Running Language Interpreters Inside SGX: A Lightweight, Legacy-Compatible Script Code Hardening Approach

Huibo Wang[1], Erick Bauman[1], Vishal Karande[1], Zhiqiang Lin[2], Yueqiang Cheng[3], Yinqian Zhang[2]
[1]The University of Texas at Dallas, [2]The Ohio State University, and [3]Baidu USA

ABSTRACT

Recent advances in trusted execution environments, specifically with Intel's introduction of SGX on consumer processors, have provided unprecedented opportunities to create secure applications with a small TCB. While a large number of SGX solutions have been proposed, nearly all of them focus on protecting native code applications, leaving scripting languages unprotected. To fill this gap, this paper presents ScriptShield, a framework capable of running legacy script code while simultaneously providing confidentiality and integrity for scripting code and data. In contrast to the existing schemes that either require tedious and time-consuming re-development or result in a large TCB by importing an entire library OS or container, ScriptShield keeps the TCB small and provides backwards compatibility (i.e., no changes needed to the scripting code itself). The core idea is to customize the script interpreter to run inside an SGX enclave and pass scripts to it. We have implemented ScriptShield and tested with three popular scripting languages: *Lua*, *JavaScript*, and *Squirrel*. Our experimental results show that ScriptShield does not cause noticeable overhead. The source code of ScriptShield has been made publicly available as an open source project.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

SGX, Scripting Language, Confidentiality, Integrity

ACM Reference Format:
Huibo Wang[1], Erick Bauman[1], Vishal Karande[1], Zhiqiang Lin[2], Yueqiang Cheng[3], Yinqian Zhang[2]. 2019. Running Language Interpreters Inside SGX: A Lightweight, Legacy-Compatible Script Code Hardening Approach. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS'19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3321705.3329848

1 INTRODUCTION

Computer programs can often be attacked due to their reliance on a large trusted computing base (TCB). Typically, an application has to rely on support from linked libraries, the operating system, and sometimes a hypervisor. If any system software is compromised, the execution of the application cannot be trusted. Therefore, Intel introduced its software guard extensions (SGX) [16], a new hardware feature that provides secure enclaves in which user level applications do not have to trust any software outside the enclave, thereby significantly reducing the attack surface.

From a software development perspective, SGX is merely a set of new instructions. To really use SGX, it is suggested that application developers create new applications by using abstractions (e.g., APIs) provided by the SGX SDK, partitioning applications into trusted and untrusted components, and building bridge functions between enclave and non-enclave code. However, this is tedious, time consuming, and error prone. By forcing developers to manually port software to SGX or create new applications from scratch, tons of legacy software cannot enjoy the benefit provided by SGX.

Today, software is typically developed and executed in two ways: (1) compiling high level languages into native code to execute directly; or (2) interpreting source code (or byte code) without compilation, as is typically done in scripting languages. Many existing SGX-based applications are in the first category. They are either built upon a library OS (as in Haven [4] and Graphene [21]) or leverage a container (as in SCONE [2]). As a result, they execute the entire application in SGX without any partitioning into trusted and untrusted components. However, they will inevitably have a large TCB. Also, there are no solutions to protect scripting code from malicious system software while also maintaining backwards compatibility. Note that while TrustJS [8] explored the direction of using SGX to protect JavaScript, it does not provide backwards compatibility.

To fill this gap, we present ScriptShield, a framework capable of running legacy scripting code while simultaneously providing confidentiality and integrity of scripting code and data. ScriptShield achieves backwards-compatibility for scripts by treating them as data streams, and ensures the confidentiality and integrity of scripts by only executing the script and its interpreter in an enclave. To deploy ScriptShield, an end user just needs to recompile and statically link the interpreter for the language they want to use with our framework (there is no need to modify the source code of the interpreter); the resulting interpreter will be then executed inside an enclave, thereby achieving software backwards compatibility.

We have implemented ScriptShield and tested with three popular scripting languages: Lua, JavaScript, and Squirrel, and demonstrated the practicality of our framework (e.g., having low runtime overhead, small TCB, and ease of use). Running an interpreter inside SGX provides many salient benefits: (1) signing it once, and executing everything; (2) confidentiality and integrity for all the content of any unmodified script; and (3) a much smaller TCB.

In short, we make the following contributions:

- We present ScriptShield, a lightweight legacy code compatible framework that provides complete confidentiality and integrity for scripting language protection.

AsiaCCS'19, July 9–12, 2019, Auckland, New Zealand
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329848

- We design an application protocol for securely sending scripts to an enclave, in which scripts are never leaked outside of the enclave.
- We have implemented a prototype and tested with three popular scripting languages—Lua, JavaScript, and Squirrel—and demonstrated the practicality of our framework.

2 BACKGROUND

2.1 Intel SGX

Intel SGX. Intel introduced SGX to provide applications with the capability to execute code and protect secrets in a secure enclave [1, 16]. More specifically, SGX provides software developers direct control over their application's security without relying on any underlying system software such as the OS or hypervisor. This significantly reduces the trusted computing base (TCB) to the smallest possible footprint (only the code executed inside the enclave), and prevents various software attacks even when the OS, BIOS, or hypervisor are compromised. By running trusted code in a secure enclave, secrets remain protected even when an attacker has full control of the entire software stack. Also, the SGX hardware prevents bus snooping, memory tampering, and even cold boot attacks [10] against memory images in RAM since the enclave contents are encrypted by a memory encryption engine (MEE). To prevent any tampering before enclave code is loaded, SGX also provides hardware based attestation capabilities to measure and verify valid code and data signatures.

Scripting Languages. Scripting languages are very high level interpreted languages that are often used to connect or extend components written in lower level languages [17]. There are multiple types of scripting languages, from domain specific languages such as OS shells (e.g., bash), to general purpose languages (e.g., Lua, JavaScript, Python, or R). While native languages are compiled ahead of time, scripts are usually compiled at run time, so there is no need to recompile if the script changes. For example, recompiling a full game can take minutes to hours, which implies a big productivity hit. Game logic and configuration are typically contained in script files. Game designers can easily tweak gameplay by updating these scripts, but they do not want players to be able to do the same and change game logic and configuration to their advantage.

Many popular games today have their game logic written in scripting languages such as Lua. Attacks on these games mainly have two key steps: first obtaining the Lua script, and then modifying or replacing the script. There are some ways to retrieve the original scripts. One is using `unluac`, which is a tool for reverse engineering Lua scripts. Another one is to dump the contents of a script from a few points (e.g., function `lua_read`, function `luaL_loadbuffer`). Therefore, our objective with ScriptShield is to protect scripts like these with integrity and confidentiality, so that attackers cannot reverse engineer or modify them.

3 OVERVIEW

Objectives. We focused on the following objectives while designing ScriptShield:

- **Language-Level Transparency (Backwards Compatibility)**. Given the prevalence of scripting languages, we would like our framework to be transparent to the languages themselves. That is, when using ScriptShield, there should be no modification at the scripting language level, allowing all legacy scripts to still be executed inside ScriptShield.
- **Easier Confidentiality and Integrity Attestation**. Our primary goal is to enable hiding secrets in scripts to defeat reverse engineering attempts and ensure integrity against tampering. We would like to minimize the signing and attestation process and achieve a "*sign once, execute many*" work flow for developers, allowing a single signed enclave to execute any arbitrary script.
- **High Efficiency**. While we could use a library OS or containers to design ScriptShield, such approaches would be inefficient as they need to package a lot of de-privileged kernel code in the enclave and fill more of the enclave page cache (EPC) with code and data. Since the EPC is small (only 128MB in SGX v1 [12]), larger enclaves will suffer performance penalties from paging EPC pages to and from main memory.
- **Small TCB**. Since we aim to protect the confidentiality and integrity of scripting languages, we would like to minimize the attack surface with as small a TCB as possible.

Threat Model and Scope. We focus on the threat model where the owner of a script does not trust the remote platform it runs on (and therefore needs to perform attestation), while the platform owner always trusts the script it is running. Such a model works perfectly in common daily computing contexts such as cloud computing and online computer gaming. In fact, we believe ScriptShield offers an alternative model for how to develop and deploy legacy compatible, secret-preserving programs in these environments.

While several studies have suggested SGX applications are vulnerable to a variety of side-channel attacks [6, 18, 22], fighting side channels is not the focus of this work. However, ScriptShield uses up-to-date cryptographic implementations to avoid, to the best we can, side-channel leakage during secure communication and script decryption. Moreover, as most demonstrated attacks exploit memory access patterns of native code execution, scripts are intuitively less vulnerable to side channels as they are processed as data by the interpreter within the enclave. Thus, we leave enhancing ScriptShield to defeat side-channel threats as future work.

Challenges and Insights. To achieve our stated objectives with SGX, we need to split scripting language execution into two components: a trusted component executed inside the enclave, and an untrusted component outside. Partitioning an application into trusted and untrusted components can be very time-consuming and tedious. While prior research from Haven [4] and SCONE [2] has shown that it is possible to run an entire application inside an enclave without partitioning, such an approach inevitably leads to a large TCB. In contrast, most scripting languages are embedded into certain applications (e.g., Javascript is often executed within a web browser, and Lua is often embedded in a game program). Therefore, library OS or container approaches are not suitable for cases in which only a portion of application code needs to be protected (e.g., game scripts).

A scripting language usually has a self-contained interpreter that interprets the script. If we are able to execute the entire

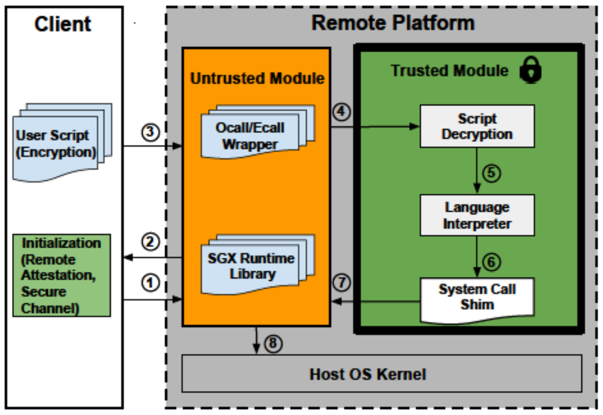

Figure 1: An overview of ScriptShield. Note that ① - ⑧ denotes the i^{th} step while executing ScriptShield.

interpreter inside the enclave, then we do not have to solve the tedious partitioning problem. However, not all interpreter code (e.g., system calls) can be executed inside an enclave. Fortunately, system calls are typically only invoked by libraries, so we can focus on just modifying the libraries dynamically linked by the interpreter by statically linking customized, enclave-aware versions of these libraries into the enclave shared object alongside the interpreter. Therefore, inspired by SCONE's design, we can just add a software shim layer atop the standard libraries. This shim layer will intercept the execution of the system calls inside the enclave and transparently redirect their execution to the outside world through bridge functions supported by SGX SDK.

In our threat model, the client (script provider, such as a game publisher) does not trust the remote platform (script executor, such as game players). However, the client still needs to verify that the remote platform is indeed running a secure enclave. Fortunately, Intel provides remote attestation for enclaves, in which Intel's attestation servers can verify that a remote enclave is secure. For the reverse case, in contrast, the remote platform in our threat model does trust the client to send non-malicious scripts. We have chosen this model because in the case of computer games, the scripts are being sent from the game servers to the player's machine. The game creator has created the software running on both platforms, but the game servers remain in the control of the developers, and the developers are trusted in our model. However, blindly trusting the client is not possible, as the remote platform must first verify that it is talking to a legitimate client. Therefore, the remote platform must have prior knowledge of the identity of the client. We solve this problem by hard-coding the public key of the client into the secure enclave.

ScriptShield Overview. An overview of ScriptShield is presented in Figure 1. There are two parties involved in executing a script: a client who would like to send a script to be executed, and a remote party out of the client's control that securely executes the script. The client could be a cloud user, a game publisher, or a web service provider. The remote party could be a cloud provider, a game player, or a web browser. From now on, we will refer to the party who wants the code protection as the *client*, and refer to the party that executes the client's code as the *remote platform*.

The client first launches the Initialization component, which performs remote attestation to make sure the remote platform is running the intended interpreter. It then generates shared secrets to establish a secure channel, which will be used to transfer the encrypted script to the remote party.

On the remote platform side, the Untrusted Module first creates the secure enclave to execute the interpreter and provides wrappers for all enclave `ecalls` and `ocalls`. After that, the interpreter executes as a daemon process and waits for client requests. Upon receiving the initialization request from the client, the remote platform creates an attestation report for the secure enclave, forwards the report to be signed by the Quoting Enclave, and returns the signed attestation report back to the client. If attestation succeeds, then the client can establish a secure channel with the secure enclave. A common approach to do so is to use the Diffie-Hellman key exchange protocol, where the public key of the secure enclave (generated from random sources after it is launched) is cryptographically bound to the attested data of the attestation report, and the public key of the client is hard-coded inside of the enclave to authenticate the client.

Next, the script is passed via the secure channel to the Script Decryption component of the remote platform, and is decrypted inside the enclave. Since no system calls can be executed inside the enclave, during the execution of the script, ScriptShield has to execute system calls outside the enclave. The control flow passes through our System Call Shim layer inside the enclave to the outside `ocalls`. ScriptShield also checks system call return values in order to protect against Iago attacks.

4 DESIGN

4.1 Client Side Initialization

Remote Attestation. ScriptShield requires remote attestation, which is the mechanism by which a third party can verify that the desired software is indeed running inside an enclave on an Intel SGX enabled platform. During remote attestation, the untrusted component that hosts the enclave asks the enclave to produce a report called a quote to identify the platform. A quote containing information about the measurement of code and data, the product ID, security version number and other attributes is securely presented to the client for verification. In particular, the remote attestation in ScriptShield takes the following four steps:

- **Step-I**: Before the client can safely send a script to the remote server, the client first needs to issue a challenge to the remote party to ensure that it is indeed running the necessary components inside one or more enclaves.
- **Step-II**: The interpreter's enclave generates a report in response to the challenge. The report includes the security version number, enclave attributes, enclave measurement, software version, software vendor security version number, and additional user-provided data. The quoting enclave verifies and signs the report, and the signed report, called a quote, is sent to the client.
- **Step-III**: The client sends the signed report (quote) to the Intel Attestation Service (IAS), and the IAS will verify the quote.

- **Step-IV**: IAS replies with an atttestation verification report which confirms or denies the authenticity of the quote and the enclave it originates from.

Cryptographic Protection. Since the interpreter needs to retrieve scripts from outside the enclave, to protect the script's integrity and confidentiality, ScriptShield needs to establish a secure channel between the client and the remote enclave. More specifically, the secure channel needs to satisfy the following requirements: (1) the client needs to verify the code running inside the enclave (i.e., the identity of the enclave) is exactly the same as expected, (2) the enclave must authenticate the client, and (3) the script is encrypted before it is sent to the enclave. As a result, the client only sends scripts to a trusted enclave, the enclave only accepts scripts from authenticated clients, and the remote platform never learns the content of the encrypted scripts.

Intel's SGX SDK provides a trusted cryptography library (called `sgx-tcrypto`) that includes the cryptography functions used by trusted libraries included in the SDK, such as the `sgx-tservice` library. The Diffie-Hellman key exchange method is a widely used approach to securely exchanging cryptographic keys over a public channel, allowing two parties with no prior knowledge of each other to jointly establish a shared secret key over an insecure channel. However, it is well-known that the Diffie-Hellman protocol is vulnerable to man-in-the-middle attacks, especially when the OS in the remote platform is not trusted. To defend against this attack, a Diffie-Hellman key exchange can be done along with remote attestation. Because a 64-byte attested data can be associated with the attestation report, the hash value of the enclave's public key can be embedded into the attested data, so that the public key is cryptographically bound to the enclave's identity. Moreover, we hard code the public key of the client in the enclave for the enclave to authenticate the client in order to prevent an attacker from impersonating the client. After the client and the enclave have established a secure communication channel, the client then can encrypt the script and securely pass it to the enclave.

4.2 Remote Platform Execution

At the remote party (i.e., the SGX machine), we need to execute scripts sent by the client. As shown in Figure 1, a typical SGX process involves an Untrusted Module and a Trusted Module.

Untrusted Module. Launching an enclave requires an untrusted component to set up the necessary environment, provide necessary libraries (e.g., the SGX runtime library) and bridge functions for the enclave, and then pass control to the enclave.

In addition to establishing the communication channel for remote attestation with the client, our Untrusted Module contains the SGX Runtime Library provided by the SGX SDK and provides `ocalls` that can be called by the enclave. The enclave exposes an `ecall` interface that can be called from untrusted code. Once remote attestation succeeds, the client's script is passed to the enclave and the untrusted module hands off control flow to the enclave via an `ecall`, which will start the execution of the interpreter.

Trusted Module. The interpreter of the corresponding scripting language is executed inside the SGX enclave. With this, script code and data are automatically protected. However, the challenge lies in how to execute the interpreter inside the enclave, considering the fact that enclaves cannot perform any system calls. Certain system calls, such as those related to networking and file systems, are crucial for the correct execution of many scripts.

At first it might appear that we would need to modify the source code of each interpreter. While possible, this approach is quite tedious and not scalable if we aim to support many interpreters. Fortunately, an interpreter is typically dynamically linked to the standard C libraries. Based on this observation, if we just statically link the interpreter with a modified version of libc, then we do not have to modify the interpreter. Therefore, the challenge now becomes how to modify the way system calls are invoked in libc so that we can statically link our modified libc with the enclave code and execute it inside the enclave.

Identifying the points of system call invocation fpriat the function level in libc would also be tedious. For instance, both `printf` and `fprintf` will invoke the `write` system call. Directly examining all of the standard library code, identifying functions invoking system calls, and adding a bridge function for each of these functions would require a lot of manual effort. Therefore, we need a more systematic approach. Fortunately, we found that in the `musl-libc` implementation of libc, all system calls are invoked from a central location rather than independently. Therefore, by directly patching the centralized system call code, there is no need to examine and patch all `libc` functions.

More specifically, we just need to add a shim layer atop the existing system call invocation point in `musl-libc`. At this shim layer, based on the system call number we invoke the corresponding bridge function (the `ocall` wrappers) and execute the system call outside of the enclave. The interpreter can now be statically linked with our modified C library and then executed inside the enclave.

Our Trusted Module also needs to decrypt the script. The decryption key is passed by the client through the secure channel. After decryption, the script can be executed. Considering that the enclave does not trust the OS in our threat model, we cannot rely on the return values from system calls, as a malicious OS can return arbitrary values. This is a well-known class of attacks called Iago attacks [5], in which malicious return values from system calls could lead to arbitrary execution inside the enclave. Fortunately, we are not the first to encounter this problem, and there is a known defense: performing checks on the system call return values inside the trusted component, as has been done in Haven [4]. For example, for a `read` system call, the return value contains the number of bytes read. However, one of the arguments is `count`, the number of bytes to attempt to read. By checking that the return value is not larger than `count`, we can verify the return value is in a legal range. Therefore, we insert such verification code inside the enclave for each system call in order to defend against Iago attacks.

5 EVALUATION

We have implemented ScriptShield[1]. In this section, we present the evaluation results. Our performance experiments were executed on a 14.04.1-Ubuntu system comprised of a 4-core Intel Core i7-6700 CPU running at 3.40GHz, with 64GiB memory, and 1Gbit/s Ethernet Connection I219-LM, running the latest Intel SDK and SGX

[1]The source code is available at https://github.com/osuseclab/scriptshield.

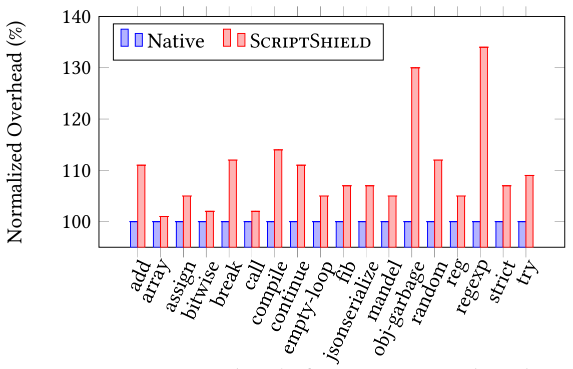

Figure 2: Percentage overhead of running MuJS benchmarks in ScriptShield normalized against native execution.

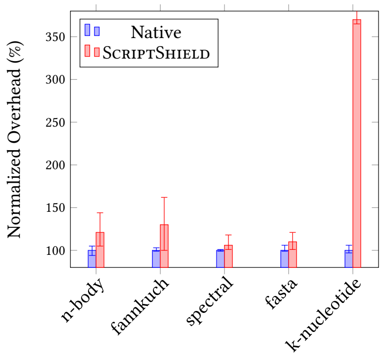

Figure 3: Percentage overhead of running lua programs in ScriptShield normalized against native execution.

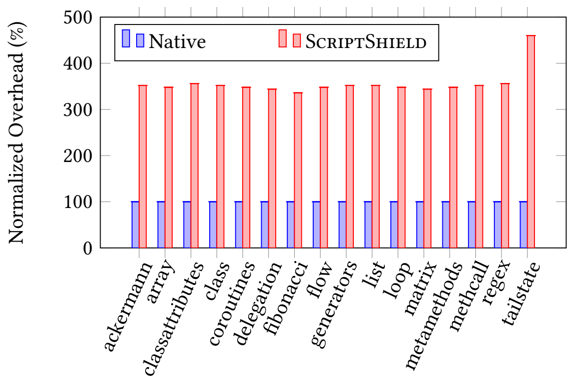

Figure 4: Percentage overhead of running Squirrel benchmarks in ScriptShield normalized against native execution.

driver. We set up two environments for running our benchmarks: (1) running natively, (2) running under ScriptShield. We measured the execution time by utilizing the OS clock outside of the enclave. In particular, we first start the clock outside an enclave, execute a script inside the enclave, and then stop the clock to calculate the total execution time.

We evaluated ScriptShield using three popular cross-platform scripting languages: Lua, JavaScript (by running the MuJS interpreter), and Squirrel. We do not need to change any lines of code in the interpreter when porting it to run in our enclave. What we must do is (1) add an `ecall` as an entry point to call the interpreter, (2) modify the interpreter makefiles to statically link `musl-libc`, and (3) modify the enclave makefiles to integrate the statically linked interpreter into the enclave and generate the final `enclave.so` with the client's hard coded public key.

Since ScriptShield is legacy code compatible, we can directly run unmodified existing scripting language benchmarks to evaluate the performance impact of ScriptShield when running the interpreter inside the enclave. To this end, we have selected the following benchmark scripts for each of the tested interpreters:

- **Lua**. Our dataset includes 5 standard benchmarks used in the Lua language evaluation, as shown in Figure 3.
- **MuJS**. MuJS is a lightweight JavaScript interpreter. There are no specific benchmarks for the MuJS engine, so we used the benchmarks from the performance tests in `Duktape`. We grouped these benchmarks into 18 sets, shown in Figure 2. Each set contains one or more JavaScript benchmarks and tests the performance of the language feature specified by that group.
- **Squirrel**. Squirrel is a Lua-like language with a C-like syntax, typically embedded in a host application. Unlike Lua, which is written in C, Squirrel is implemented in C++ but exposes a C API modeled after Lua's stack-based API. We have 15 sample benchmarks that we run, as shown in Figure 4. .

Results. Figure 2, Figure 3, and Figure 4 show the percentage overhead of running the MuJS, Lua, and Squirrel benchmarks within ScriptShield compared with native execution. We observe that the overhead for the Lua benchmarks in Figure 3 is reasonable except for `k-nucleotide`. However, this apparently high overhead is due to the short runtime of the benchmark. Therefore, any fixed overhead (e.g. entering and exiting the enclave) may dominate the runtime of the benchmark when running under ScriptShield. This leads to a high overhead percentage, even though the actual time it takes to execute is quite small, both natively and in ScriptShield. The overheads for Squirrel scripts in Figure 4 appear high for the same reason: the benchmarks' very short runtimes result in higher overhead. In contrast, the MuJS benchmarks take longer to execute and have comparatively lower overhead, as shown in Figure 2.

6 LIMITATIONS AND FUTURE WORK

There are still limitations to ScriptShield. Currently, we only support applications written purely in scripting languages. We do not directly support applications with an interpreter depending on native components (e.g., a web browser, in which both native code and scripting code work together). We leave the additional porting efforts to support the native components to future work.

We only ported three interpreters so far. While we do not have to modify the interpreter source, we must recompile and statically link it with our modified `musl-libc`. This process often involves tedious engineering efforts. A next step is to investigate automatic compilation dependency resolution and makefile patching, so we can automatically recompile and statically link with our new library.

Finally, we did not implement bridge functions and `ocalls` for every system call. Currently, we support 60 commonly used system

Systems	Year	Backward Compatibility	Small TCB	Binary Apps	Scripts
Haven [4]	2015	✓	✗	✓	✗
Scone [2]	2016	✓	✓	✓	✗
Panoply [20]	2017	✓	✓	✓	✗
TrustJS [8]	2017	✗	✓	✗	✓
Glamdring [15]	2017	✗	✓	✓	✗
Graphene [21]	2017	✓	✗	✓	✗
ScriptShield	2019	✓	✓	✗	✓

Table 1: Comparing ScriptShield **with the related works.**

calls (all of which are used in our testing benchmarks). We plan to add support for the rest of the system calls as future work.

7 RELATED WORK

Protecting Applications with SGX. Since SGX offers strong protection to applications, SGX has been used to build several security systems. Haven [4] ports a Windows library OS to SGX to achieve shielded execution of unmodified legacy applications. Haven has a large TCB and thus a large attack surface. Panoply [20] provides a micro-container isolating data and code with SGX. Panoply bridges the gap between SGX-native abstractions and standard OS abstractions, but it must change the application code. Scone [2] isolates docker containers running on a public cloud by using SGX enclaves. Ryoan [11] protects secret data in a distributed sandbox while it is processed by services on untrusted platforms, leveraging hardware enclaves to protect sandbox instances from potentially malicious platforms. SGX-Elide [3] protects the secrecy of SGX code itself by enabling the dynamic update of the enclave code. Glamdring [15] automatically partitions applications into trusted and untrusted parts.

TrustJS [8] explored the direction of using SGX to protect JavaScript. While it made a first step of using SGX to protect scripting languages, their design requires script modifications. Also, it does not attempt to provide a general execution framework to execute legacy applications developed in scripting languages. In contrast, ScriptShield aims to protect the confidentiality and integrity of scripting languages without script modifications.

A comparison of ScriptShield with closely related works is illustrated in Table 1. ScriptShield is the first scheme with both a small TCB and backward compatibility, while all other schemes either focus on binary applications or lack backward compatibility.

SGX Attacks and Defenses. One of the earliest attacks on SGX is the controlled-channel attack [23], in which a malicious OS infers the secrets of SGX applications by observing page fault patterns. Recently, even higher resolution side channels have been found by exploiting timer interrupts and cache misses [9]. Other attempts to attack SGX include using ROP [13] or branch shadowing [14].

To counter controlled-channel attacks, T-SGX [18] leveraged transactional memory and compiler extensions to instrument enclave code and detect attack attempts. There are also other solutions that mask page fault patterns by either determining memory access behavior [19] or using large pages [7].

8 CONCLUSION

We have presented ScriptShield, a backwards compatible application execution framework that automatically ensures the confidentiality and integrity of scripts by executing scripting language interpreters inside SGX enclaves. We have implemented a prototype of our framework and tested with Lua, JavaScript and Squirrel. Our experimental results show that ScriptShield does not introduce noticeable overhead. We also demonstrate the benefits of running interpreters inside SGX, such as only signing the trusted enclave once while allowing execution of arbitrary scripts and transparent protection of the confidentiality and integrity of scripts.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by the NSF grants 1750809, 1718084, 1834213, 1834215, and 1834216.

REFERENCES

[1] "Intel software guard extensions programming reference," https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.
[2] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx." in *OSDI*, 2016.
[3] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "Sgxelide: enabling enclave code secrecy via self-modification," in *CGO*. ACM, 2018, pp. 75–86.
[4] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, 2015.
[5] S. Checkoway and H. Shacham, "Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface," pp. 253–264.
[6] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," in *EuroS&P'19, year=2019*.
[7] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults," in *RAID'17*, 2017.
[8] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "Trustjs: Trusted client-side execution of javascript," in *EuroSec*, 2017.
[9] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *USENIX ATC 17*, Santa Clara, CA, 2017.
[10] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, 2009.
[11] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *TOCS'18*, vol. 35, no. 4, p. 13, 2018.
[12] Intel, "Intel Software Guard Extensions Programming Reference (rev1)," Sep. 2013, 329298-001US.
[13] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *USENIX Security 17*, 2017.
[14] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *USENIX Security 17*, 2017.
[15] J. Lind, C. Priebe, D. Muthukumaran, D. OâĂŹKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for intel sgx," in *USENIX ATC 17, Santa Clara, CA*, 2017.
[16] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagankar, "Innovative instructions and software model for isolated execution," in *HASP'13*, Tel-Aviv, Israel, 2013.
[17] J. K. Ousterhout, "Scripting: Higher level programming for the 21st century," *Computer*, vol. 31, 1998.
[18] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs," in *NDSS'17, San Diego, CA*, 2017.
[19] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Asia CCS'16*. ACM, 2016, pp. 317–328.
[20] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," *National University of Singapore, Tech. Rep*, 2016.
[21] C. Tsai and D. Porter, "Graphene-sgx," https://github.com/oscarlab/graphene.
[22] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *ACM CCS*. ACM, 2017, pp. 2421–2434.
[23] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*, 2015.

A ADDITIONAL PERFORMANCE RESULT

When a scripting language interpreter is executed inside a secure enclave, the script execution incurs the extra cost of running enclave boundary operations and memory access operations (due to running inside an MEE). Enclave boundary operations are required for communication between trusted enclave code and untrusted code. Such operations include data transfer and control transfer operations. Enclave boundary transfer operations are necessary to perform system calls. Similarly, since SGX protects data using memory encryption, read and write memory access operations incur additional overhead. Thus, we would like to closely examine the overhead of these operations by designing corresponding micro benchmarks.

In particular, we designed two micro benchmarks in Lua to evaluate ScriptShield's performance. Our first micro benchmark evaluates system call overhead, which is an important part of ScriptShield's design, while the other benchmark measures memory access overhead with memory copy operations. Our language choice is less important here because there is very little script code executing; the overhead mostly comes from the system calls or memory accesses. However, it is important to note that these benchmarks are still taking place within an interpreter, run natively and in ScriptShield.

System Call Overhead. System calls are used in essentially all applications, but are not allowed inside SGX enclaves. A key component in ScriptShield is the shim layer atop `musl-libc` that redirects all system calls from the enclave to ocalls that perform the actual system call outside the enclave. Since the system call shim is such a significant part of our framework, we would like to quantify the system call overhead. Clearly, we cannot test all of the system calls we implemented due to the challenge of designing the corresponding test cases. Therefore we would like to focus on the most representative system calls.

Interestingly, we notice `readv` and `writev` are the two most frequently used system calls in many of our daily applications. We thus design two benchmarks that write to a file and read from a file, which ultimately invoke the `readv` and `writev` system calls, respectively. We use different input data sizes for this performance test.

Figure 5 shows the execution time comparison of running the `readv` and `writev` system calls 1,000 times in ScriptShield, normalized against native execution. ScriptShield actually has no noticeable overhead for `readv`, while overhead is reasonable for `writev`.

Memory Access Overhead. SGX uses a memory encryption engine (MEE) to encrypt the memory for the enclave. Any access inside the enclave will retrieve the plaintext whereas any access outside the enclave will only obtain the ciphertext. Naturally, this comes with a cost. In particular, when an enclave reads data from and writes data into enclave memory, the requested or updated pages are accessed by or written into the page cache. For cache misses or when a page needs to be written into memory, the Memory Encryption Engine (MEE) has to encrypt and decrypt the cache line because of the EPC page protection mechanism. When enclave code memory requirements exceed the EPC size, overhead will be very large.

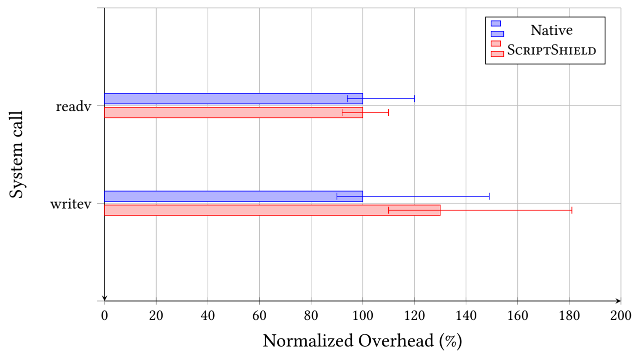

Figure 5: Percentage overhead of running `readv` and `writev` benchmarks in ScriptShield normalized against native execution.

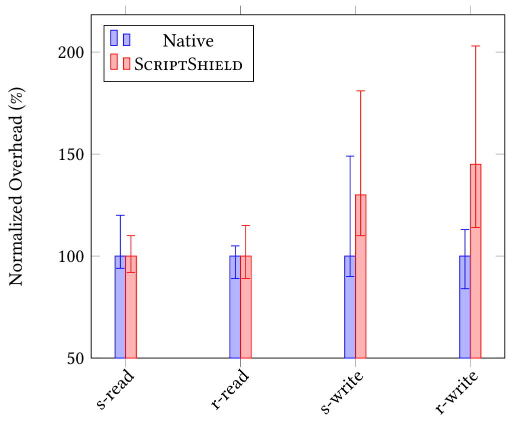

Figure 6: Percentage overhead of running memory access benchmarks in ScriptShield normalized against native execution.

To measure the memory access overhead inside the enclave, we leveraged our system call testing and designed another micro benchmark to both sequentially and randomly read and write enclave memory because sequential and random memory accesses have different cache miss frequencies. More specifically, this benchmark measures the time for both sequential and random read/write operations, normalized against a deployment without an enclave. All operations access a 256K memory region. Figure 6 shows the percentage overhead of running memory access benchmarks within ScriptShield normalized against native execution. We can see ScriptShield does not significantly impact memory read and write performance.

B SECURITY APPLICATIONS

Once we have enabled running the legacy scripting code inside SGX enclaves, we are now able to protect the secrecy and integrity of this code without modifying any script code. To demonstrate how to use ScriptShield to protect (legacy) scripting languages, we next present a few security applications.

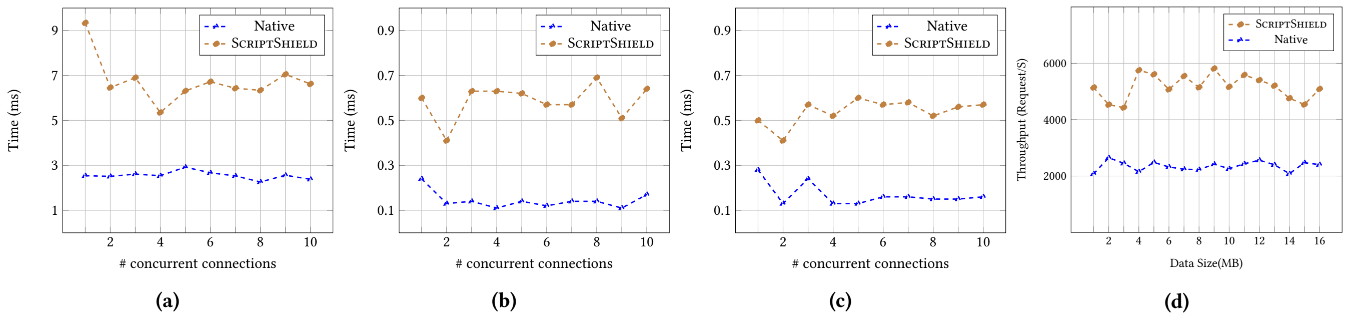

Figure 7: Average time required to run Xavante benchmarks' (a) `loop` script, (b) sample `html` file and (c) 10 MB text file without (native) and with ScriptShield. (d) Average throughput measurement for Xavante without (native) and with ScriptShield.

Protecting HTTP Servers. When running applications in a cloud environment, customers would like to protect the confidentiality and integrity of their code and data. ScriptShield's ability to easily place an interpreter in an enclave is ideal for securing applications with an embedded scripting engine. A prominent example of this kind of application is HTTP servers, many of which offer support for multiple scripting languages for server-side scripting. Scripts are more flexible, dynamic, and are easier to use to build web applications than compiled code. Because of this, many websites use scripting languages for their web servers.

Lua has been used for server-side scripting on several popular web sites, including Taobao, and as a templating language on MediaWiki. Therefore, we decided to demonstrate protecting lua scripts running on an HTTP server. Xavante is an HTTP server written entirely in Lua that also offers support for Lua scripting. Because of this, we chose to use Xavante to demonstrate ScriptShield. Since Xavante is written in Lua, we can protect more than just server-side scripts; we can place the entire web server inside the enclave, ensuring its integrity in addition to the confidentiality and integrity of the scripts it runs.

Xavante has a modular architecture based on URI mapped handlers. It offers a file handler, a redirect handler and a WSAPI handler. To use Xavante, we just need to configure it (no code modification at all) and then execute it in the Lua interpreter we have ported. We also used a set of benchmarks to evaluate the overhead while running Xavante with ScriptShield.

Specifically, we have three benchmarks: (1) running `loop.lp`, which is a lua script running on the Xavante server that performs a simple loop and displays the result on a web page, (2) accessing the default index.html hosted by Xavante, and (3) accessing a 10MB file hosted by Xavante. We use `ab`, the Apache HTTP server benchmarking tool, to run these tests. Also, we measure ScriptShield throughput overhead by running `ab` with varying amounts of concurrent connections to the server. We show the results of the three benchmarks in Figure 7. From Figure 7, the average overhead of `loop.lp` is around 2 ms, the average overhead of the `sample html file` is around 0.5 ms, and the average overhead of the `10 MB text file` is around 0.4 ms. The overhead of these three benchmarks is mostly due to the short runtime of the benchmarks. Therefore, any fixed overhead (e.g. entering and exiting the enclave) may dominate the runtime of the benchmark when running under ScriptShield. This leads to a higher overhead percentage, even though the actual time it takes to execute is quite small, both natively and in ScriptShield. Xavante scales well (as the overhead does not increase with an increase in concurrent connections), while the whole HTTP server remains protected within a secure enclave.

Protecting Computer Games. Computer games can often be reverse engineered. There is often an arms race between game publishers and game cheaters in protecting the secrecy of certain game logic. Also, many of today's computer game engines use scripting languages such as Lua and Squirrel for game logic. Since we have enabled running the Lua interpreter in ScriptShield, we can then offer the capability to protect the game logic against reverse engineering.

LÖVE is a game engine used to make 2D games with Lua, and has been used for commercial projects. We managed to execute LÖVE inside ScriptShield so we are able to run games written in Lua. Due to the fact that different games require different versions of LÖVE, we just run Mr. Rescue for this proof of concept to demonstrate how ScriptShield could protect the game logic from malicious attacks. Mr. Rescue is written purely in Lua. Due to the highly interactive nature of computer games, we did not attempt to measure its performance overhead, but we did observe there is no user experience differences when compared using ScriptShield or not.

Since our framework is designed specifically for running scripts, applications written purely in a scripting language are the most straightforward to run in our framework. For applications that are only partly written in a scripting language, extra efforts are needed, as the interactions between the scripts and native application components need to be handled.