SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks

Shachee Mishra
Stony Brook University
New York, USA
shmishra@cs.stonybrook.edu

Michalis Polychronakis
Stony Brook University
New York, USA
mikepo@cs.stonybrook.edu

ABSTRACT

Intel SGX is a hardware-based trusted execution technology that partitions an application into trusted and untrusted parts. The trusted part, known as an enclave, executes within an encrypted memory environment, preventing the host application and the OS from being able to access its memory. The enclave, however, has the ability to access the host's memory. When considering malicious code running in an enclave, the strong memory isolation and encryption properties offered may aid the stealthiness of malware, since malware detection tools cannot inspect the enclave. The enclave and the host communicate over bi-directional interfaces that the Intel SGX SDK generates.

In this work, we present SGXPecial, a best-effort interface specialization tool that statically analyzes both the host and the enclave to generate interfaces tailored only to their needs. SGXPecial is implemented as an extension to the Edger8r tool of the SGX SDK, and performs API specialization at build time. In particular, SGXPecial performs function, argument, and type-based specialization to restrict the valid control flows across the host-to-enclave boundary. We evaluate SGXPecial's security impact by testing it on SGX SDK sample applications and four open-source SGX applications. SGXPecial effectively prevents five proof-of-concept code reuse attacks in all tested applications.

ACM Reference Format:
Shachee Mishra and Michalis Polychronakis. 2021. SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks. In *14th European Workshop on Systems Security (EuroSec'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3447852.3458716

1 INTRODUCTION

Intel Software Guard Extensions (SGX) can be used to isolate trusted enclaves that are part of a host application from the rest of the system using hardware-based isolation. Using SGX, the protected enclave remains secure even when the host application, operating system, or hypervisor turn malicious. The same isolation, however provides a means for malicious code to safely hide in an enclave without getting detected by generic malware detection tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSec'21, April 26, 2021, Online, United Kingdom
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8337-0/21/04…$15.00
https://doi.org/10.1145/3447852.3458716

While SGX's memory encryption enhances the security of enclaves, an attacker could take advantage of SGX to prevent malware from being inspected. Furthermore, SGX supports the use of generic loaders which could load the executable code from a remote source only at runtime, preventing install-time analysis of the enclave code. Recently, several attack scenarios in which malicious enclaves exploit the host application have been proposed [12, 19, 23].

Application debloating and specialization have recently gained popularity as a means to reduce the code present in the address space of a running process that an adversary could use as part of an exploit. Sources of code bloat include unused application features and unused functions from shared, among others. Debloating techniques either statically analyze an application's source code [1, 18] or binary [16, 21], or execute the application with a representative set of inputs to extract the set of functions that are used [2]. Once identified, the unused functions are removed.

API specialization, on the other hand, does not remove any code, but tightens function interfaces so that calls to these functions are restricted in the arguments that can be accepted [14, 15]. Similarly, the system calls available to a process [7] or even a whole container [6] can be restricted by applying a Seccomp filter tailored to their needs. These techniques have been used in a variety of domains, including end-user applications, Java bytecode, interpreters, containers, and the OS kernel. To the best of our knowledge, our work is the first to study the application of code specialization in the domain of trusted execution environments.

Intel SGX provides a communication interface between the host and the enclave which is analogous to the library call interface in regular applications. The Intel SGX runtime libraries provide a variety of functionalities, only few of which are actually used during an application's lifetime. Thus, a specialization approach that performs static analysis of the enclave and host application and specializes interactions between them could be used to restrict an attacker who has compromised the host or the enclave. To that end, we propose *SGXPecial*, a lightweight, best-effort approach for analyzing and specializing the SGX interfaces that handle the communication between the enclave and the host application.

We summarize our main contributions as follows.

- We propose SGX interface specialization, a best-effort specialization technique for host-to-enclave interfaces.
- We designed and implemented *SGXPecial* based on the Intel SGX SDK, which employs interface specialization to transparently protect the host application and enclaves from exploiting each other through code reuse attacks.
- We studied five proof-of-concept code reuse exploits that perform attacks against SGX-based systems, and demonstrate the effectiveness of SGXPecial in stopping blocking them.

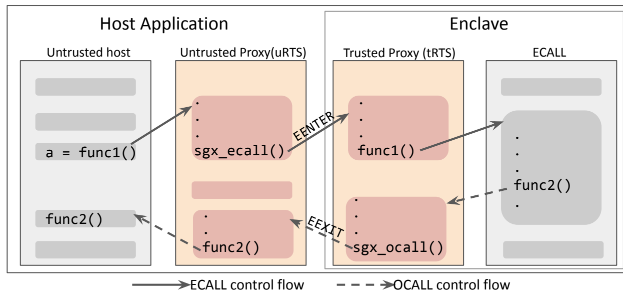

Figure 1: SGX Internals

2 BACKGROUND AND MOTIVATION

2.1 Intel SGX Internals

In this section we provide a brief introduction to Intel SGX and its SDK. The Intel SGX SDK is one of the most widely used platform for enclave development. We focus our discussion on the various transition interfaces in an enclave's lifetime.

Enclaves are loaded to the memory space of the host application in the form of a shared library, the memory of which is encrypted. According to SGX's threat model, the enclave is considered trusted, and thus has complete access to the host's memory. This access is required for sharing data in and out of the enclave.

As an application is essentially divided into trusted and untrusted parts, their interaction is handled via two runtime system layers, one on each side of the trust boundary. The Trusted Runtime System (tRTS) is statically compiled into the enclave code, while the Untrusted Runtime System (uRTS) is dynamically loaded when the host executes.

An Enclave Description Language (EDL) file defines the entry points into the enclave. When a project is compiled, the Edger8r tool parses the EDL file and generates the appropriate interfaces, in the form of wrapper functions for the functions mentioned in the EDL. Each function is assigned two wrapper functions, one in the untrusted half and one in trusted half. These interfaces marshal data in and out of the enclave and maintain function success status.

The entry points are accessed by making a special function call, known as ECALL. These ECALLs start with an EENTER instruction. Figure 1 shows the process of making an in-enclave function call. The host calls a trusted function `func1()`, which is executed after crossing the enclave boundary by calling `sgx_ecall()`. The secure functions are specified by an index in the array of ECALLs.

Inside the trusted environment, when the enclave requires the invocation of an external function, these are accessed using another special set of functions called OCALLs. These OCALLs could be functions in the host application, or system calls to the OS. Similarly to ECALLs, OCALLs are also specified by an index in the array passed from the host application. In Figure 1, the ECALL function calls `func2()`, which is a function in the host application, and is transitioned to the host via `sgx_ocall()`.

2.2 Control Flow Hijacking in SGX

The Intel SGX SDK allows enclave development in C/C++, which means that traditional memory corruption vulnerabilities are also possible in enclaves. Conventionally, enclaves are assumed secure from code reuse attacks like return-oriented programming (ROP) due to enclave memory encryption. An attacker thus cannot read the memory from outside an enclave and find gadgets to chain together. While this seems like a robust design at first, there have been several attacks which manage to successfully explore the enclave memory [3, 4, 11]. At the same time, attacks where malicious enclaves exploit the host have been recently explored as well [19, 23].

Dark-ROP relies on repeatedly executing the enclave, under the assumption that in-enclave memory layout does not change across multiple runs. SGX-Shield [20], a fine-grained randomization defense, offers some protection against the Dark-ROP attack. However, the trusted enclave runtime system cannot be randomized using the techniques proposed in SGX-shield. The Guard's Dilemma [3] and TeeRex [4] attacks also reuse instructions from the trusted runtime system libraries.

2.3 Securing SGX Interfaces

Memory corruption vulnerabilities can be present in both the untrusted and trusted parts of an application. In the TeeRex [4] and Coin Attacks [9], the authors discuss memory vulnerabilities in a number of popular enclave setups. They found vulnerabilities using symbolic execution and were also able to hijack control flow inside enclaves using specially crafted arguments to ECALLs. On the other end, SGXJail [23] and SGX-ROP [19] have highlighted that a malicious enclave can hijack the control flow of the host. Given that an enclave has complete access to the host's memory, SGXJail and SGX-ROP have demonstrated that the current model of considering enclaves as fully trusted could lead to security implications for the host.

With these two attack scenarios in consideration, there is an emergence of a potential enclave malware threat, in addition to the threat of untrusted hosts. We believe that instead of considering one half (enclave) as trusted, while the other (host and host OS) as untrusted, the two halves should both be segregated as two separate domains, and they should both be treated as untrusted. The only trusted component in this architecture should be the interfaces generated by the SDK.

3 THREAT MODEL

3.1 SGX's Threat Model

In the threat model for SGX, every piece of software outside the enclave, including the host application and the OS, are deemed untrusted. However, features such as enclaves having access to the entire address space of the application, and the fact that enclaves can avoid getting their code inspected by using a generic loader, make it possible for a malicious enclave to exploit the host application. Recent attacks [19, 23] have proven that the Intel SGX threat model could put host applications at risk.

3.2 Our Threat Model

For our work, we identify three key parties in the application ecosystem:

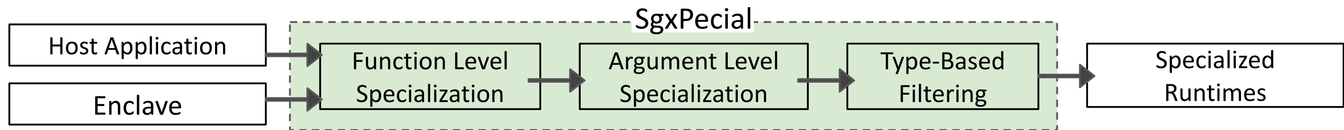

Figure 2: Overview of SGXPecial.

(1) Host Application: A host application is deemed untrusted by the SGX threat model. We assume the host application source code is available for us to analyze.
(2) Enclave: An enclave or the trusted part of the SGX ecosystem is considered to be a self-contained entity in itself. Enclaves can either be part of the host application, or third-party enclaves that the current application imports and uses.
(3) Interface: In this work we focus on the Intel SGX SDK, as it is one of the most popular tools for enclave development, but any development environment can be considered as an interface. Interfaces facilitate data and control transfer between trusted and untrusted components. The SGX runtime libraries which perform data marshaling and unmarshaling, copying data into and out of the enclave and use EENTER, EEXIT and similar instructions to also move control in and out of the enclave. For Intel SGX SDK, this set of libraries includes libsgx_trts.a (Trusted Runtime System) and libsgx_urts.so (Untrusted Runtime Systems) amongst others.

We consider two different attack profiles: *Malicious Host Application or Malicious Host OS [3, 4, 11]*. This is the standard SGX threat model described previously. The host application is untrusted, and hence, enclave memory should not be visible to it. Along with this, as SGX and similar trusted environments are used by more applications, and more code is added to enclaves, the chances of exploitable vulnerabilities within the enclaves also increase. An untrusted host application could exploit these vulnerabilities and try to gain control of enclave.

Malicious Enclaves [19, 23]. As third party enclaves gain popularity, an application developer using such enclaves has little control on what the enclave does. The attackers could defer fetching malicious payload to execution time, which would prevent the detection of malicious behavior before they are executed. The application should be protected against any malformed inspection or update of its memory by an enclave. But the model of memory access is still maintained as in SGX, i.e., the enclave has complete access to the host's address space.

SGXPecial identifies that the application could not be secured efficiently if the enclave and host application boundaries are not clearly defined. For example, a too permissive ECALL interface, where every argument is marked as `user_check`, would not be verified by SGX boundary checks, and thus would make the defenses worthless. Limitations in enclave design are thus out of scope.

4 DESIGN

In this section, we present SGXPecial, a novel technique to protect i) naive trusted enclaves from untrusted host applications, and ii) host applications from malicious third-party enclaves.

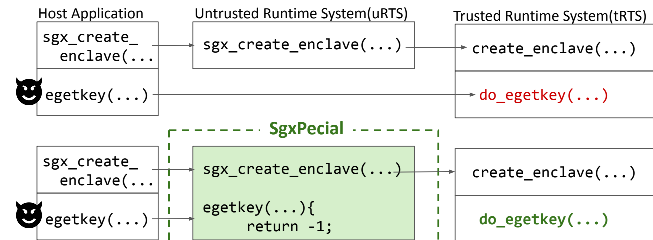

Figure 3: Function Specialization for SGXPecial. SGXPecial identifies the functions that are not used by the application or by the enclave and creates wrappers to stop them from crossing the interface.

4.1 SGX Runtime Specialization

The communication between trusted and untrusted worlds happens via SGX interfaces. SGXPecial creates a specialized set of interfaces customized to the current application.

Figure 2 presents an overview of SGXPecial. For every statically identified function call that goes through the interfaces, we derive policies specific to each call site. We perform three levels of specialization in the SGX interface: i) Function level specialization, ii) Argument specialization, iii) Type-based filtering.

4.1.1 Function-level Specialization. Attack Surface reduction techniques [1, 2, 7, 10, 18] that identify the set of functions that an application uses and restrict access to any other functions have gained popularity in recent years.

Although this is a popular technique, there is one major difference while porting it to SGX. Identifying a function that is not used by both the application and enclave does not make it a candidate for removal from address space. For example, it might still be used for remote attestation or for report generation purposes. Removing a function from the address space, only because it is not used by the application and the enclave, thus, might lead to execution errors.

Therefore, we statically analyze the source to identify the set of function calls that are made across the interface and at runtime restrict any other functions calls from crossing the host-to-enclave boundary. We create wrapper functions for every function used by the application and the enclave. For every in-enclave function that an application uses, we create a custom wrapper function in the untrusted bridge. For the functions that the current application uses, these wrappers call the actual function in the uRTS, but for the functions that the application does not use, the function calls are returned without execution of any further code. Figure 3 shows a sample application which only uses `sgx_create_enclave()`, but an adversary can execute function `egetkey()` to retrieve sensitive cryptographic keys. With SGXPecial, the function `egetkey()` is

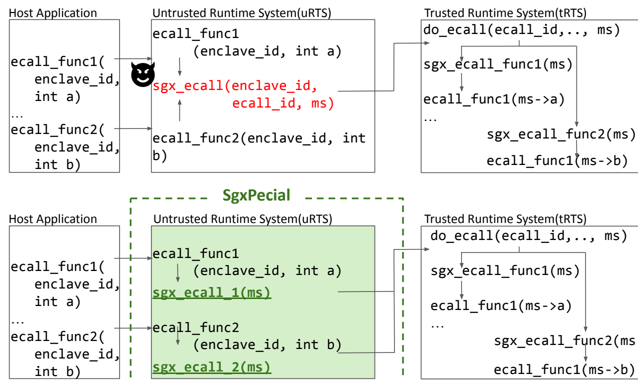

Figure 4: Argument Specialization. A custom function is generated for every call site, which neutralizes static arguments from the original function call.

intercepted and blocked. Similar wrappers are created in tRTS for calls made from within the enclave.

For the OCALLs, i.e. calls from inside the enclave to outside, this function level specialization identifies the external functions (from the host application, system calls or from uRTS), that the enclave uses. Using the same wrapper function based technique, every call from inside the enclave is intercepted by its custom wrapper in the trusted runtime(tRTS).

4.1.2 Argument Specialization. Communication in and out of enclaves happen via means of function calls. While some functions are meant for specific purposes, for e.g. create or launch an enclave, destroy or delete an enclave etc, other functions such as ECALLs are used to invoke different functionalities inside the enclave. One of the arguments passed to an ECALL/OCALL is the index of the function to be executed. Changing this argument value can make a specific call site alter the function that gets executed. Thus, it is critical to ensure that an attacker can not make arbitrary changes to the arguments.

SGXPecial creates specialized versions of functions for each of the function call sites. These specialized functions are tailor made by neutralizing the arguments used at the call site. Figure 4 explains argument specialization step of SGXPecial. The top half of the figure, shows a standard `sgx_ecall()`. The functions `ecall_func1()` and `ecall_func2()`, access `sgx_ecall()` within the uRTS. The only way to differentiate between the two functions is by the `ecall_id` argument passed to `sgx_ecall`. SGXPecial performs argument specialization and creates two specialized versions `sgx_ecall_1()` and `sgx_ecall_2()`, one for each `sgx_ecall` invocation. As can be seen, the `ecall_id` is neutralized and removed from the arguments. At the same time, other statically known arguments are also neutralized.

4.1.3 Type-based Filtering. For generic implementation purposes, the arguments for individual functions are marshaled into a structure and a pointer to this structure is passed as the argument to `sgx_ecall` or `sgx_ocall`. An adversary can take advantage of this generic struct passing, by marshaling any set of values into the struct and passing them into the enclave. A similar attack scenario

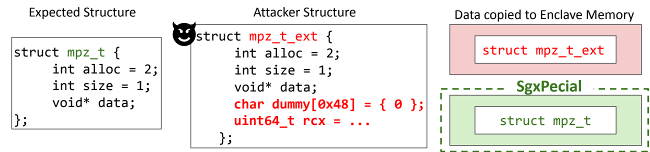

Figure 5: Type-based Filtering. SGXPecial verifies that the number and types of arguments are the same as those expected by the called function.

is used in Teerex [4], shown in Figure 5. While the original ECALL function expects an argument of type `mpz_t`, the attackers alter the elements in the structure creating a new type `mpz_t_ext`.

Type-based filtering ensures that the number and types of arguments marshaled into the structure are the same as the type and number of arguments in the original function.

SGXPecial does not make any changes to the enclave or to the host. This means that SGXPecial has complete backward compatibility, thus an application can be run using a pair of interfaces that are not specialized.

Path of an ECALL with SGXPecial. The host makes a function call to one of the trusted functions. This function call is intercepted by the untrusted interface. Next, this SGXPecial enabled interface performs verification at function, argument and type levels to ensure that the function call is in accordance with the statically defined behavior. When the function is part of the naive host application, the static arguments are removed from the argument set and a call to is made to the `sgx_ecall()` function corresponding to this call site, tailored only to the arguments of this invocation.

Lastly, SGXPecial performs type based filtering to ensure that only the argument number and types from the current function invocation are copied into the enclave. At this point, EENTER instruction is used to call the actual in-enclave function. A path similar to this is followed for an `sgx_ocall`.

SGXPecial enforces every communication between the trusted and untrusted worlds to occur over the Intel SGX SDK bridge. Every attempt to call a function within the enclave is captured by the SGXPecial enabled uRTS, which after thorough verification lets the function execute.

5 IMPLEMENTATION

We implemented SGXPecial by extending the Edger8r tool provided by the Intel SGX SDK.

Function-level Specialization. SGXPecial ensures that every in-enclave function that is used by an application is part of the naive host. We augment the edger8r tool, to create wrappers for every function within the uRTS. A custom wrapper is also created for every function that is not used by the application, which returns without executing the function. For example, if an application uses the function `sgx_create_enclave()`, but does not use `sgx_create_enclave_ex()`, the call simply returns for the latter, while the former one is executed within the enclave.

Argument Specialization. The core of SGXPecial is its ability to neutralize argument values. We modify the Edger8r tool to identify

values of arguments known at build time. The static arguments are introduced inside the interface code by creating specialized wrapper versions for each such call.

For OCALLs, the SGX SDK passes arguments to the host by writing them to application's stack using `sgx_ocalloc`. An enclave can modify the arguments to `sgx_ocalloc` and alter the contents of the stack. To prevent this, SGXPecial specializes OCALL functions as well.

Type-based Filtering. Edger8r creates functions that marshal the arguments into structures. We extend edger8r to augment this with a type verification step. SGXPecial validates the number of elements and their types. The types are only verified for primitive data types using `sizeof`. Using type based filtering, only valid argument types are allowed to pass in form of marshaled structure. This restricts adversaries like those in Teerex [4], from exploiting the argument structure.

6 EVALUATION

In this section, we present results of our experimental evaluation in terms of the security impact and performance overhead of interface specialization using SGXPecial. We also present case studies of five real world exploits and SGXPecial's effectiveness against them.

6.1 Data Set

We studied a wide range of exploits against Intel SGX. Along with code reuse exploits, we also studied other exploits like side-channels to gain insight into the malicious operations they perform, to come up with a list of security-critical functions in SGX runtimes. These functions include `sgx_is_within_enclave`, `asm_oret`, `do_egetkey`, `sgx_ocalloc`, `trts_mprotect`, `do_oret`, `restore_xregs` and `do_ecall`.

We studied five real-world exploits and the functions they invoke. Each of these exploits use one or more critical functions.

- Dark-ROP [11]: `do_ereport`, `do_egetkey`, `memcpy`, `do_ecall`
- Guard's Dilemma [3]: `do_ereport`, `do_egetkey`, `do_ecall`
- TeeRex [4]: `asm_oret`, `do_egetkey`, `do_ecall`
- SgxJail [23]: `sgx_ocall`
- SGX-ROP [19]: `sgx_ocall`

As can be seen, `do_ecall()` is used in every untrusted application attack scenario, and `do_ocall()` is used in every attack where the enclave is malicious. This is because `do_ecall()` is one of the most frequently used entry points into the enclave. Most of the exploits use `do_ecall()` or `do_ocall()` with malicious functions and arguments to hijack the control flow of the enclave. As these functions are used in the application, the access to these can not be completely blocked using mere function level specialization.

6.2 Application Specialization

6.2.1 Intel SGX SDK Sample Enclave Applications. We tested SGX-Pecial with sample applications installed with Intel SGX SDK. We discuss our findings in the first half of Table 1. Each row in table is for one application. The first two columns provide the percentage of functions from uRTS and tRTS that the application along with its trusted enclave use. For instance, in SampleEnclave only 9 (43%) of the 21 functions exported by the uRTS are used by the host. Similarly the enclave uses only 7 (7%) out of 98 functions that tRTS exports. In absence of SGXPecial all the other functions can be exploited by an attacker. But with SGXPecial, the attacker can only access the functions used by the application/enclave.

In the next two columns we present the number of ECALL invocations in each functions and the percentage of arguments within these call sites that we could statically identify. In SampleEnclave, for 64 ECALLs, 78 (30%) out of 256 arguments could be statically known, while for Switchless 5 (31%) of 16 arguments were known from 4 ECALLs.

There is no direct way of gauging the security benefit of type based filtering, thus we leave that discussion to specific exploits in the next section.

6.2.2 Real-world Applications. We picked four real world open-source SGX applications to test SGXPecial, listed in the bottom half of Table 1. MbedTLS, a popular open source SSL library, only uses 4 (19%) out of 21 exported functions from uRTS, while the SGX based implementation of SQLite database only uses 10 (47%) functions.

For argument level specialization, third and forth columns of the table provide the number of ECALLs and the percentage of arguments that could be statically known and thus specialized. For the SGX implementation of SSL, there are two ECALLs, half of the arguments to which were neutralized. For the SGX-based file encryption engine SGXCryptofile, for four OCALLs, half of the arguments are statically known and were neutralized.

6.3 Security Case Studies

6.3.1 Dark-ROP [11]. Dark-ROP exploits an enclave by using `pop` instructions to set registers and find gadgets in the enclave. The adversary uses ECALLs to propagate malicious instructions into the enclave. When executed, these instructions find useful gadgets. The attack starts by overwriting the `rbx` register to address of `import_data_to_enclave()` function inside the enclave. Through our experiments, we deduce that this function is never used by the host application, thus function level specialization prevents the attack.

6.3.2 Guard's Dilemma [3]. Guard's dilemma attack looks for ROP gadgets in the tRTS. The first step however, is to execute one of their two primitives from outside the enclave. This attack uses two primitives: 1. *CONT*- triggered by `continue_execution()`, to restore context after an exception or, 2. *ORET*- triggered by function `asm_oret()` used to restore context after an OCALL. Once SGXPecial specializes these functions and neutralizes arguments passed to them, both these primitives are prevented.

6.3.3 TeeRex Exploits [4]. Teerex exploits detect five vulnerability classes that occur in enclaves. In one of their PoC exploits, they replace arguments passed to the ECALL (Figure 5). Inside the enclave, the additional fields overwrite the stack. The control is then transferred to `asm_oret()`, which restores a counterfeit context inserted as part of the same struct. Using SGXPecial, before argument marshaling the arguments types are verified. This way a struct with extra elements would not be copied into the enclave.

6.3.4 SGXJail Exploits [23]. SGXJail considers a malicious enclave and uses shared memory and Seccomp filters to restrict the system

Application	% of uRTS function used	% of tRTS functions used	#ECALLs	% of Known Arguments	#OCALLs	% of Known Arguments
SampleEnclave	43%	7%	64	30%	22	55%
LocalAttestation	19%	6%	14	25%	4	50%
SealUnseal	43%	6%	3	25%	10	50%
Switchless	43%	7%	4	31%	6	59%
PowerTransition	47%	2%	2	25%	6	50%
Cxx11SGXDemo	14%	4%	26	50%	6	50%
SampleEnclavePCL	38%	3%	32	31%	11	55%
mbedtls-SGX	19%	8%	8	34%	11	50%
SGX_SQLite	47%	11%	3	33%	24	50%
Intel SGX SSL	38%	9%	2	50%	9	55%
SGXCryptofile	14%	5%	5	25%	4	50%

Table 1: Results from sample and real-world applications.

Table 2: Average Runtime overhead (in ns) for single ECALL and OCALL. The numbers in braces correspond to the standard deviation.

Latency	ECALL	OCALL
Vanilla	1244 (±56)	1410 (±78)
SGXPecial	1380 (±75)	1537 (±102)

calls an enclave has access to. But, a system call can be a valid OCALL in the enclave EDL and SGXJail would still block its execution. However, using SGXPecial, the system calls are identified and allowed to execute securely.

6.3.5 SGX-ROP [19]. SGX-ROP uses write-anything-anywhere primitive from a malicious enclave to create a fake stack and pivot the host control to it. This fake stack has the attacker inserted payload as sequence of gadgets. SGXPecial monitors and blocks writes which do not correspond to the original enclave. SGX-ROP uses Intel TSX to handle memory exceptions. In absence of TSX, this attack requires careful memory probing via OCALLs, which are specialized using SGXPecial.

6.4 Runtime Performance

Measuring the runtime overhead of a system like SGXPecial is a challenging task. Host and enclave interactions are infrequent and thus the real overhead depends on the number of function calls made during the test run. In order to measure the runtime overhead of SGXPecial, we used a set of custom host and enclave. All the tests are performed over a 64-bit Intel Core i5 machine with 32GB of RAM running Ubuntu 18.04 Bionic. To measure an ECALL overhead, we perform ECALL invocations for 1 million times, each invocation to a random trusted function. This is orders of magnitude above the average number of ECALLs made by real applications.

The results are summarized in Table 2. The time taken for individual function calls, only marginally increases with SGXPecial enabled SDK against Vanilla implementation.

7 LIMITATIONS AND FUTURE WORK

SGXPecial enforces three phases of specialization that collectively increase the efforts required by an adversary to cross the trust boundary. An attacker can however still launch a successful attack by creating payloads closely corresponding to the benign application/enclave. This can be done by using the same set of functions as the enclave or the host, or by either using the same arguments as the functions in enclave or the host and also by exploiting the cases of unknown argument values. Similarly, by using argument structures which closely correspond to the actual types of arguments, an adversary can successfully bypass SGXPecial. SGXPecial thus should be viewed as defense-in-depth solution that considerably increases the bar for the attackers.

With third-party enclaves, attacker can modify the interfaces to make them more permissive. This can be done by including functionalities that are required by the malicious enclave loaded over network using the generic loader. However, if SGXPecial is integrated with Intel SGX SDK, creating such libraries would imply modifying the SDK itself.

We currently do not perform sophisticated static analysis on the host or on the enclave. Thus, there is a scope for gaining more insights by detailed analysis to strengthen the specialization. We leave such analysis to our future work.

Our current work prototype is built on Intel SGX SDK, however there are multiple SDKs which provide the ability to build applications on SGX. These include Open Enclave [13], SGX-LKL [17] and Asylo [8]. Open Enclave provides *oeedger8r*, an extension to edger8r from Intel SGX SDK and creates the marshaling runtimes for the applications. Keystone, an SDK for building TEEs on RISC-V, provides a set of libraries: *Host libraries* and *Edge libraries* provide interface for enclave creation, management and communication. Thus in future, SGXPecial can be extended to these SDKs as well.

8 RELATED WORK

Using Control Flow Integrity to prevent against control hijacking does not block all attack vectors as explained in SGXJail [23]. Attacks like Guard's Dilemma [3], COIN attacks [9] specifically target the interfaces to mount such attacks. SGX-ROP [19] uses Intel TSX

to probe host memory from an enclave and steer control flow to a fake stack. Bulck et al. [22] explore runtime environments of trusted environments including SGX and conclude that a significant number of memory safety vulnerabilities still exist in them. SnakeGX [5] is a novel attack that exploits `oret()`, the secure function that handles returns from OCALLs and fake contexts to achieve arbitrary code execution and backdoor installation inside the enclave.

Attack surface reduction has been established as an effective and low overhead defense against code reuse attacks. Quach et al. [18] perform dynamic library debloating by only loading the required functions from shared libraries. Nibbler [1] does the same at binary level. Temporal specialization [7] identifies the different system call needs of a server application at initialization vs. serving phases and uses Seccomp filters to cater to the changing needs. Shredder [14] performs binary instrumentation to restrict arguments to kernel API functions. Saffire [15] performs call-site specific argument specialization for system calls at build time.

9 CONCLUSION

We presented SGXPecial, a novel interface specialization technique for Intel SGX based applications. An SGX enclave is hardware isolated from rest of the execution environment. While the traditional threat model of untrusted host and trusted enclave is valid for most cases, third party enclaves pose dangers for the host as well. SGXPecial secures an enclave from a malicious host and a naive host from a potentially dangerous enclave.

SGXPecial is a best-effort tool that performs static analysis of host and enclave source to create SGX interfaces that are specialized to the needs of only that application. SGXPecial starts by performing function based specialization by identifying used functions from the SGX SDK runtimes, followed by extracting static argument values from the used function arguments. Finally, SGXPecial performs validations for the data type of arguments. All these validations are instrumented into the SGX runtime libraries by modifying the edger8r tool provided by Intel SGX SDK.

By performing these sequence of function, argument and type based specialization, SGXPecial prevents a malicious actor from crossing the host-to-enclave boundary without proper verification. We evaluated SGXPecial with real world applications and exploits and found it to be highly effective in preventing code-reuse attacks.

ACKNOWLEDGEMENTS

This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, and the National Science Foundation (NSF) through award CNS-1749895.

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the 35th Annual Computer Software and Applications Conference (ACSAC)*.

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*.

[3] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*. 1213–1227.

[4] 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)*.

[5] Mauro Conti Flavio Toffalini, Mariano Graziano and Jianying Zhou. 2021. SnakeGX: a sneaky attack against SGX Enclaves. In *19th International Conference on Applied Cryptography and Network Security (ACNS'21)*.

[6] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*.

[7] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA.

[8] Google. 2019. Asylo: An open and flexible framework for enclave applications. http://asylo.dev.

[9] 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 (ASPLOS '20)*.

[10] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on System Security (EuroSec)* (Dresden, Germany).

[11] 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)*.

[12] Marion Marschalek. 2018. The Wolf in SGX Clothing.

[13] Microsoft. 2019. Open Enclave sdk. http://openenclave.io/sdk/.

[14] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*.

[15] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *2020 IEEE European Symposium on Security and Privacy (Euro S&P)*. IEEE.

[16] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. Black Hat USA.

[17] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143* (2019).

[18] Anh Quach, Aravind Prakash, and Lok Kwong Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*.

[19] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical enclave malware with Intel SGX. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

[20] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *NDSS*.

[21] Benjamin Shteinfeld. 2019. *LibFilter: Debloating Dynamically-Linked Libraries through Binary Recompilation.* Undergraduate Honors Thesis. Brown University.

[22] 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.

[23] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 353–366.