definition—and also to find some typical memory-corruption vulnerabilities. Violations to this definition uncovered by Guardian are likely to expose vulnerabilities arising from improper interactions between the trusted and untrusted resources. We also demonstrate that our framework is capable of efficiently identifying violations in practical enclaves, uncovering in some case subtle behaviours. We identify violations in practical enclaves which went undetected when analysed by similar state-of-the-art tools. These vulnerabilities newly discovered by Guardian have been acknowledged and fixed by the enclave maintainers. These results show that Guardian can be a valuable tool in improving the security of enclaves, helping developers to be more confident about their design and implementation.

We plan to add some robust coverage metrics and to experiment with fine-grained notions of confidentiality. Currently, we have a rough coverage metric which counts the number of executions examined that have reached the code of the ecall under analysis. Furthermore, we plan to improve Guardian to handle reentrant ecalls (i.e. an ecall function call from a ocall execution) and asynchronous enclave exists. Another topic for future work is the feasibility of automatic binary hardening for the violations that we find, namely, can we automatically patch enclaves to prevent the violations found from being reached?

[5] https://github.com/apache/incubator-teaclave-sgx-sdk/pull/322, https://github.com/intel/sgx-gmp-demo/commit/a05da606b2cfd4710c80e3c99068a6b97dc31888

REFERENCES

[1] 2015. *Triton: A Dynamic Symbolic Execution Framework*. SSTIC.

[2] Fritz Alder, Jo Van Bulck, David Oswald, and Frank Piessens. 2020. Faulty Point Unit: ABI Poisoning Attacks on Intel SGX. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 415–427.

[3] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 689–703.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2016. Enhancing Symbolic Execution with Veritesting. *Commun. ACM* 59, 6 (May 2016), 93–100.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

[6] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks against Intel SGX. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1213–1227.

[7] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469.

[8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages.

[10] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *2011 33rd International Conference on Software Engineering (ICSE)*. 1066–1071.

[11] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 142–157.

[12] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D. Garcia. 2021. VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C.

[13] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 185–200.

[14] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 841–858.

[15] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[16] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries.. In *NDSS*.

[17] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. 1–6.

[18] S. Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security Privacy* 14, 6 (2016), 54–62.

[19] Intel Corporation 2020. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4*. Intel Corporation. Available at: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/332831-sdm-vol-3d.pdf.

[20] Intel Corporation 2020. *Intel(R) Software Guard Extensions Developer Guide*. Intel Corporation. Available at: https://download.01.org/intel-sgx/sgx-linux/2.12/docs/Intel_SGX_Developer_Guide.pdf.

[21] Intel Corporation 2020. *Intel(R) Software Guard Extensions Developer Reference for Linux* OS*. Intel Corporation. Available at: https://download.01.org/intel-sgx/sgx-linux/2.12/docs/Intel_SGX_Developer_Reference_Linux_2.12_Open_Source.pdf.

[22] Intel Corporation 2020. *Intel(R) Software Guard Extensions for Linux* OS Repository*. Intel Corporation. Available at: https://github.com/intel/linux-sgx.

[23] Intel Corporation 2021. *Demo program for the GNU Multiple Precision Arithmetic Library for Intel(R) Software Guard Extensions*. Intel Corporation. Available at: https://github.com/intel/sgx-gmp-demo/tree/85cd0409175d7b87160f6966c79a1be08d6b0aac.

[24] Intel Corporation 2021. *Intel(R) Architectural Enclaves Code*. Intel Corporation. Available at: https://github.com/intel/linux-sgx/tree/sgx_2.12/psw/ae.

[25] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

[26] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 205–221.

[27] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages.

[28] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-Oriented Programming against Secure Enclaves. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 523–539.

[29] K Rustan M Leino. 2008. This is boogie 2. *manuscript KRML* 178, 131 (2008), 9.

[30] M. Lindner, J. Aparicius, and P. Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. 108–114.

[31] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *Proceedings of the 2017 Network and Distributed System Security Symposium*.

[32] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2018. Hardware-Based Trusted Computing Architectures for Isolation and Attestation. *IEEE Trans. Comput.* 67, 3 (2018), 361–374.

[33] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[34] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[35] Job Noorman, Jo Van Bulck, Jan Tobias Mühlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes Götzfried, Tilo Müller, and Felix Freiling. 2017. Sancus 2.0: A Low-Cost Security Architecture for IoT Devices. *ACM Trans. Priv. Secur.* 20, 3, Article 7 (July 2017), 33 pages.

[36] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. 2, 2, Article 28 (June 2018), 30 pages.

[37] Open Enclave Community 2020. *Intel(R) Software Guard Extensions Developer Guide*. Open Enclave Community. Available at: https://openenclave.io/sdk/.

[38] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (2013), 391–425.

[39] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6, Article 130 (Jan. 2019), 36 pages.

[40] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. Enclavedb: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.

[41] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. *CrossTalk: Speculative Data Leaks Across Cores Are Real*. Vol. 2021-May. Institute of Electrical and Electronics Engineers Inc., United States.

[42] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2017. Bootstomp: on the security of bootloaders in mobile devices. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 781–798.

[43] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[44] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331.

[45] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society.

[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker *(USENIX ATC'12)*. USENIX Association, USA, 28.

[47] AMD SEV-SNP. 2020. Strengthening VM isolation with integrity protection and more. (2020).

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157.

[49] Signal developers 2021. *Signal app private contact discovery micro-service*. Signal developers. Available at: https://github.com/signalapp/ContactDiscoveryService/tree/a6a0c1505003865f5f99497caf59f6d31cebf660.

[50] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P Lopes, Sriram Rajamani, Sanjit A Seshia, and Kapil Vaswani. 2016. A design and verification methodology for secure isolated regions. *ACM SIGPLAN Notices* 51, 6 (2016), 665–681.

[51] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. 2015. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1169–1184.

[52] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2435–2450.

[53] teaclave developers 2021. *teaclave-sgx-sdk sample enclaves repository*. teaclave developers. Available at: https://github.com/apache/incubator-teaclave-sgx-sdk/tree/v1.1.3/samplecode.

[54] TeeRex authors 2021. *TeeRex binaries repository*. TeeRex authors. Available at: https://github.com/uni-due-syssec/teerex-exploits/tree/60604e574ea4bf5a319c2713efe6864aa6864e1c.

[55] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 991–1008.

[56] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1741–1758.

[57] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 178–195.

[58] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York,

NY, USA, 178–195.

[59] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX Fails in Practice. https://sgaxeattack.com/.

[60] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.

[61] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2333–2350.

[62] Wolfssl developers [n.d.]. *Wolfssl SGX enclave example*. Wolfssl developers. Available at: https://github.com/wolfSSL/wolfssl-examples/tree/e98762b110d5614a9faa0942a9f98b66350ee299/SGX_Linux.

[63] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. 2020. ACE: Asynchronous and Concurrent Execution of Complex Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 587–600.