Sigy: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals

Supraja Sridhara
ETH Zurich
Zürich, Switzerland
supraja.sridhara@inf.ethz.ch

Andrin Bertschi
ETH Zurich
Zürich, Switzerland
andrin.bertschi@inf.ethz.ch

Benedict Schlüter
ETH Zurich
Zürich, Switzerland
benedict.schlueter@inf.ethz.ch

Shweta Shinde
ETH Zurich
Zürich, Switzerland
shweta.shinde@inf.ethz.ch

Abstract

User programs recover from hardware exceptions and respond to signals by executing custom handlers that they register specifically for such events. We present Sigy attack, which abuses this programming model on Intel SGX to break the confidentiality and integrity guarantees of enclaves. Sigy uses the untrusted OS to deliver fake hardware events and injects fake signals in an enclave at any point. Such unintended execution of benign program-defined handlers in an enclave corrupts its state and violates execution integrity. 7 runtimes and library OSes (OpenEnclave, Gramine, Scone, Asylo, Teaclave, Occlum, EnclaveOS) are vulnerable to Sigy. 8 languages supported in Intel SGX have programming constructs that are vulnerable to Sigy. We use Sigy to demonstrate 4 proof of concept exploits on webservers (Nginx, Node.js) to leak secrets and data analytics workloads in different languages (C and Java) to break execution integrity.

CCS Concepts

• **Security and privacy → Systems security**.

Keywords

TEE; Intel SGX; Exception, Signal, and Interrupt handling

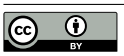
ACM Reference Format:
Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, and Shweta Shinde. 2025. Sigy: Breaking Intel SGX Enclaves with Malicious Exceptions & Signals. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3708821.3710838

1 Introduction

User programs rely on exceptions and signals to manage unexpected events or errors that may occur during execution. Programming languages allow developers to express event and error-specific logic in the form of *handlers*, such that when the program is notified of a particular event, the handler is executed automatically. These rich abstractions are facilitated by the cooperation of the hardware and the operating system (OS). When the hardware encounters runtime errors during the program execution (e.g., page faults, segmentation faults, timers, divide by zero), it notifies the OS via interrupts. The OS either handles the faults itself (e.g., load the page into memory) or forwards it to the user program's signal handler (e.g., DivideByZeroException). The program can also request the OS for notifications about events of its interest that either emanate from the system (e.g., Ctrl+C) or other processes on the system (e.g., synchronization between parent and child processes) and register handlers that should execute on such events [24]. Thus, the OS not only monitors for such events on behalf of the application and notifies it, but also diverts the control of the application to the event handlers in the program. Both these mechanisms facilitate rich functionality in the programs, while the hardware and the OS provide efficient notification and handler invocation.

Intel SGX provides a user-level abstraction called *enclaves* that protects sensitive data and code execution [38, 45]. The hardware protects enclave confidentiality and integrity even when the OS and other user processes are compromised. With such a strong threat model, Intel SGX limits the attack surface to the critical code running in an enclave. Since the enclave memory is rendered inaccessible to the OS, traditional programs that were written with the assumption of a trusted OS simply cannot execute inside enclaves (e.g., `syscall` instruction is illegal inside an enclave).

Due to its unique programming model, existing programs do not execute out of the box on Intel SGX [8, 13–18, 22, 23, 25, 28, 32, 41, 58]. As a solution, programmers can use SGX runtimes that provide a small trusted runtime that interfaces with the SGX hardware to expose a new high-level interface to the programmer. Alternatively, programmers can use a trusted library OS inside an enclave that can execute unmodified applications that were not programmed for enclaves. Runtimes and library OSes for Intel SGX support exception and signal delivery to enclaves since it is a much-required feature for programs. The hardware or the OS can inform the enclave about an exception or a signal by inducing an asynchronous exit. The enclave safely stores its current execution state and exits to untrusted code. The enclave can then be re-entered from another fixed entry point to execute corresponding pre-registered handlers for the exceptions or signals. During this flow, the hardware and the trusted software ensure that the OS cannot subvert the execution



ASIA CCS '25, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/25/08
https://doi.org/10.1145/3708821.3710838

of the enclave—it executes the handler and then resumes the execution at the point where it was interrupted. This mechanism allows enclaves to handle runtime events even when the OS is untrusted.

Heckler and WeSee introduce a new class of attacks called Ahoi attacks where an attacker uses notifications to compromise Confidential VMs (CVMs) enabled by Intel TDX and AMD SEV-SNP [56, 57]. Ahoi attacks use interrupts under the control of a malicious hypervisor that can trigger interrupt handlers in the CVMs. These interrupt handlers alter the global execution state of the CVMs and compromise them. In light of these findings, we revisit Intel SGX and analyze if an attacker can use notifications to compromise the security of enclaves. We investigate two lines of inquiry: (i) what events can the OS fake to trigger handler execution inside the enclave? and (ii) can such handler execution bring about direct changes to the enclave's global execution state (e.g., variables)?

In this paper, we introduce a new attack called Sigy where the OS compromises the enclave execution by inducing fake events and signals to execute benign handlers registered by the enclave. Intuitively, enclaves want to recover from a divide-by-zero and expect signals from another enclave. To handle such events, enclaves will register handlers that explicitly update the enclave state, say by changing the denominator to a non-zero value or invoking an event handler to respond to another enclave's request. If the OS convincingly tricks the enclave into falsely believing that such an event occurred, the enclave will stop its current execution and execute the handler that will explicitly update the enclave state (e.g., change a variable to a non-zero value or execute a function). In the least, this will result in corruption of the enclave's state resulting in a crash. If the OS injects the event at an opportune moment, it can use the effects of the handler to compromise the enclave. We demonstrate this phenomenon by introducing a new attack called Sigy, which exploits the OS's ability to fake signals to execute enclave handlers and subvert SGX guarantees.

We show that existing runtimes, library OSes, and programming language constructs are vulnerable to Sigy. We first analyze existing support to execute SGX applications: 8 runtimes (Intel SGX SDK, Open Enclave, Teaclave SGX-SDK, Asylo, Rust EDP, GoTEE, Enarx, and EGo) and 6 library OSes (Gramine, Scone, EnclaveOS, EdgelessRT, MystikOS, and Occlum). Then, we analyze the signal delivery mechanism and handlers for programs written in 9 languages (C, C++, Java, Python, Go, JavaScript, Rust, Julia and Wasm) to observe their behavior in enclaves. We find that 3/8 runtimes (Open Enclave, Teaclave SGX-SDK, and Asylo) and 4/6 library OSes (Gramine, Scone, EnclaveOS, and Occlum) are susceptible to Sigy because they do not detect the fake signals injected by the OS. Of the 9 languages we study, 8 (C, C++, Java, Python, Go, JavaScript, Rust, and Julia) offer language constructs for programs to register custom handlers. We hand-code applications in each of these languages to register handlers and execute them in vulnerable SGX runtimes and library OSes to confirm that they are indeed vulnerable to Sigy. Next, we demonstrate that Sigy breaks the confidentiality and integrity of 4 open-source applications (Nginx, Node.js, machine learning) that have been ported to Intel SGX by prior works. Our proof of concept exploits on these enclaves leak secrets and change outputs. Depending on the victim enclave, Sigy may need to inject signals in a particular window of execution. We construct a proof of concept exploit against a worst-case application, a multi-layer perceptron, that requires 186 billion injections to bias the output.

Our proposed software defenses serve as point-wise solutions against Sigy. The vulnerable runtimes and library OSes have to make a design choice between either disabling functionality for security or leaving the onus on the developer to reason about the security. While the latter can be a pragmatic solution, new attacks like Sigy serve as an example that programmers using runtimes and library OSes for lift-and-shift should not be burdened with this decision. We conclude that some programs simply cannot be protected without limiting functionality. Our conclusion encourages runtime and library OS maintainers to disable vulnerable exception and signal delivery interfaces. Our detailed analysis of existing enclave ecosystems spanning runtimes, library OSes, programming language support, and existing enclave applications provides an in-depth exploration to help future lift-and-shift solutions in making judicious choices. In summary, Sigy draws attention to a new attack surface that requires a re-examination of the enclave ecosystem.

Contributions. The paper makes three main contributions:

(1) Sigy *Attack.* We present a novel attack on Intel SGX where a malicious OS can send fake signals enclave and trick them into executing enclave-registered handlers that change the enclave state.
(2) *Analysis.* Of the 14 frameworks for running applications in Intel SGX, 7 are vulnerable to Sigy because they forward fake signals to the enclave whereas 8/9 popular languages used for enclave programming support custom handlers.
(3) *Exploits.* We build exploits on 4 open-source enclave applications to demonstrate Sigy.

Responsible Disclosure. We informed all the 7 impacted runtimes and library OSes from September 2023 to January 2024. All the vendors of the runtimes and library OSes acknowledged that this is an issue. Sigy is tracked under 3 CVEs: CVE-2024-25371 for Gramine, CVE-2024-29971 for Scone, CVE-2024-29970 for EnclaveOS. Gramine has mitigated this issue with a patch [7] and other vendors are taking steps to fix it.
Sigy's tooling and PoC exploits are open-source at https://ahoi-attacks.github.io/sigy.

2 Sigy Overview

For functionality, applications register custom exception and signal handlers that alter the global state of the program. To preserve this functionality on Intel SGX, runtimes and library OSes provide mechanisms to send signals to applications that execute in enclaves. Sigy tricks the benign runtime and library OS signal handling mechanisms which results in the enclave executing the handlers. Therefore, the custom handlers in applications put together with the signal propagation infrastructure of runtimes and library OSes render the enclaves vulnerable to Sigy. Like previous works, we deem this new attack vector, which was previously unknown, as a vulnerability and not a bug in the enclave implementation [33].

Threat Model. We trust the hardware in the Intel CPU package and assume that it is free from bugs. The enclaves are launched and attested according to the Intel SGX specification, and all software that executes inside the enclave is assumed to be bug-free. This includes the enclave application code, trusted runtime, and

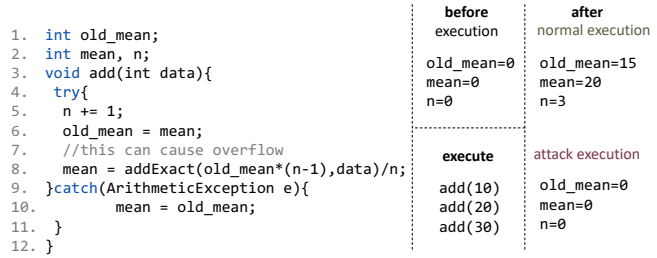


Figure 1: SIGY on Java applications. Attacker injects `sigfpe` 3 times to change the execution and data integrity.

library OSes. We assume that all software that executes outside the enclave, including the OS, untrusted runtime, and other processes are untrusted and can be malicious. All the SDKs. library OSes and runtimes we investigate except Occlum assume this threat model. For Occlum, we assume Occlum's threat model of distrusting processes within a single enclave [59]. In SIGY, the attacker does not have the ability to change the enclave's code (e.g., to introduce signal handlers). Instead, the attacker reuses existing handlers in the enclaves to compromise them.

2.1 Motivating Examples

Faking hardware exceptions. Consider a Java function in Fig. 1 that executes in an enclave whose add function (Line 3) is called 3 times with data as 10, 20, 30. The function computes and stores the new mean on every invocation. Now, consider a malicious operating system (OS) that wants to compromise the data integrity of this enclave. In this example, the attacker's goal is to ensure that the application never updates the `mean`, i.e., it remains 0 despite the 3 invocations of `add`. Because this code executes in an enclave, the OS cannot change the values in memory directly. However, observe that this enclave catches and handles `ArithmeticExceptions` to deal with bad data. The handler reverts the value of `mean`, discarding the effects of the bad data. If the OS can trigger this handler every time the `add` function is invoked, then the `mean` will not be updated.

The OS can use SIGY to achieve its goal. Specifically, the Java runtime converts the signal for floating point exceptions (`sigfpe`) that it gets from the OS to an `ArithmeticException`. This exception is then caught and handled by the enclave. Therefore, using SIGY the OS injects `sigfpe` to the Java code in the enclave when it executes Lines 4-9 in Fig. 1. This will result in the enclave always executing the exception handler. With 3 such signal injections, the OS ensures that the `mean` does not change, thus breaking integrity.

Faking user-defined signals. Several library OSes (e.g., Gramine, Scone) allow lifting and shifting unmodified applications like Nginx to execute in enclaves. Nginx is a web server that is highly optimized to provide maximum uptime, configured using a `config` file. Consider an unmodified Nginx server that executes in an enclave to serve `http` data with `config` in Lst 1. At a later point, an Nginx administrator introduces authentication tokens (JSON Web Tokens (jwt) [19]) in the `config` file (Lst.2). To enable the administrator to refresh the `config` file without downtime, Nginx performs the configuration refresh on receiving `sighup`. When the administrator sends `sighup` to the Nginx process, the Nginx process reads the new `config` file and starts using it.

```
1 ...
2 server {
3   listen 80;
4    location /products/ {
5      ...
6
7      }
```

Listing 1: Old `config`

```
1 ...
2 server {
3   listen 80;
4   location /products/{
5   auth_jwt  "API";
6   auth_jwt_type
7             encrypted }
```

Listing 2: New `config`

Now, consider a malicious OS that aims to disable this authentication mechanism in the Nginx server. To do this, the OS should be able to force the server to use the old configuration file without the authentication enabled. Library OSes protect enclave files by encrypting them. Therefore, the malicious OS cannot directly edit the `config` file. However, the OS can capture the old `config` file as an encrypted blob from the file-system before it is replaced by the administrator. Then, once the Nginx configuration upgrade is complete, and checked by the administrator, the OS writes the encrypted blob back to replace the new `config` file. Note that, this is not sufficient to trick the Nginx server into using the compromised configuration without restarting the enclave. The OS's goal is to ensure that when users connect to the Nginx server, they are served using the older configuration instead of the configuration that the administrator upgraded and checked, thus mounting a time-of-check time-of-use (TOCTOU) attack. For this, the OS uses SIGY to force the Nginx server to use this compromised configuration after the administrator checks that the configuration reload was successful by injecting `sighup` to the Nginx process.

2.2 SIGY Attacks on Real-world Enclaves

SIGY uses asynchronous signal injection to compromise enclaves by triggering expressive signal handlers. This requires a runtime or library OS that propagates hardware exceptions and signals to the enclave application. Further, SIGY uses handlers in the applications that perform computations that alter the enclave's global state. The enclave handlers depend on programming language constructs used in the application (e.g., signal registration constructs, and signal handling constructs). Therefore, we first evaluate 14 runtimes and library OSes and check if they propagate signals to the enclaves (Sec.3 and Sec. 4) by building proof-of-concept exploits for each of them. Next, we examine 9 programming languages and systematically analyze the constructs they provide for programs to register and execute custom signal handlers (Sec.5.2). Finally, we use the insights from our runtime and programming language analysis to demonstrate SIGY on 4 publicly available enclave applications (Sec.6). We extract secrets and change program execution to break confidentiality and integrity.

3 Faking Signals in SDKs

Background: Sending signals to threads. SDKs are typically used to execute a single process in an enclave. So, they may not enable mechanisms for the enclave to send signals to other processes. However, they do support multi-threading inside the same enclave. For sending signals between threads, the OS exposes the `tkill` system call. Along similar lines, some SDKs add mechanisms to enable enclave threads to send signals to each other. To support

Table 1: Library OS and Runtime analysis for SIGY. ✓: interface supported ✗: interface not supported ❉: cannot be analyzed as they are closed source. The last two columns indicate whether SIGY can compromise the enclave by injecting signals.

Type	Name	Interfaces		Can compromise with SIGY?	
		HW exception interface	Other signal interface	HW exception signals	Process signals
SDK	Intel SGX SDK [18]	✓	✗	no	no
	Open Enclave [23]	✓	✓	yes	yes
	Teaclave [28]	✓	✓	yes	yes
	Asylo [8]	✓	✓	yes	yes
	Rust EDP [25]	✗	✗	no	no
libos: 1-process per enclave	Gramine [17]	✓	✓	yes	no
	Scone [26]	❉	❉	yes	yes
	EnclaveOS [16]	❉	❉	yes	yes
	EdgelessRT [13]	✗	✗	no	no
libos: n-process per enclave	Mystikos [22]	✗	✓	no	no
	Occlum [58]	✓	✓	yes	yes
language runtime	GoTEE [41]	✗	✗	no	no
	Enarx [15]	✗	✗	no	no
	EGo [14]	✗	✗	no	no

sending signals to other threads, the SDKs add a new ocall interface to send a signal to the target enclave thread via the OS (Steps 1-4 in Fig. 2(b)). When the OS sends the signal to the target enclave thread, the untrusted runtime's handler propagates it to the trusted runtime using an ecall (Step 5) which in turn invokes the target thread's signal handler in the enclave.

Tab.1 shows an overview of our library OS and runtime analysis, and indicates whether SIGY can compromise an enclave that uses the respective library OSes and runtimes by injecting signals.

3.1 Intel SGX SDK

The Intel SGX SDK allows enclaves to register handlers for hardware exceptions (e.g., divide-by-zero) to perform custom handling when hardware exceptions occur. During normal operation, if the enclave executes an instruction that triggers a hardware exception (Fig.2(a)), the SGX hardware triggers an asynchronous exit. On an asynchronous exit, the trusted hardware stores the current execution state of the enclave into the protected state save area (SSA).

```
1 struct _exit_info_t {
2  uint32_t  vector;    // exception vector number
3  uint32_t  exit_type; // HW or SW exceptions
4  uint32_t  valid;     // supported/unsupported  }
```

Listing 3: Exit information stored in the SSA.

Then, it raises an exception that the untrusted OS traps on where the hardware stores information about the exit into the SSA as shown inLst. 3. Crucially, in SGX 2, the hardware stores information about the exit into the SSA as shown in Lst. 3 while SGX 1 does not store this hardware information [39].

```
1 if (ssa_gpr->exit_info.valid == 1) {
2   // info used to forward exception to enclave app
3   info->exception_valid = ssa_gpr->exit_info.valid
4   info->exception_vector = ssa_gpr->exit_info.vector;
5   info->exception_type = ssa_gpr->exit_info.exit_type; ... }
```

Listing 4: Exception handling for Intel SGX SDK.

This includes the validity, type, and reason (i.e., exception vector) for the asynchronous exit. Enclave software can access this information while untrusted software (e.g., untrusted runtime, OS)

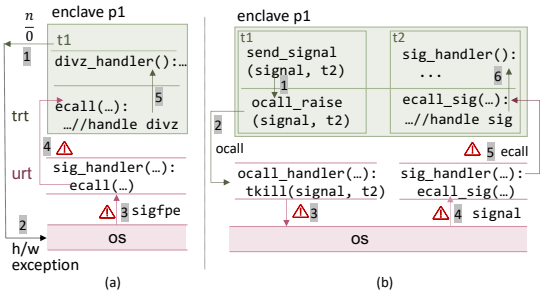


Figure 2: SDK interfaces. (a) Handling hardware exceptions (b) Handling intra-enclave signals.

cannot. The Intel SGX SDK's trusted runtime uses the exit information from the SSA to deduce the validity (see Line 1 in Lst.4) and reason for any asynchronous exit (Line 4 – 5). When a hardware exception causes an exit from the enclave (Step 1 in Fig. 2(a)), the trusted hardware saves the exit information in the SSA and raises a hardware exception to the OS (Step 2). The OS converts the hardware exception to a signal.[1] Then, it identifies the enclave process that caused the exception and sends it a signal (Step 3). This signal is caught and handled by Intel SGX SDK's untrusted runtime. The signal handler converts the signal to the corresponding hardware exception and notifies the enclave with the exception information by entering the enclave (Step 4).[2]

SGX 1. The trusted runtime uses the hardware exception vector from the untrusted runtime to call the enclave application's exception handler. Therefore, SGX 1 enclaves that do not have hardware support to store the exit information are vulnerable to SIGY. Specifically, the OS can arbitrarily inject a signal to the enclave to cause an asynchronous exit. Then, the untrusted runtime enters the enclave with the hardware exception vector corresponding to the signal. The exception handling in the trusted runtime executes the enclave's exception handler (Step 5) without any filtering. Therefore, an attacker can use SIGY to asynchronously send signals to the enclave and trigger its exception handlers.

SGX 2. Enclaves that execute in SGX 2 with the Intel SGX SDK are not vulnerable to SIGY. In SGX 2, the trusted runtime uses the exit information from the SSA to determine the validity and exception vector when performing exception handling. If the OS maliciously injects signals to the enclave, it causes an invalid exit (i.e., the hardware stores 0 in Line 4 in Lst. 3). When the trusted runtime checks the exit information, the guard check on Line 1 of Lst. 4 will fail. So the trusted runtime will discard the exception and will not execute the handler in the enclave (see Fig. 3(a)).

3.2 Open Enclave

Open Enclave is a widely used open-source SDK that allows developers to write custom applications to execute in SGX enclaves.

Filtering using exit information. To support exception handling, Open Enclave allows applications to register handlers for hardware exceptions. An instruction that raises a hardware exception in the

[1]OS does not raise a signal for page faults. It simply resumes the untrusted runtime
[2]The trusted runtime implements 2-level exception handling for asynchronous exits. We abstract this detail in our discussion.

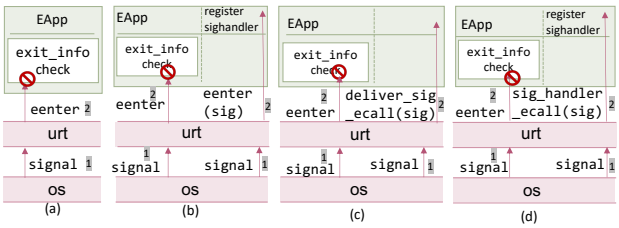


Figure 3: (a) Intel SGX SDK (b) Open enclave (c) Asylo (d) Teaclave SGX-SDK.

enclave causes an asynchronous exit and traps into the OS. The OS and the untrusted runtime then propagate this exception to the enclave's trusted runtime. Open Enclave's exception handling uses the exit information that the hardware stores in the SSA like in Lst. 4. If the OS attacks an enclave by maliciously injecting a signal, this will cause an asynchronous exit from the enclave. However, the hardware will indicate that this is an invalid exit in the exit information stored in the SSA. When the trusted runtime handles this exception, it will detect the attack and not execute the exception handler (LHS of Fig. 3(b)).

Supporting inter-thread signals. The hardware exception interface allows applications to only register handlers for hardware exception events. Open Enclave does not allow enclaves to register handlers for other signals (e.g., `sigusr1`, `sighup`) which limits the expressiveness of applications that can execute with Open Enclave. To improve expressiveness, Open Enclave introduces a separate mechanism to allow enclave threads to send signals to each other. This allows enclave threads to explicitly enable signals from the host and register signal handlers (Line 2 and Line 3 in Lst. 5). With this, the threads can send signals (Line 4) to other enclave threads that are routed through the trusted runtime.

```
1 ...
2 oe_add_vectored_exception_handler(false, sigusr_handle)
3 //enable signal
4 oe_sgx_td_register_host_signal(thread, SIGUSR1)
5 //do ocall
6 host_send_interrupt(target_thread, SIGUSR1) ...
```

Listing 5: Enable, register, and send a signal in Open Enclave.

When one enclave thread wants to send a signal (e.g., `sigusr1`) to the target enclave thread, it invokes the trusted runtime which performs an ocall (Step 2 in Fig. 2(b)). In the ocall context, the untrusted runtime raises a signal to the target enclave thread using the `tkill` system call. When the target enclave thread is resumed through `eenter` (Step 5 in Fig. 2(b)), the trusted runtime calls the target enclave thread's signal handler (Step 6 in Fig. 2(b)).

Attacking Open Enclave. When threads send signals to each other, the hardware exit information cannot be used to determine if the signals are legitimate. Specifically, the exit information stored in the SSA is only useful to determine the legitimacy of hardware exceptions and not explicit exits caused by sending signals. Therefore, the enclave cannot validate if the signal was legitimately raised by one of its threads or if it was maliciously injected by untrusted software. We can use the signal injection interface to compromise the security of Open Enclave using Sigy. Concretely, untrusted software like the OS can directly send signals to enclave threads. This

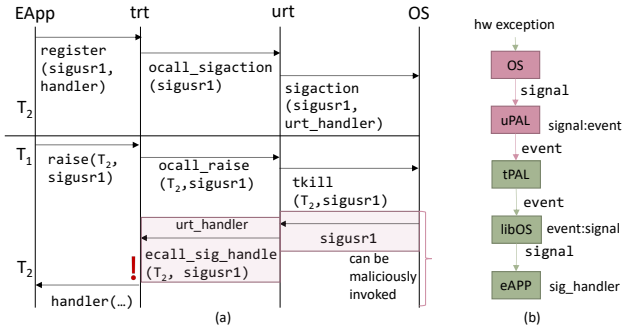


Figure 4: (a) Signal support in Teaclave. T_1 and T_2 are threads of the same enclave process. Black: Normal operation with ocall and ecall interfaces in Teaclave. Pink: Interfaces that can be maliciously invoked. (b) Hardware exception/signal handling in Gramine.

causes the untrusted runtime to enter the enclave with the signal. Because the trusted runtime does not validate the source of this signal, it will invoke the enclaves signal handler (RHS of Fig. 3(b)). Therefore, an attacker can arbitrarily execute the enclave's signal handler using Sigy.

3.3 Teaclave SGX-SDK

Teaclave SGX-SDK (Teaclave for short) enables developers to write programs in Rust and execute them in enclaves. It is based on Intel SGX SDK and implements different Rust libraries to ease enclave application development (i.e., standard library functionality).

Hardware exceptions. For hardware exceptions, it relies on the Intel SGX SDK's exception handling mechanism. The trusted runtime of Intel SGX SDK detects and discards all maliciously injected exceptions by checking the exit information in the SSA as discussed in Sec. 3.1. Hence, this interface cannot be used to compromise Teaclave enclaves using Sigy.

Signal support. Next, we analyze Teaclave's support that enables enclave threads to send signals to each other. Teaclave includes wrappers for Rust signal libraries. This allows enclave threads to register handlers and send signals to each other. To enable this functionality, Teaclave introduces a new ocall that its Rust library wrapper invokes as shown in Fig. 4(a). The ocall function in Teaclave's untrusted runtime performs `sigaction` and `raise` libc calls Teaclave also adds a public ecall (`t_signal_handler_ecall`) to be invoked by the untrusted runtime to forward signals from one enclave thread to another.

During normal operation, an enclave thread can raise a signal to another thread using Teaclave's signal library. This is translated into an ocall (`u_raise_ocall`) by the library which transfers control to Teaclave's untrusted runtime. The untrusted runtime sends a signal to the target thread using `tkill`. When the OS sends the signal to the target thread, Teaclave's untrusted runtime invokes the ecall (`t_signal_handler_ecall`) to handle the signal in the enclave. The ecall implementation in Teaclave's trusted runtime then triggers the signal handler registered in the target thread.

Attacking Teaclave. SIGY needs the ability to arbitrarily inject signals that trigger the signal handler in the enclave. To gain this capability, SIGY can abuse the `t_signal_handler_ecall` interface. If an enclave registers signal handlers, SIGY can arbitrarily inject signals into the untrusted runtime and trigger this ecall. Alternatively, because the untrusted runtime is attacker-controlled, SIGY could directly invoke this ecall without the need for signal injection. In both cases, the enclave application will always execute the signal handler (Fig.4(a)). Therefore, an attacker can use SIGY to trigger computation in the enclave. Note that the trusted runtime does not have any mechanism to distinguish legitimate signals (e.g., from one enclave thread to another) from those that are maliciously injected by the OS. Further, the `t_signal_handler_ecall` is a public root ecall (i.e., invoked from any untrusted software) for functionality such that it can be invoked by the untrusted runtime to forward signals to the enclave. Therefore, Teaclave's signal handling design makes it vulnerable to SIGY (RHS in Fig. 3(d)).

3.4 Asylo

Asylo is an open-source framework that provides a POSIX interface to enable enclave application development. It implements wrappers for POSIX functions that invoke ocalls to interact with the untrusted OS. Asylo uses the Intel SGX SDK and preserves the hardware exception handling interface from the SGX SDK. This interface checks the exit information in the SSA and discards maliciously injected exceptions. So, this interface is not vulnerable to SIGY.

Signal support. Asylo introduces a new signal handling mechanism to allow enclave applications to register and handle signals. In Asylo, this signal handling interface can be used by the enclave application to register all POSIX signals. To support signals, Asylo introduces a new ocall (`ocall_enc_untrusted_register_signal_handler`) to register a signal handler and an ecall (`ecall_deliver_signal`) to propagate the signal from the OS to the enclave. When the OS sends a signal, the untrusted runtime invokes the ecall and transfers the execution to the enclave. The enclave's trusted runtime uses the signal from the ecall's parameters, looks up the corresponding handler that was registered, and invokes it.

Attacking Asylo. The ecall used to deliver the signal to the enclave is a public root ecall. Therefore, SIGY can use this interface to attack enclaves in Asylo. Concretely, when the OS sends a signal to the enclave, the untrusted runtime invokes the ecall to enter the enclave's trusted runtime (RHS Fig. 3(c)) The trusted runtime executes the enclave's signal handler without any additional checks.

In summary, our analysis shows that all SDKs use the exit information that the hardware stores to handle hardware exceptions. However, they introduce new mechanisms to support signal handling between threads in the enclaves which render 3 out of the 4 SDKs vulnerable to SIGY.

4 Faking signals in Library OSes

Background. Library OSes support rich exception handling and signal interfaces for enclave applications. Unlike the SDKs, they also implement mechanisms to execute multi-process applications by adding support for calls like `fork`, `vfork`, and `execv`. In some library OSes (e.g., Gramine, Scone) calls to these functions spawn new enclave processes (Fig.5(b)). Like the OS, the library OSes also add support to send signals from one enclave process to another. For this, they route the signals through the untrusted operating system and runtime (Fig.5(b)). Other library OSes support multi-processing, by implementing process abstractions inside the enclave (e.g., Occlum). Such library OSes route inter-process signals inside the enclave itself and do not have to exit or use the untrusted runtime and OS to propagate signals (Fig.5(c)).

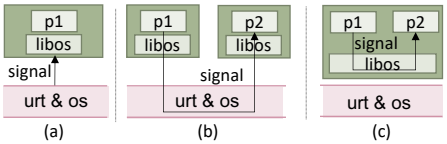


Figure 5: Signal propagation with library OSes. (a) OS or untrusted runtime sends signal to enclave process. (b) One enclave process sends signal to another enclave process through the untrusted runtime and OS. (c) LibOS creates a process abstraction such that 2 processes run in the same enclave. These processes can send signals to each other via the LibOS.

4.1 Gramine

Gramine is an open-source library OS that enables executing unmodified multi-process applications in SGX enclaves. It has 3 parts—a library OS, a Platform Adaptation Layer (PAL), and a patched C standard library. For simplicity, we refer to the part of the PAL that runs outside the enclave as untrusted PAL (uPAL) and the part that executes inside the enclave as (tPAL).

Gramine's signal support. Crucially for SIGY, Gramine's C standard library allows support for signal handling in enclave applications. Specifically, the wrappers for C's `sigaction` calls allow the application to register handlers for signals. When the enclave triggers a hardware exception, the untrusted OS traps on it and raises a signal that is caught by the uPAL. The uPAL's signal handling function (`handle_sync_signal`) converts the signal into a corresponding Gramine-specific PAL event. For example, the uPAL maps `sigfpe` to `pal_event_arithmetic_error`. Then, it invokes tPAL using the `sgx_raise` (Fig.4(b)) who forwards the event to the library OS. The library OS converts the event to a signal, creates the signal struct (`siginfo_t`) like in Linux, and raises the signal to the enclave application. This finally results in the enclave executing its signal handler.

Attacking Gramine. The trusted PAL does not check if the event was raised because of a real hardware exception in the enclave or by a signal injected by the untrusted OS, thus making Gramine vulnerable to SIGY. Specifically, the untrusted OS can inject a signal arbitrarily to a Gramine application (see Fig. 5(a)). This signal is converted to an event in the uPAL and forwarded to the tPAL which eventually executes the enclave application's handler.

Gramine supports executing multi-process applications. It allows enclave processes to send software-generated signals to each other using a message-passing framework in the trusted PAL. Using this message-passing framework, the enclave's inter-process signals are not sent through the untrusted OS. Therefore, with SIGY we can only inject signals that map to hardware exceptions to compromise enclaves that execute with Gramine (Sec.5.1).

4.2 Scone and EnclaveOS

They are closed-source library OSes that support executing unmodified applications in SGX based on Intel SGX SDK. As we don't have the source code, we perform a black-box analysis of these library OSes. Specifically, we write a program that registers handlers for all signals (Sec.5.1) and observe which handlers are executed. For both Scone [32] and EnclaveOS [16], our analysis shows that we can arbitrarily inject most signals from the untrusted OS to the enclaves (Sec.5.1). This successfully triggers signal handlers in the enclave making them vulnerable to Sigy. Scone forwards all signals from the OS to the enclaves. For EnclaveOS, our analysis shows that `sigusr2` is reserved for library OS-specific operations, and enclave applications cannot register signal handlers for these. Besides this, we observe that all other signals from the OS execute enclave handlers. Because these are closed-source we cannot comprehensively analyze their behavior. We suspect that they allow all signals because they introduce a new ecall which enables the OS to send signals to the enclaves.

4.3 Occlum

Occlum is an open-source library OS that enables multi-process applications by executing them in a single enclave [58]. As shown in Fig. 5(c), Occlum operates under a threat model that is not typical of SGX enclaves. Specifically, it assumes untrusted co-resident processes inside an enclave [59]. It performs inter-process isolation using Intel MPX for software-fault isolation. Crucially, it allows application processes in the enclave to send signals to each other. In this threat model that assumes untrusted processes inside the enclave, Sigy can inject signals from one process to another. For Occlum, we do not need the untrusted OS to inject signals into the enclave. Instead, we can use malicious attacker-controlled processes to send signals to the victim process in the same enclave. The library OS in Occlum does not filter such injections and simply forwards the signals from the attacker-controlled process to the victim process. For completeness, we checked if the OS can inject signals into the enclave to trigger signal handlers. We report that while we can inject signals, these signals do not result in the enclave executing its signal handlers. Instead, the library OS invokes the kernel's default signal handling which always crashes the process. We observe that, if Occlum did not assume untrusted co-resident processes inside the enclave, it would not be vulnerable to Sigy as it does not expose any signal interfaces to the untrusted OS.

4.4 Other Runtimes

Runtimes without signal support. Of the runtimes and library OSes that we surveyed, the language runtimes GoTEE, EGo, and Enarx are not vulnerable to Sigy because they do not support signal handling [14, 15, 41]. Similarly, EdgelessRT (a library OS) and RustEDP (an SDK) also do not support signal handling in enclaves [13, 25]. Therefore, these also cannot be attacked using Sigy.
Runtimes with limited signal support. MystikOS is a library OS that allows multiple application processes to execute in a single enclave process [22]. Similar to Occlum, MystikOS implements mechanisms to allow application processes inside the enclave to send signals to each other. All signals are routed through the library OS and do not leave the enclave. Therefore, it doesn't expose interfaces for the untrusted software OS to inject signals. Unlike Occlum, it assumes that processes in the enclave mutually trust each other. So, while processes can send signals to each other, this cannot be abused by an attacker to compromise the enclave's execution. Therefore, MystikOS is not vulnerable to Sigy.

5 Which signals can we inject?

In Sec. 3 and Sec. 4, we discussed SDKs and library OSes that are vulnerable to Sigy. Next, we systematically analyze each of the vulnerable SDKs and library OSes to determine which signal handlers are potentially of interest to the attacker. Then, we examine programming language support for signals to determine if applications written in them might be vulnerable to Sigy.

5.1 SDK and Library OS

To determine which signals are of interest to Sigy in vulnerable SDKs and library OSes, we write test applications that register handlers for all signals in the range 1-31. We execute our test applications in each of the 7 vulnerable SDKs and library OSes. Then, for each run of the test application we inject a signal from the OS and check if the application: (a) executes the registered handler, (b) crashes, or (c) has no effect as shown in Appx. A.1.
Our findings. Our experiments show that when our test applications invoke the `rt_sigaction` system call and try to register handlers for `sigkill` and `sigstp`, the system call fails and the application always crashes for all SDKs and library OSes. This is because these signals are reserved by the operating system for process management (e.g., `sigkill` is used to force kill the process). Occlum and Scone allow handlers for all other signals except `sigkill` and `sigstp` to be registered and we report that their signal handlers are executed. EnclaveOS reserves `sigusr2` for library OS operations but forwards all other signals except `sigkill` and `sigstp` to the application and executes its registered handlers. Notably, Teaclave does not allow injecting the signals that it expects to get from hardware exceptions (`sigill`, `sigtrap`, `sigbus`, `sigfpe`, and `sigsegv`) through the signal interface used for inter-thread communication. In contrast, Gramine only allows applications to use signals that map to a limited set of hardware exceptions (e.g., `sigill`, `sigbus`, `sigfpe`, and `sigsegv`). Open Enclave only allows a small subset of signals (`sigill`, `sigbus`, `sigfpe`, and `sigsegv`) to be forwarded to the enclave through its signal handling interface.

5.2 Programming Languages

The SDKs and library OSes that we analyzed allow developers to execute programs written in different languages in the enclaves. While some of them provide support for specific languages (e.g., Teaclave is used to develop and run Rust programs), others support a wide range of languages (e.g., Scone supports executing programs written in C, C++, Java, Python, Rust, Go, JavaScript).[3] Therefore, we analyzed 9 popular programming languages (see Appx. A.2) to check if they allow applications to register and execute signal handlers. To do this, we wrote programs in each of the languages to register signal handlers for all signals from 1 to 31. We report if the corresponding signal handlers were executed for each programming language in Appx. A.2. The language and signal pairs

[3]We analyze server-side JS, specifically Node.js

that execute the registered signal handlers are of specific interest for Sigy. An attacker can use these signals to compromise applications written in the corresponding language. For completeness, we also analyzed which signal handlers the programming language standard libraries or interpreters register by default i.e., when the program does not register any signal. We report our findings in Appx. A.2. Our experiments show that signal number 10 (`sigusr1`) and signal number 3 (`sigquit`) in Julia and Java respectively write debug logs to `stderr` and `stdout`. In Sigy, `stdout` and `stderr` are not accessible to the attacker and are therefore not interesting. Further, Go starts profiling on signal 27 (`sigprof`), however this does not lead to any changes in the program's global state. Interestingly, NodeJS starts a debug server on signal 10 (`sigusr1`). We exploit this behavior to demonstrate Sigy on a NodeJS application in Sec. 6.2. Finally, we find that all other signal handlers that programming language runtimes register do not perform any computation that changes the program's global state and simply crash the application. These handlers are therefore not interesting to Sigy.

Our findings. WebAssembly system interface (WASI), the standard interface definition for WebAssembly, does not yet support signals [29]. So, programs compiled to Wasm binaries which use WASI cannot register or execute signal handlers. For the other languages we broadly classify the signal support into 2 categories: explicit and implicit support (Appx.A.2). Programming languages that offer explicit signal support allow applications to register signal handlers for specific signals (e.g., using the `signal(signum,handler)` libc function) which are executed when the OS sends a signal to the enclave. Therefore, Sigy can asynchronously trigger these handlers when enclave programs are written in these languages.

Languages which provide implicit support for signals (e.g., Java, Julia) register signal handlers directly with the OS. They do not provide any constructs for the programs to register signal-specific handlers. Instead, the language runtime converts signals into software exceptions. These software exceptions are caught and handled by the applications. For example, Java converts `sigfpe` to `ArithmeticException` and forwards it to the application. The application can execute custom handling for the exception in its `catch` block (e.g., Line 9 in Fig.1). Similarly, Julia converts `sigfpe` to `DivideError` which the application can catch and handle. Sigy can trigger these `catch` blocks to change the execution and data integrity of applications written in these programming languages.

Go uses `sigprof` for internal CPU profiling and so allows programs to register handlers for all other signals except `sigprof`. All other languages that provide explicit signal support, allow programs to register handlers for all signals between 1-31 except 9 and 19 (i.e, `sigkill` and `sigstp`). Note that, this is a direct consequence of the kernel blocking all requests to register handlers for `sigkill` and `sigstp`. On the other hand, Java only converts `sigfpe` to `ArithmeticException`. Therefore, Sigy can only use `sigfpe` to compromise programs written in Java.

In summary, of the 9 languages that we analyzed, we found that 1 does not support any signal handling, 2 only supports implicit signal handling, and the 6 others support explicit signal handling. So, programs written in any of the 8 languages that provide signal support may be vulnerable to Sigy and should be reanalyzed.

6 Case studies

We confirmed our findings from Sec. 3-Sec.5 using hand-coded enclaves. Then, we surveyed publicly available SGX enclave applications from Intel, and applications ported to library OSes and runtimes to define our case studies (Appx.A.4). Here, we first discuss end-to-end case studies and explain how Sigy can be used to compromise enclave execution. Then, we build a framework and demonstrate the feasibility of Sigy's signal injection architecture.

6.1 Nginx

We surveyed widely used open-source webservers optimized for high uptime and found that many of them use signals to upgrade configuration without have to restart the server. Specifically, httpd and Nginx use `sigusr1`, and squid proxy server uses `sighup` to upgrade the server's configuration. We choose to demonstrate Sigy on Nginx as it is ported by library OSes (Gramine and Scone) to run in SGX enclaves.

By default, Nginx allows a system administrator to upgrade its configuration and binaries using signals (`sighup` and `sigusr1`) without degrading the uptime of the server. For Sigy, this gives the attacker the ability to change the configuration and binary of the server by injecting `sighup` and `sigusr1`. Gramine, Scone, and EnclaveOS port Nginx to run in SGX enclaves. Our analysis from Sec. 5.1 shows that Sigy cannot inject `sigusr1` into enclaves running with Gramine. Further, the port of Nginx by EnclaveOS reduces the functionality of Nginx and does not allow administrators to refresh configuration files without restarting the server. Like EnclaveOS, Scone's encrypted file-system also limits the administrator's capabilities to refresh configuration files without restarting the server. To support this functionality Scone's encrypted file-system will need to be modified. So, we demonstrate Sigy using Scone with the encrypted filed-system turned off.

Benign Nginx configuration and binary upgrade. Consider an Nginx webserver executing inside an enclave (t_0 in Fig. 6). At time t_1, the Nginx administrator replaces the configuration file with an updated file and sends `sighup` to the Nginx process. On receiving this, the Nginx process (at time t_2) reads the new configuration file and begins to use it. Similarly, the administrator upgrades the binary of the server by writing a new file and sending `sigusr1` at time t_3 which is read and used by the server at t_4. Note that, in an enclave setting, the administrator encrypts the configuration and binary files before writing them to the OS accessible file-system. These files are only decrypted inside the enclave.

Attacking Nginx with Sigy. Because the administrator encrypts the configuration and binary files before writing them into the file-system, the OS cannot directly manipulate them to compromise Nginx. However, a malicious OS can record the encrypted blobs of configuration and binary files when the administrator writes them. The OS can also observe the new configuration and binary files when the administrator sends the signals to upgrade them in the enclave. Once the administrator has finished the configuration and binary upgrade, the OS uses Sigy to manipulate the state of the Nginx server.

Specifically, at t_5, the OS replaces the new configuration file with the old file it captured and sends malicious `sighup` to the enclave. The enclave reads in this configuration and updates the

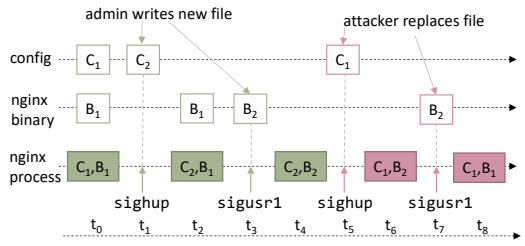


Figure 6: Maliciously injecting signals to Nginx to trigger insecure states (pink). C: Configuration, B: Binary.

Nginx process (t_6). This ensures that the Nginx process uses an old configuration with a new binary. Furthermore, the OS can also replace the new binary file with an old one that it captured and send `sigusr1` (t_7). With this, the OS has successfully used SIGY to restore the Nginx process to the state that it was in at time t_0 before the administrator performed the upgrades. This exploit will undo any performance and security improvements provided by the new binary and configuration and can be used by the OS to bring back old security bugs (e.g., forcing the enclave to use an Nginx version before v1.23.2 which patched critical issues [11]).

6.2 Node.js Server

Node.js is an open-source JavaScript runtime used for server-side scripting. By default, a Node.js server starts a debug web socket on localhost (127.0.0.1) when it receives `sigusr1`, even if the server was started without debugging enabled. To demonstrate SIGY, we use Scone's Node.js port that executes an HTTPs server in an enclave. When the server is up and running, the malicious OS sends `sigusr1` to the Node.js server which opens a debug web socket. Using this web socket, the attacker can leak the server's memory and inject arbitrary code. In our exploit, we attach Google Chrome browser's debugger to the web socket in `chrome://inspect/#devices`. Then, to demonstrate an attack, we use Google Chrome's debugger to leak the RSA private keys used for TLS by the HTTPs server. We show that this attack needs only 1 injection of `sigusr1`. This attack is feasible on production enclaves and does not need the enclave to be in debug mode. Further, it does not depend on what a developer implements in the Node.js server but instead relies on Node.js's feature to open a debug server. Note that, a malicious network adversary cannot do this exploit (e.g., by sending a socket request for the debug socket to the Node.js server).

6.3 Multi-Normal Distribution

We demonstrate SIGY on a Java application, inspired by Heckler, to show how implicit signal support in programming languages make applications vulnerable [57]. JSAT is a statistical analysis tool for machine learning applications in Java. JSAT implements a `MultiVariateNormal` class that can be used to create a Multivariate Gaussian distribution. The class implements a function that updates the mean and covariance of the distribution using the `setUsingData` function as shown in Lst. 6. If data added to the dataset causes an `ArithmeticException`, `setUsingData` discards the data and reverts the mean to the original value.

```
1 public boolean setUsingData(List dataSet) {
2  Vec origMean = this.mean;
3  try { //can overflow
4   Vec newMean = meanVector(dataSet);
5   Matrix covarianc = covarianceMatrix(newMean,dataSet);
6   this.mean = newMean;
7   setCovariance(covarianc);
8  } catch(ArithmeticException ex) {
9   this.mean = origMean; } }
```

Listing 6: JSAT mean and covariance.

As noted in Sec. 5.1 and Sec. 5.2: (a) the Java runtime converts `sigfpe` to `ArithmeticException` that can be caught and handled by Java applications, and (b) Gramine forwards `sigfpe` to the enclaves. Therefore, we use SIGY to trigger expressive application logic in JSAT executing in Gramine by injecting `sigfpe`. We use one of the tests in JSAT to train a Learning Vector Quantization (LVQ) with Multivariate Gaussian distribution as a local classifier. This setup calls the `setUsingData` function during the training process of the classifier. We use SIGY to inject `sigfpe` every time the program executes Lines 4 – 8 in Lst. 6. In our attack, we inject `sigfpe` 240 times and drop the error rate of the classifier from nearly 0% to 66%. Our attack adds an additional overhead of overhead of 3.4 seconds (3.34×) to the training. Therefore, SIGY can be used to bias the classifier and consequently any inference that a user might execute on the model that it builds. We choose Java instead of Julia for our case study because library OSes support running Java applications in enclaves. However, if library OSes support Julia, a similar attack would be possible on Julia's text analysis framework [57].

6.4 Synthetic Example: Multi-Layer Perceptron

To demonstrate the feasibility of SIGY when an application requires a large number of signal injections we build a signal injection framework with sgx-step and Gramine. To test the robustness of our framework, we identify an application that requires a large number of signal injections (on the order of 10^8). To start with this synthetic example, we used an open-source implementation of a multi-layer perceptron written in C [21]. This library uses `tanh` as an activation function which is called over 10^8 times during the training process. Mathematically, $tanh(x)$ tends to 1 when x tends to ∞ which occurs in the `tanh` function if the input overflows.

```
1 jmp_buf buf;
2 void sigfpe_handler(int signum)
3     longjmp(buf, 1);
4 void tanh(int input[...]) ...
5   for (i = 0; i < n; i++){
6     if (sigsetjmp(buf, 1))
7       output[i+1] = 1;                // on SIGFPE
8     else
9       output[i+1] = tanh(input[i]);  // no overflow
```

Listing 7: Tanh activation function for MLP.

To demonstrate our signal injection framework (Sec.7), we introduce a signal handler to the tanh function to handle overflows as shown in Lst. 7. Note that, unlike our other case-studies in this section, this is only a synthetic example where we introduce a signal handler for ease of explanation. In a real attack setting, an attacker in SIGY does not have this capability.

We train the MLP model with 3 hidden layers and up to 6 units in each layer (tanh activation function in hidden layers, sigmoid in output layer). Our training consists of 2000 epochs and 1096 training samples from the Banknote data set [10].

Using SIGY, if we inject `sigfpe` when execution is in between Lines 7-9, it will trigger the signal handler and set `output[i+1]` to 1 (Line 7). We inject `sigfpe` into the `tanh` function $1,8636 \times 10^7$ times during the training of the neural network. Without SIGY, the training achieves an accuracy of 97.09%. With SIGY, the accuracy drops to 59.27%. Further, SIGY adds 970 seconds (53.7×) overhead to the training process. This large overhead is primarily because of the asynchronous exits that sgx-step triggers and the signal handling in the enclave on every execution of the `tanh` function which the training invokes $\approx 10^8$ times in total.

7 Proof of Concept Exploits

We perform our experiments on machine setups (Appx.A.5). We set up the laptop with sgx-step compatible configurations. The server has a kernel with an in-tree driver. Both machines have hardware support to store exit information in the SSA.

Sending Signals. We send signals to enclaves with the `util-linux` program `kill` [20]. The utility program is a wrapper around the `kill` syscall in Linux. The `kill` syscall populates the `siginfo_t` struct to indicate that the signal originated from user-space. Some of our experiments (e.g., injecting to Java applications) require the struct to indicate that the signal is the result of an integer division by zero error (`fpe_intdiv`) or floating point overflow (`fpe_fltovf`). To do this, we implement a kernel module that correctly populates this information and sends it to the user-space process (98 LoC).

Nginx. Scone is closed-source and only supports running Nginx servers in the paid versions. Therefore, to perform our experiments we ported Nginx versions 1.22.1 and 1.24.0 to execute in a Scone v5.8.0 enclave [32]. To do this, we remove some system calls (e.g., `fcntl`) that Scone does not support. Further, we adapted Nginx to correctly propagate Scone configurations when spawning new processes (e.g., by invoking the `execve` system call). Note that our modifications do not change the functional behavior of Nginx. We build Nginx with the `select` event method and a minimal configuration to serve HTTP websites.

Node.js. We run a Node.js v10.14.1 web server (28 LoC) with standard TLS libraries such as express and https using Scone's v5.8.0 publicly supported Node.js port and configuration [27]. After sending `sigusr1` to Node.js, we use Chrome v120.0.6099.224 Developer tools to connect to the server, dump its memory, and extract the RSA private key used for TLS.

MLP (Synthetic) and JSAT. To demonstrate SIGY on our synthetic MLP training (Sec.6.4) we build a signal injection framework using sgx-step v1.5.0 [63] and Gramine commit `211ec447e` [17]. First, as a preparatory step, we run the training of the model in Gramine in debug mode, to identify the instruction pages of the `tanh` function. To bias the training process, we must inject `sigfpe` every time the `tanh` function is invoked. When the `tanh` function starts executing, sgx-step generates an asynchronous exit using timer interrupts and page faults. We should inject `sigfpe` on this event but this is not straightforward. On an asynchronous exit, control switches from the enclave to Gramine's uPAL. If we inject `sigfpe` when uPAL is executing, Gramine will crash. First, we change Gramine's uPAL which executes outside the enclave and can be attacker-controlled (with 212 LoC) to ignore our signals to avoid it from crashing. Next, we use our framework to ensure that we inject `sigfpe` only when the enclave has resumed executing the `tanh` function. For this, we use 2 threads. The main thread executes the enclave and handles the asynchronous exit on each `tanh` invocation. Then, a worker thread injects `sigfpe` to the enclave after it is resumed. Java applications require more profiling than ahead-of-time compiled languages. For simplicity, we perform our experiments on JSAT by ensuring that our target function (Lst.6) waits for an `ArithmeticException`.

8 Potential Defenses

Current signal handling mechanisms render SDKs and library OSes vulnerable to SIGY. We propose techniques to potentially address the root cause of these issues. Then, we consider orthogonal protection techniques that can be used to diminish per-application SIGY impact.

8.1 Detecting Fake Signals

Hardware exceptions in SDKs. All the SDKs that we analyzed implement a hardware exception handling interface that checks the exit information before invoking application-registered exception handlers as shown in Fig. 2(a). Except for Intel SGX SDK, all of them introduce additional mechanisms to support enclave threads to send and receive signals. This makes the SDKs vulnerable to SIGY (Fig.2(b)) and should be hardened. First, they should not accept signals from the OS that usually are a result of hardware exceptions (e.g., `sigfpe`, `sigsegv`) as these signals should be sent through the hardware exception handling interface and never through the signal handling interface for custom signals.

Protecting inter-thread signals in SDKs. To prevent SIGY, the SDKs should introduce a mechanism to check if a signal that a thread receives was in fact raised by an enclave thread. For example, the trusted runtime can set up a protected shared memory between the enclave threads. Then, to indicate that a signal was raised by an enclave thread, the trusted runtime can write the signal number and target threadID in the shared memory region and only then exit the enclave with an ocall. Further, when the untrusted runtime enters the enclave with the signal, the trusted runtime in the target thread can look up the shared memory region to check the legitimacy of the signal. This defense breaks functionality of applications that require signal injection from outside the enclave (e.g., upgrading Nginx configuration). If enclaves need signals from outside that are not raised for hardware exceptions for functionality, there is no mechanism to defend them against SIGY.

Hardware exceptions in library OSes. Gramine only supports signals that are raised because of hardware exceptions (Sec.4.1). To stop SIGY, Gramine's exception handling can be enhanced to use the exit information that the hardware stores in the SSA. Gramine has used this technique in a patch which stops SIGY [7]. Unlike Gramine, other library OSes support signals that are both raised because of hardware exceptions (e.g., `sigfpe`) and explicitly by the processes (e.g., `sigusr1`, `sighup`). Currently, library OSes do not distinguish between signals from hardware exceptions and signals that originate from within the application in the enclave. First, their signal handling logic should be separated for these two cases. Then, for the first case of hardware exceptions, the library OSes can filter illegitimate injections using the exit information stored by the hardware. Next, we discuss mechanisms to protect against injections of other signals that processes originate.

Interprocess-signals in library OSes. Our analysis in Sec. 4 shows that library OSes that create distinct enclaves for application processes route the signal through the untrusted runtime and OS which makes them susceptible to SIGY. Protecting this interface is challenging as enclave processes need a mechanism to check if the signal is from another trusted enclave process. SGX does not support setting up shared memory regions between multiple enclave processes. Therefore, a method similar to the one that we outlined for signals between threads is not straightforward to implement between enclave processes. To protect against SIGY, the library OSes should implement a message passing framework (e.g., using local attestation to set up a trusted channel) to communicate the signal information to the target trusted processes which their library OS looks up to check the signal's legitimacy. Gramine implements a similar message-passing framework in its trusted PAL which allows enclave processes to send software-generated signals to each other.

Occlum is vulnerable to SIGY because it assumes a threat model where untrusted processes reside within the same enclave. While setting up a shared memory region to communicate the legitimacy of signals between processes in Occlum is easier, establishing the trust relationship between the processes is more challenging. To this end, Occlum should introduce a notion of attestation across process groups inside the same enclave. Currently, all processes inside the enclave are mutually distrusting so this would be a fundamental architectural change to Occlum. With this notion, Occlum can use a message-passing mechanism to protect against SIGY. Finally, library OSes should not accept signals from untrusted software. If such signals that are not a result of hardware exceptions are required for functionality then there is no defense against SIGY.

8.2 Limiting SIGY Impact

SIGY relies on the fact that the signal is propagated to the enclave application by the runtime or library OS. Once the signal is sent to the application, it will execute the signal handler in the enclave which compromises the enclave's execution integrity. To amplify the effect of the signal handler, the attacker might need other capabilities (e.g., replace encrypted blobs of files for Nginx, connect to a port for Node.js). If there are orthogonal protections like file-system rollback prevention, network firewalls, or dynamic attestation in place, these end-to-end attacks will be stopped. Of the library OSes we surveyed, only Gramine and Scone provide protections for filesystems and network. Gramine has filesystem protections to add trusted (i.e., hashed), and encrypted files to the enclave, but does not provide rollback protection. Scone paper does not discuss any port or protocol filtering in network protection for enclaves; when filesystem protection is enabled it ensures that the files are encrypted, rollback, and integrity protected. Next, we checked publicly available Scone versions—its network shields can be configured during enclave creation with a whitelist of allowed ports. However, this network shield is not activated by default allowing SIGY to compromise Node.js. Thus, developers should consider the new attack surface introduced by SIGY when configuring enclaves. Lastly, orthogonal protections cannot stop SIGY attacks which exploit `sigfpe` and compromise enclaves as these attacks directly affect enclave memory and do not need any external subsystem (e.g., network, files). So, library OSes should build comprehensive defenses for this new attack surface.

9 Related work

Malicious synchronous interfaces. Ports et al. comprehensively analyze the threats to applications from an untrusted OS [53] in the context of Overshadow [34], and note that because the untrusted OS manages the signals it can maliciously alter them. However, the analysis focuses on synchronous attacks from the OS that redirect signals or send bad return values. Iago builds on this, where the attacker *synchronously* manipulates the return values for system calls when the enclave makes *explicit* requests [33]. These bad return values can be used to trick the enclave into performing unintended computations. Iago demonstrates how the limited sanitization in the system call interface between an enclave and OS can be used to synchronously alter enclave execution. This synchronous interface has been studied in the context of SGX enclaves [30, 44, 48, 61].

Malicious asynchronous interfaces. Previous works have demonstrated using *asynchronous* timer interrupts and page faults at arbitrary points of the enclave's execution [63]. The attacker manipulates the software to trigger these hardware events, which then trigger handlers that have *fixed* effects—timer interrupts and page faults cause an exit from the enclave, but do not execute any handlers in the enclave that change the program state. In contrast, to the best of our knowledge, SIGY is the first work on Intel SGX that injects signals and exceptions *asynchronously* at any point during the enclave's execution. Depending on enclave-specific logic, such arbitrary signal injection can induce *varied effects* depending on the injected signal and the enclave's logic for it. This allows SIGY to execute such signal handlers at any point during enclave execution to bring about changes to the enclave's state.

Bugs in enclave runtimes and library OSes. Previous works have studied and demonstrated attacks on buggy enclave-OS interface implementations (e.g., Application Binary Interface, Application Programming Interface) [48, 61]. AsyncShock and Game of Threads exploit synchronization bugs in multi-threaded enclave applications by interrupting the threads at specific points in their execution [67] or using race-conditions [55]. Similarly, SmashEx demonstrates attacks that use the lack of atomicity in signal handlers to compromise enclaves [39]. In constrast, SIGY does not rely on bugs in the runtimes, library OSes, or signal handlers.

Bugs in enclave applications. Prior works have used vulnerabilities in enclave application code to compromise it using code-reuse attacks [49]. SIGY does not rely on bugs in enclave application implementations. However, SIGY can be used to bring back old bugs. For example, by forcing Nginx to use old binaries, SIGY brings back vulnerabilities in older Nginx versions.

Detection tools. Several works have used fuzzing [36, 47, 70] and symbolic execution [30, 35, 48, 66] to detect vulnerabilities in enclaves. These detection tools can be used to analyze applications with the intention of finding other effects of signal handling in enclaves. Lefeuvre et al., investigate the fragility of interfaces for software compartmentalization and build a fuzzer to detect interface vulnerabilities but do not consider signals [40]. Beyond Intel SGX, SIGY's observations can be applied to other such research avenues.

Side-channels. Prior works have used timer interrupts and page faults as side-channels [43, 54, 64, 65, 69]. Similarly, others have leveraged other side-channels to compromise enclaves [42, 52]. Sgx-step is a framework that enables attackers to precisely single-step

enclaves [62]. While SIGY is not a side-channel attack, side-channels can be used to amplify SIGY's effects.

Kernel bugs due to interrupts and signals. Buggy implementation of signal handlers in the kernel can compromise the security of applications by inducing memory-corruption from race conditions in the handlers [6]. Prior works have analyzed the possibility of using buggy signal handlers , interrupt remapping, and kernel races to corrupt applications executing with trusted OSes [12, 50, 68]. Signal handlers have been known to have race condition bugs that can be exploited to compromise applications [6]. SIGY does not assume any buggy or racy signal handlers.

SIGY on Arm TEEs. Several interface attacks have been demonstrated on Arm TrustZone [46, 60]. Unlike SGX, TrustZone does not expose signal interfaces to the malicious OS and filters interrupts from the untrusted OS to the trusted applications [31]. So, SIGY cannot be used to attack TrustZone. Heckler shows that Arm CCA's VMs are not vulnerable to attacks using malicious interrupts to trigger signal handlers because of Arm's interrupt architecture [57]. Furthermore, in Arm CCA's VM setting, the guest OS is trusted and the CVM does not accept exceptions and signals from the untrusted hypervisor. Arm CCA is not vulnerable to SIGY.

10 Discussion

Finding vulnerable applications. For SIGY, an attacker needs the ability to detect whether the enclave application has a signal handler that affects the global state. To detect these handlers, an attacker can use publicly available documentation or manually analyze the source code. We manually analyzed publicly available applications to identify vulnerable signal handlers as shown in Appx. A.4. Of these, we chose applications (e.g., Nginx, Node.js) that were already ported to run in library OSes for our case-studies. Similarly, we manually surveyed Java applications on Github to find several potentially vulnerable applications (see Appx. A.4). We chose JSAT for our case-study as it was the most maintained.

Alternatively, if the attacker has the source code or an intermediate representation, they can use static or dynamic analysis techniques (e.g., taint analysis, symbolic execution) [30, 35, 48, 66] to identify global effects of signal handlers in enclave applications. In the absence of source-code, the attacker can detect these handlers by fuzzing the enclave binary. Concretely, the attacker can use fuzzing techniques to inject signals at random, guided by traditional coverage-metrics (e.g., instructions, blocks, edges) [51, 71, 72].

Ahoi attacks. There are three ways of delivering notifications to a user-level computation: interrupts, exceptions, and signals. Since Intel TDX and AMD SEV offer a VM abstraction, the attacker can only use interrupts as a notification mechanism. Specifically, when the hypervisor delivers an interrupt, the guest OS can either handle it in its kernel with an interrupt handler or convert it to an exception or signal for the user process currently executing on the vCPU. Recent works introduce a new family of attacks called Ahoi Attacks that use notifications to compromise hardware-based TEEs [56, 57]. SIGY is an instance of Ahoi attacks. Heckler and WeSee exploit CVMs provided by AMD-SEV and Intel TDX by injecting malicious interrupts from hypervisors [56, 57]. In particular, Heckler injects interrupts which are then delivered as signals to the victim user-level program. In contrast, SIGY investigates Intel SGX which is a user-level abstraction and shows that a malicious OS can deliver interrupts, exceptions, and signals to an enclave. When a CPU executing an enclave triggers a hardware interrupt (e.g., INT0 for divide by zero faults), Intel SGX hardware sets the SSA to indicate that the interrupt did indeed originate on the core. This way, the enclave software can check the SSA before executing the handler. Fortunately, our investigation shows that all current enclave SDKs (Intel SGX SDK, Open Enclave, Teaclave SGX-SDK) do perform the interrupt authenticity check before executing the handler. However, this covers only a fraction of the attack surface.

Detecting SIGY with AEX-Notify. SIGY injects signals into enclaves which cause asynchronous exits. One way to protect an enclave against SIGY is to detect if an asynchronous exit was caused by fake signal injections. Intel introduced a hardware extension called AEX-Notify to perform checks on reentry after an interrupt or exception [9]. AEX-Notify proposes a defense to prevent precise single-stepping of enclaves using the new hardware [37]. For this, they implement handlers for timer-interrupts that speed up the execution of the successive instructions. AEX-Notify cannot mitigate SIGY when the applications register handlers that only need to be invoked once (Sec.6.2 and Sec.6.1) to compromise the enclave. Specifically, the OS can send the signal when the enclave has legitimately exited (e.g., timer interrupt, page-fault) without AEX-Notify detecting it. This is because SIGY will not cause any asynchronous exits, let alone require any single-stepping.

On the other hand, our attacks that need to send signals when the enclave application is executing a specific set of instructions can be harder, but not impossible, to perform in the presence of AEX-Notify. However, this does not completely stop SIGY. Note that, for SIGY the attacker does not need to single-step the enclave. Instead, the attacker needs a mechanism to determine when the enclave is executing a set of instructions. Further, if the size of this instruction set is larger than the number of instructions executed between timer interrupts, there will always be a legitimate enclave exit for the OS to inject the signal. Then, when a genuine timer interrupt occurs in between these instructions, the attacker can inject the signal. For applications where this is not the case, the frequency of timer interrupts is controlled by the untrusted OS. So, the attacker can tune the frequency to any value to send signals to enclaves when the enclave executes the target instruction set.

11 Conclusion

SIGY demonstrates that a malicious OS can exploit Intel SGX enclaves by delivering malicious exceptions and signals to trick the enclave into executing handlers. Our analysis of various runtimes and library OSes shows that they are vulnerable to SIGY. Programming languages as well as native and ported enclave-bound programs that need exceptions and signal handling should consciously choose between functionality and security.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their feedback; and Mélisande Zonta-Roudes and Mark Kuhne for their comments to improve the paper. Thanks to the developers of Gramine, Scone, Open Enclave, Teaclave, Asylo, Occlum, and EnclaveOS for their discussion and engagement during the disclosure process.

References

[1] accessed 01.01.2025. DartRuntime. https://github.com/GGiry/DartRuntime/blob/9d452f1981493c61866b391c87534971b26c8207/rt/FiboAsDart.java#L34.

[2] accessed 01.01.2025. GovWay - API Gateway for Public Administration. https://github.com/link-it/govway/blob/2557c1d46c57222b4ddac70fce6e0a5e0b519e21/core/src/org/openspcoop2/pdd/core/token/parser/TokenUtils.java#L50.

[3] accessed 01.01.2025. jsjs: JavaScript runtime. https://github.com/forax/jsjs/blob/1ab2f488c5298c5110f2e06427e652393aaca273/src/com/github/forax/jsjs/Builtins.java#L72.

[4] accessed 01.01.2025. OSP: An Open Source OWL DL reasoner for Java. https://github.com/OpenSourcePhysics/osp/blob/0ab7fe7d15b8f1e14058c7876b5a86102950089c/src/org/opensourcephysics/numerics/LUPDecomposition.java#L75.

[5] accessed 01.01.2025. Pellet: An Open Source OWL DL reasoner for Java. https://github.com/stardog-union/pellet/blob/4c7d16bd1811ec04117fa4cd96ed592c6cfa956b/core/src/main/java/com/clarkparsia/pellet/datatypes/types/real/Rational.java#L229.

[6] accessed 17.04.2024. Unsafe function call from a signal handler . https://owasp.org/www-community/vulnerabilities/Unsafe_function_call_from_a_signal_handler.

[7] accessed 22.04.2024. [PAL/Linux-SGX] Cross-verify SW signals vs HW exceptions. https://github.com/gramineproject/gramine/commit/a390e33e16ed374a40de2344562a937f289be2e1.

[8] accessed 28.01.2024. Asylo Github. https://github.com/google/asylo.

[9] accessed 28.01.2024. Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function. https://www.intel.com/content/www/us/en/content-details/736463/white-paper-asynchronous-enclave-exit-notify-and-the-edeccssa-user-leaf-function.html.

[10] accessed 28.01.2024. Banknote Database. https://banknotedb.com.

[11] accessed 28.01.2024. CVE-2022-41741. https://nvd.nist.gov/vuln/detail/CVE-2022-41741.

[12] accessed 28.01.2024. Delivering Signals for Fun and Profit. https://lcamtuf.coredump.cx/signals.txt.

[13] accessed 28.01.2024. Edgelessrt Github. https://github.com/edgelesssys/edgelessrt.

[14] accessed 28.01.2024. Ego Github. https://github.com/edgelesssys/ego.

[15] accessed 28.01.2024. Enarx Github. https://github.com/enarx/enarx.

[16] accessed 28.01.2024. Fortanix Runtime Encryption® Platform. https://resources.fortanix.com/hubfs/Fortanix_RTE_Platform_Whitepaper.pdf.

[17] accessed 28.01.2024. Gramine Github. https://github.com/gramineproject/gramine/tree/211ec447ee69f16139520fc3a17c561c36a00943.

[18] accessed 28.01.2024. Intel SDK. https://github.com/intel/linux-sgx.

[19] accessed 28.01.2024. JSON Web Token (JWT). https://datatracker.ietf.org/doc/html/rfc7519.

[20] accessed 28.01.2024. kill(1) — Linux manual page. https://man7.org/linux/man-pages/man1/kill.1.html.

[21] accessed 28.01.2024. Multi Layer Perceptron in C. https://github.com/manoharmukku/multilayer-perceptron-in-c.

[22] accessed 28.01.2024. MystikOS Github. https://github.com/deislabs/mystikos.

[23] accessed 28.01.2024. Openenclave Github. https://github.com/openenclave/openenclave.

[24] accessed 28.01.2024. POSIX.1-2017. https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/.

[25] accessed 28.01.2024. Rust EDP Github. https://github.com/fortanix/rust-sgx/tree/master.

[26] accessed 28.01.2024. Scone. https://sconedocs.github.io.

[27] accessed 28.01.2024. Scone Nodejs. https://sconedocs.github.io/Nodejs.

[28] accessed 28.01.2024. Teaclave Github. https://github.com/apache/incubator-teaclave-sgx-sdk.

[29] accessed 28.01.2024. WASI: signal handling. https://github.com/WebAssembly/WASI/issues/166.

[30] Fritz Alder, Lesly-Ann Daniel, David Oswald, Frank Piessens, and Jo Van Bulck. 2024. Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes.

[31] ARM. 2021. Learn the Architecture: TrustZone for AArch64. https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/trustzone-in-the-processor.

[32] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[33] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 253–264.

[34] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. *SIGOPS Oper. Syst. Rev.* 42, 2 (March 2008), 2–13.

[35] 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. https://www.usenix.org/conference/usenixsecurity20/presentation/cloosters

[36] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. 2022. SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3147–3164. https://www.usenix.org/conference/usenixsecurity22/presentation/cloosters

[37] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. 2023. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 4051–4068. https://www.usenix.org/conference/usenixsecurity23/presentation/constable

[38] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[39] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. SmashEx: Smashing SGX Enclaves Using Exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*.

[40] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2023. Assessing the impact of interface vulnerabilities in compartmentalized software. In *NDSS*.

[41] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 571–586. http://www.usenix.org/conference/atc19/presentation/ghosn

[42] 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* (Belgrade, Serbia) *(EuroSec'17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[43] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. 2018. Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 108–114.

[44] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. *SIGPLAN Not.* 48, 4 (March 2013), 265–278.

[45] Intel. accessed 28.01.2024. Intel Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[46] Peipei Jiang, Qian Wang, Jianhao Cheng, Cong Wang, Lei Xu, Xinyu Wang, Yihao Wu, Xiaoyuan Li, and Kui Ren. 2023. Boomerang: Metadata-Private Messaging under Hardware Trust. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 877–899. https://www.usenix.org/conference/nsdi23/presentation/jiang

[47] Arslan Khan, Muqi Zou, Kyungtae Kim, Dongyan Xu, Antonio Bianchi, and Dave Jing Tian. 2023. Fuzzing SGX Enclaves via Host Program Mutations. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 472–488. https://doi.org/10.1109/EuroSP57164.2023.00035

[48] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 971–985. https://doi.org/10.1145/3373376.3378486

[49] 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 *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 523–539. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk

[50] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2363–2380. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan

[51] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[52] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX

Association, Boston, MA, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko
[53] Dan R. K. Ports and Tal Garfinkel. 2008. Towards application security on untrusted operating systems. In *Proceedings of the 3rd Conference on Hot Topics in Security* (San Jose, CA) *(HOTSEC'08)*. USENIX Association, USA, Article 1, 7 pages.
[54] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. 2021. Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 663–680. https://www.usenix.org/conference/usenixsecurity21/presentation/puddu
[55] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W. Fletcher. 2020. Game of Threads: Enabling Asynchronous Poisoning Attacks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 35–52. https://doi.org/10.1145/3373376.3378462
[56] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. 2024. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE S&P*.
[57] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security*.
[58] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 955–970. https://doi.org/10.1145/3373376.3378469
[59] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.
[60] Darius Suciu, Stephen McLaughlin, Laurent Simon, and Radu Sion. 2020. Horizontal Privilege Escalation in Trusted Applications. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/suciu
[61] 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. https://doi.org/10.1145/3319535.3363206
[62] Jo Van Bulck and Frank Piessens. 2023. SGX-Step: An Open-Source Framework for Precise Dissection and Practical Exploitation of Intel SGX Enclaves. (2023).
[63] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (Shanghai, China) *(SysTEX'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3152701.3152706
[64] 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. https://doi.org/10.1145/3243734.3243822
[65] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page {Table-Based} attacks on enclaved execution. In *26th USENIX Security Symposium (USENIX Security 17)*. 1041–1056.
[66] Yuanpeng Wang, Ziqi Zhang, Ningyu He, Zhineng Zhong, Shengjian Guo, Qinkun Bao, Ding Li, Yao Guo, and Xiangqun Chen. 2023. SymGX: Detecting Cross-boundary Pointer Vulnerabilities of SGX Applications via Static Symbolic Execution. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2710–2724. https://doi.org/10.1145/3576915.3623213
[67] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*. Springer, 440–457.
[68] Rafal Wojtczuk and Joanna Rutkowska. 2011. Following the White Rabbit: Software attacks against Intel (R) VT-d technology. https://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf.
[69] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*.
[70] Donghui Yu, Jianqiang Wang, Haoran Fang, Ya Fang, and Yuanyuan Zhang. 2023. SEnFuzzer: Detecting SGX Memory Corruption via Information Feedback and Tailored Interface Analysis *(RAID '23)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3607199.3607215
[71] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/ Retrieved 2024-07-01 16:50:18+02:00.
[72] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (Sept. 2022), 36 pages. https://doi.org/10.1145/3512345

A Appendix

A.1 SDK and Library OS Analysis

Tab. 8 shows our findings for signal support in various SDKs and Library OSes.

A.2 Programming Language Analysis

Tab. 6 shows the signal handlers that were executed for different programming languages. Tab. 7 shows the signals that are registered by default interpreters and compilers. Further, Tab.3 shows the different compiler versions we used for our programming language analysis. Tab. 9 shows the different languages we analyzed and the type (implicit or explicit as shown in Fig 7) of signal handling support they provide.

A.3 Signal Number to Names Mapping

Tab.4 shows mappings from signal numbers to names.

A.4 Vulnerable Applications

Tab.5 shows the publicly available applications we manually analyzed and found to have signal handlers with global effects.

Lst.11, Lst.8, Lst.10, Lst.9, and lst.12 show Java code snippets we found by manually surveying applications along with JSAT (Sec. 5) on Github for vulnerable signal (`ArithmeticException`) handlers with global effects. All these applications catch `ArithmeticException` and handle it in a way that affects the global state of the program.

```
try {
    q = n.divide( d );
    ex = true;
} catch( ArithmeticException e ) {
    q = n.divide( d, MathContext.DECIMAL32 );
    ex = false;
}
```

Listing 8: Snippet from Pellet: An Open Source OWL DL reasoner for Java [5]

```
try {
    r3 = RT.addExact(r1, r2);
    _r3 = null;
} catch(ArithmeticException e) {
    _r3 = RT.addOverflowed(r1, r2);
    r3 = 0;
}
```

Listing 9: Snippet from DartRuntime [1]

```
parity = 1;
try {
  for(int i = 0; i<n; i++) {
    swapRows(i, largestPivot(i));
    pivot(i);
  }
} catch(ArithmeticException e) {
  parity = 0;
}
```

Listing 10: Snippet from OSP: Open Source Physics Core Library during LUPDecomposition [4]

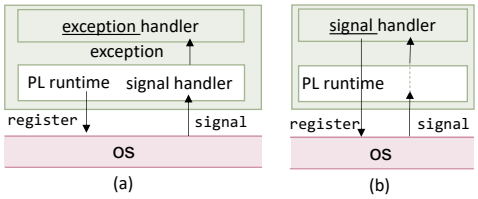


Figure 7: (a) Languages with implicit support for signals. The PL runtime registers signal handlers with the OS and converts signals from the OS to exceptions for the applications. (b) Languages with explicit support for signals. The PL runtime allows the applications to directly register handlers with the OS which are invoked when OS sends signals.

Table 2: Experiment setup.

Machine	Intel CPU	Cores	RAM	OS (Ubuntu)	Kernel	SGX SDK	SGX Driver	Open JDK
Laptop	Core i7-10875H	8	32 GB	20.04	5.10.2	v2.16	2.11.0	17
Server	Xeon Gold 6346	32	378 GB	20.04	5.19.0	v2.16	in-tree	17

```
BigDecimal expIn = new BigDecimal(exp);
long lMs = 0;
try {
    @SuppressWarnings("unused")
    long lSeconds = expIn.longValueExact();
    expIn = expIn.multiply(BigDecimal.valueOf(1000l));
    lMs = expIn.longValueExact();
}catch(ArithmeticException ae) {
    //System.out.println("PARSE OVERFLOW ["+exp+"]");
    lMs = Long.MAX_VALUE;
    expIn = BigDecimal.valueOf(lMs);
}

if(lMs>0) {

    try {
        expIn = expIn.add(BigDecimal.valueOf(now.getTime()));
        lMs = expIn.longValueExact();
    }catch(ArithmeticException ae) {
        //System.out.println("PARSE OVERFLOW 2 ["+exp+"]");
        lMs = Long.MAX_VALUE;
        expIn = BigDecimal.valueOf(lMs);
    }
    ...}
```

Listing 11: Snippet from GovWay - API Gateway for Public Administration [2]

```
public static Object multiply(Object o1, Object o2) {
    if (o1 instanceof Integer && o2 instanceof Integer) {
      try {
        return Math.multiplyExact((Integer)o1, (Integer)o2);
      } catch(ArithmeticException e) {
        // do nothing
      }
    }
    return ((Number)o1).doubleValue() *
    ((Number)o2).doubleValue();
}
```

Listing 12: Snippet from jsjs: JavaScript compiler [3]

A.5 Experimental Setup

Tab.2 shows the experimental setup we use to demonstrate SIGY.

Table 3: Compiler/Interpreter version.

Language	Version
C	gcc (GCC) 13.2.1
C++	gcc (GCC) 13.2.1
Java	OpenJDK 17.0.10
Python	Python 3.11.6
Go	go 1.21.6
Node.js	v21.6.1
Rust	cargo 1.75.0
Wasm	wasmtime-cli 17.0.0
Julia	julia version 1.10.2

Table 4: Signal number to signal name mappings. * Signals from hardware exceptions

#	Name	#	Name
1	SIGHUP	2	SIGINT
3	SIGQUIT	4*	SIGILL
5*	SIGTRAP	6	SIGABRT/SIGIOT
7*	SIGBUS	8*	SIGFPE
9	SIGKILL	10	SIGUSR1
11*	SIGSEGV	12	SIGUSR2
13	SIGPIPE	14	SIGALRM
15	SIGTERM	16	SIGSTKFLT
17	SIGCHLD	18	SIGCONT
19	SIGSTOP	20	SIGTSTP
21	SIGTTIN	22	SIGTTOU
23	SIGURG	24	SIGXCPU
25	SIGXFSZ	26	SIGVTALRM
27	SIGPROF	28	SIGWINCH
29	SIGIO	30	SIGPWR
31*	SIGSYS/SIGUNUSED		

Table 5: Analysis of applications for signal handlers

Application	Signal	Description
nginx	SIGHUP	reload configuration files and replace old processes
httpd	SIGUSR1	reload configuration files and replace old processes
squid proxy server	SIGHUP	reload configuration files and replace old processes
redis	SIGUSR1	kill a child process during backup without the parent treating it as an error
Prometheus	SIGHUP	reload configuration files
mongodb	SIGUSR1	write log files to disk
Node.js	SIGUSR1	open debug server

Table 6: Signal support in programming languages. ✓executes signal handler, ✗ crashes the program, ❉ no observable behavior.

Language	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
C++	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Java	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Python	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Go	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	❉	✓	✓	✓	✓
JS	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Rust	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Wasm	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Julia	✗	✓	✗	✗	❉	✗	✗	✓	✗	✓	✗	❉	❉	✗	✗	✗	❉	❉	✗	✓	✓	✓	✓	✗	✗	✗	✗	❉	✗	✗	✗

Table 7: Signal handlers registered by default in interpreters/compilers. ✓handler registered, ✗no handler registered.

Language	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
C	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
C ++	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Java	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Python	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Go	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
JS	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Rust	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Wasm	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Julia	✗	✓	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

Table 8: Signal support in SDKs & library OSes. ✓executes signal handler, ✗ crashes the enclave, ❉ no observable behavior. (Appx. A.3 for mappings from signal number to signal name).

Runtime/LibOS	1	2	3	4*	5*	6	7*	8*	9	10	11*	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31*
Open Enclave	✓	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗	❉	❉	✗	❉	❉	❉	❉	✗	✗	✗	✗	❉	❉	✗	✗
Teaclave	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Asylo	✓	✓	✓	❉	✓	✓	❉	✓	✗	✓	✓	✓	✓	✓	✓	✓	❉	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	❉	✓	✓
Gramine	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓	❉	❉	✗	❉	✗	❉	❉	✗	❉	❉	❉	❉	✗	✗	✗	✗	❉	✗	✗	✗
Scone	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EnclaveOS	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Occlum	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 9: Implicit: Runtime converts signals into s/w exceptions. Explicit: Processes can register signal handlers.

Language	Implicit	Explicit	Language	Implicit	Explicit
C	✗	✓	Go	✗	✓
C++	✗	✓	JS	✗	✓
Java	✓	✗	Rust	✗	✓
Python	✗	✓	Wasm	✗	✗
Julia	✓	✗			