Binary Code Retrofitting and Hardening Using SGX

Shuai Wang[1], Wenhao Wang[2,3], Qinkun Bao[1], Pei Wang[1], XiaoFeng Wang[2], and Dinghao Wu[1]

[1]The Pennsylvania State University
[2]Indiana University Bloomington
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
szw175@ist.psu.edu,ww31@indiana.edu,{qub14,pxw172}@ist.psu.edu,xw7@indiana.edu,dwu@ist.psu.edu

ABSTRACT

Trusted Execution Environment (TEE) is designed to deliver a safe execution environment for software systems. Intel Software Guard Extensions (SGX) provides isolated memory regions (i.e., SGX enclaves) to protect code and data from adversaries in the untrusted world. While existing research has proposed techniques to execute entire executable files inside enclave instances by providing rich sets of OS facilities, one notable limitation of these techniques is the unavoidably large size of Trusted Computing Base (TCB), which can potentially break the principle of least privilege.

In this work, we describe techniques that provide practical and efficient protection of security sensitive code components in legacy binary code. Our technique dissects input binaries into multiple components which are further built into SGX enclave instances. We also leverage deliberately-designed binary editing techniques to retrofit the input binary code and preserve the original program semantics. Our tentative evaluations on hardening AES encryption and decryption procedures demonstrate the practicability and efficiency of the proposed technique.

1 INTRODUCTION

With the increasing needs to deploy applications on the third-party untrusted environments (e.g., cloud), software attacks are widely launched to steal privacy information from the victim programs. Cutting-edge code reuse attacks exploit software vulnerabilities (e.g., buffer overflow) and leverage code snippets in the victim program to undertake attack activities. In addition, many sophisticated attackers can (indirectly) inspect the execution behavior of victim programs and reveal secret information of the running process.

A promising direction to protect benign applications from cutting-edge threats (e.g., code reuse attacks) is to execute programs in the Trusted Execution Environment (TEE), such as Intel Software Guard Extensions (SGX). In general, SGX technique provides a secure memory region (called SGX enclave) in the process address space where the application code and data can be executed safely. Hardware guarantees the isolation of the enclave instances, and it also encrypts the memory pages before storing on the disk.

FEAST'17, November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5395-3/17/11…$15.00
https://doi.org/10.1145/3141235.3141244

To protect (legacy) binary code from adversaries using SGX techniques, existing research has proposed techniques to put the entire executable file into the enclave for execution [2, 4, 28]. By providing rich sets of system supports (e.g., a library of OS facilities), legacy binaries can be directly executed inside SGX enclaves without modifications for most of the cases. However, a general concern for such "heavy-weight" approaches is that usually the Trusted Computing Base (TCB) is largely increased, which potentially violates the principle of least privilege [16, 22]. We also notice some recent work proposing to partition the software system into several components according to their dependency on program secrets [16]; code components that depend on the program secrets would be protected by the SGX enclave instances. While the overall approach shall notably decrease the size of TCB, their approach is designed to instrument source code, which limits its application scope since there are large amount of legacy binaries in the wild.

In this research, we propose novel techniques to perform binary retrofitting and harden (security sensitive) functions with SGX enclaves. Our technique is designed to dissect binary code into multiple components; each component contains one or several functions, and each component will be put into an SGX enclave instance for protection. To deliver a flexible design, we create an SGX "interface" library for each enclave, where "interface" functions are provided to perform SGX enclave initialization, teardown as well as invoking each protected function in the corresponding enclave instance. The input binary code is instrumented, where the body of protected functions are rewritten into "trampoline" code and the assembly instructions of the protected functions are built into SGX enclave instances. During runtime, the "trampoline" code redirects the control flow from the original function entry point into the corresponding SGX interface library, and further reach the protected functions in the SGX enclave instance. Also, since many challenges in binary retrofitting are fundamentally undecidable, we propose well-designed exception handling techniques to capture and fix execution errors, which deliver a faithful runtime behavior.

To protect security sensitive functions in cryptosystems, we evaluate our technique towards a widely-used cryptographic algorithm implementation, i.e., the AES implementation in OpenSSL. Our preliminary evaluations protect AES core encryption and decryption procedures, and the experimental results have demonstrated the feasibility and efficiency of the proposed technique.

2 BACKGROUND OF INTEL SGX TECHNOLOGY

Software Guarded Extensions (SGX) is a set of extended x86 instructions that provide isolated execution environments, called enclaves, within a single process. Technically, an enclave run in the user mode (ring 3) of an Intel CPU, but the SGX hardware guarantees that accesses from outside are always properly mediated even if the entire software stack, including the operating system, driver, BIOS and VMM, is compromised. In other words, the CPU and code inside an enclave are enough to form a Trusted Computing Base (TCB). A physical memory region called Processor Reserved Memory (PRM) is reserved exclusively for SGX execution. SGX enforces page based access control by extending the processor's Page Miss Handler (PMH). The Enclave Page Cache (EPC) resides in the PRM and holds enclave code and data. The system software is in charge of managing EPC pages for enclaves. While the system software is untrusted, SGX tracks the properties of EPC pages through a data structure called the Enclave Page Cache Metadata (EPCM), which is also located in the PRM.

The functionalities of SGX are encoded as leaf functions within the `ENCLS` (enclave supervisor) and `ENCLU` (enclave user) instruction mnemonics. The system software uses the `ENCLS` instruction to invoke the specified privileged leaf function for managing and debugging code in enclaves. Users utilize the `ENCLU` instruction to invoke the specified non-privileged leaf functions for enclave state transitions and retrieving key materials inside the enclave. The life cycle of an enclave begins with the creation of an SGX Enclave Control Structure (SECS) page for the enclave when the system software issues the `ECREATE` leaf function; The system software then uses the `EADD` leaf function to load initial code and data into the enclave. While loading the enclave, the system software also updates the enclave measurement using the `EEXTEND` leaf function. After the enclave code and data are loaded, the system software can execute the `EINIT` leaf function to mark the enclave as initialized and finalize the enclave measurement to establish the enclave identity. The correctness of the enclave can be verified by generating a cryptographic report of the enclave measurement with the `EREPORT` leaf function and attested to a local enclave through local attestation or to a remote party through remote attestation. The ring-3 application is now allowed to perform a controlled jump into the enclave code using `EENTER` leaf function; `EENTER` also switches the processor to enclave mode. The enclave code can use the `EEXIT` leaf function to return the execution to the host application. Whenever an exception or fault occurs inside the enclave, the processor performs an Asynchronous Enclave Exit (`AEX`) before invoking the system software's exception handler. The `AEX` saves the enclave state in the enclave's State Save Area (SSA) frame, restores the state saved by `EENTER` and changes the instruction pointer (`RIP`) to point to a trampoline area (referred to as the asynchronous exit handler) in the host application. The `RIP` is pushed onto the stack before jumping to the system software's exception handler. As a result, after the exception is handled by the system software, execution is returned to the trampoline area which is expected to return to enclave mode and continue the computation using the `ERESUME` leaf function. An enclave is destroyed with the `EREMOVE` leaf function after the EPC pages are deallocated by the system software.

3 RELATED WORK

SGX has been receiving much attention since it was introduced as it is widely supported on commodity CPU hardwares. Extensive works have been published to enable the quick deployment of various applications on SGX. The library OSes, Haven [4] for Windows and Graphene-SGX [28] for Linux, enable running unmodified binaries inside SGX enclaves. Scone [2] uses SGX to protect Linux container processes with a small TCB and a low performance overhead. Eleos [19] implements exit-less system calls and exit-less paging in enclaves. It introduces a Secure User-managed Virtual Memory (SUVM) abstraction that implements application-level paging inside the enclave to reduce the overhead of enclave exits due to paging. PANOPLY [27] provides a new abstraction called a micro-container which is a unit of code and data isolated in the enclaves. It has a minimized TCB and yet offers rich OS abstractions to enclave code. SGX has also been used to enable trustworthy data analytics in the cloud [23] and secure isolation of the states of Network Function Virtualization (NFV) applications [24], to enhance the security and privacy of Tor [14], to protect distributed sandbox instances [13] and data-oblivious machine learning algorithms [18] from potentially malicious computing platforms.

Lind *et al.* propose an automatic source-level partitioning framework called Glamdring [16]. A developer first annotates security-sensitive application data. Glamdring then automatically partitions the application into untrusted and enclave parts, places security-sensitive functions inside the enclave, and adds runtime checks and cryptographic operations at the enclave boundary to enforce the confidentiality of sensitive input and integrity of sensitive output.

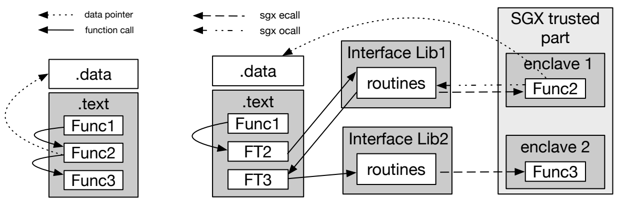
Intel states that SGX does not provide explicit protection from side-channel attacks. It has been shown that SGX suffers from a type of controlled-channel attacks [34] and a list of microarchitectural attacks [5, 10, 15, 17]. Defenses are proposed to detect the frequent AEXs in these attacks using Transactional Synchronization Extensions (TSX) [25] or checking program execution time in its control-flow graph [8]. A recent work [33] further demonstrates that a page-level attack can still steal information from the enclaves without inducing a large number of `AEX`s.

4 DESIGN

In this section we outline the design of the proposed technique. To this end, we first present an instrumentation example and explain each component of a typical instrumentation product. We then present overview of each instrumentation step.

Instrumentation Example In general, the proposed technique supports flexible configurations to utilize SGX techniques. That is, users can configure our tool to put one or multiple functions into different SGX enclaves. The Intel SGX SDK provides a set of routine functions to support SGX enclave initialization, destruction, access control, and other security-related operations. To provide a flexible and mostly reusable design, functions belonging to the same enclave are associated with a set of such standard routines. Each set of routines are compiled into one shared library, providing interface functions to invoke the protected code in the enclave instances. By maintaining the interface of each enclave as a shared library, the protected functions become mostly "reusable" to provide functionalities secured by SGX.

Figure 1 presents an instrumentation example, in which the input binary is compiled from a program of three functions (Figure 1a). In this example we put two functions (`Func2` and `Func3`) into two enclaves, separately. The instrumentation output is shown in Figure 1b. As previously presented, we maintain common routine code for each enclave as one shared library (second column in Figure 1b); "interface" functions to trigger protected code in the associated enclave of the shared library are exported (such library is denoted as *SGX interface library* later in this paper). The original content of both functions are rewritten into trampolines (`FT2` and `FT3`),



(a) Input Binary. **(b)** Instrumented Outputs.

Figure 1: Example of binary instrumentation using SGX enclaves. Two functions (`Func1` and `Func2`) are put into two enclaves for protection.

which forward function calls to corresponding interface code, and further to code in the enclaves. While control transfers between the application code and each shared library are normal function calls, specific SGX instructions are used to bridge libraries and enclave instances (i.e., through SGX `ECALL` and `OCALL`).

Instrumentation Overview To perform SGX-based binary instrumentation, we first launch in-place binary editing to rewrite several leading bytes of the target binary functions into trampolines; trampoline code will redirect control-flow transfers to its corresponding SGX interface library and further invoke the protected function in the enclave instance. Well-designed in-place binary rewriting can preserve the original binary context, and hence delivering a faithful rewritten output (§4.1). For each protected function, we perform disassembly and recover their assembly instruction sequences. We then launch a set of analysis passes to recover higher-level information (e.g., function prototype) of the protected function; such information is critical to preserve the functionality correctness and support SGX access control in the instrumented output (§4.2). Furthermore, considering the general difficulty in retrofitting binaries, it is not inaccurate to assume that some data or code pointers would become broken due to the relocation of the SGX-hardened functions. Hence, we propose a deliberately designed exception handling mechanism to catch and process potential runtime exceptions (§4.3).

Scope and Limitations Our tool is mainly designed to protect legacy binaries on x86 platforms. The instrumented binary can benefit from the SGX technique which is widely supported by recently-released Intel hardware. In this research we propose to perform function-level protections. As Intel SGX supports to execute arbitrary length of code components, it should be also interesting to investigate the feasibility on protecting finer-grained code snippets, such as critical control predicates.

Our tool is designed to protect legacy code, including stripped binaries containing no or minimal debug and relocation information. While function information is mostly absent in such binaries, recent work has made promising progress in this direction [3, 26, 32]. Hence, in this research we assume the function information is available before instrumentation. The current implementation instruments 64-bit ELF binaries since ELF is the default format on Linux platforms and 64-bit is the mainstream. Nevertheless, the proposed technique is independent with the underlying architecture details, and hence not difficult to port to other platforms, like 32-bit Linux.

```
trampoline_foo:
  push   %rbp
  mov    %rsp,%rbp
  push   $return_addr
  push   %rax
  mov    $sgx_interface_foo,%rax
  xchg   %rax,(%rsp)
  ret
  pop    %rbp
  ret
```

Figure 2: Trampoline Code for In-Place Binary Editing of Function `foo`.

4.1 Binary Editing

We now elaborate on the design of the binary editing process. In general, code components in (stripped) binary code are pointed to through concrete addresses (in terms of absolute or relative addresses). Hence, binary code is generally considered “un-relocatable”; any manipulation that changes the relative position of binary components can potentially break the pointers of concrete addresses in the context and lead to ill-functionalities during runtime.

In general, existing binary instrumentation approaches can be divided into several categories. The first category delivers in-place editing which performs byte-level binary rewriting and can preserve the original positions of all reachable binary components in the context [20]. Some other commonly-used techniques perform patch-based or replica-based instrumentation [6, 11]. Wrappers are broadly adopted to redirect the control transfers and preserve the original semantics, which can usually lead to non-trivial execution slowdown or binary space increase. We also notice a number of recent techniques proposing to fully recover relocation information before instrumentation; once the code components become relocatable, it shall be safe to perform arbitrary instrumentation [29, 30].

Since one promising application of our technique is to harden cryptosystems, where complex data structures and control flows shall exist, in this research in-place binary editing is adopted for the seek of delivering a conservative and faithful instrumentation. In general, for each function that is going to be placed inside the SGX enclave, we analyze the input binary and locate its position in the binary code. We then rewrite the starting bytes of the target function with a trampoline routine; this routine will forward the execution to the corresponding SGX interface library and further into the enclave. By editing the beginning bytes of the target functions, it is safe to assume that any function call towards these functions can be rerouted to their callees in the enclave, and thus retaining the inter-procedural transfer correctness.

Trampoline Code In-place binary editing rewrites bytes in a binary code. To be compatible with the editing context (e.g., avoid breaking pointers), usually code snippets used for substituting the original binary content are deliberately designed to be as concise as possible.

A sample trampoline code used in this research is shown in Figure 2. Note that libraries (including the SGX interface libraries) are usually loaded at the higher addresses in a process memory space. In other words, memory address of each SGX interface function is usually larger than 4 bytes on 64-bit x86 platform. On the other hand, each time `push` instruction can only store a 4-byte value on top of the stack.

One common trick is that `mov` instruction can operate a 8-byte operand one time as long as the destination operand is `rax`. Hence, we first store the address of the trampoline callee into register rax (line 6), and then use xchg to exchange value in register with the top of the stack (where the original value of rax stores). ret instruction is then used to perform unconditional control transfers with the address on top of the stack as its destination (line 8).

Since the trampolines are located in the application code at the lower address of a process memory space, we can directly store the "return_addr" on the stack with one instruction (line 4); "return_addr" represents the address of the instruction on line 9. After finishing the execution of sgx_interface_foo function, the control flow will return back to line 9.

As previously mentioned, we place this instruction sequence at the beginning of foo by editing the leading bytes of its function body. Our deliberately selected instructions are indeed very concise (only 28 bytes), hence any non-trivial function shall provide enough space to be rewritten with such trampoline.

4.2 Assembly Function Analysis

For each target function, we perform an intra-procedural analysis to recover high-level program information. The recovered information is critical for preserving functionality correctness, supporting SGX enclave access control, as well as further instrumentation and retrofitting.

Function Prototype Recovery One attractive feature provided by SGX SDK is "access control". In particular, for a function call which has parameters of pointer types, users can annotate these pointer parameters and Intel SDK would generate additional routines for pointer legitimacy checks. A number of pointer properties (e.g., length of the pointed memory region) are checked before entering enclave functions.

While it is straightforward to annotate sensitive parameters of pointer types for programs written in C, there is no type information available in the assembly code. To benefit from the validation checking routines for pointer parameters, we first recover type information on the target assembly functions, and hence revealing the function prototype information. Function parameters of pointer type can be configured to enable the checking process.

Relocation Information Recovery We recover program relocation symbols for each target function. In particular, we identify addresses for intra-procedural control transfers, code pointers as well as global data pointers. Such concrete addresses (i.e., absolute or relative addresses) are recognized through an existing reverse engineering tool [30]. For addresses of control transfer destinations, we lift them into relocatable symbols to support arbitrary code manipulation. On the other hand, we keep global data pointers in its original concrete value format, since after in-place binary editing, global data sections would be loaded in their original locations.

Inter-procedural control flow transfers (i.e., function calls) can be performed directly or indirectly. For direct function calls, control flow destinations can be recognized from the operand of the call instruction. As for the indirect function call which takes a code pointer as its destination, we identify code pointers embedded in the instructions. All of these control flow destinations are collected to build OCALL procedures (details are given shortly).

While recent work on recovering program relocation information has reported promising results, still, this problem is in general undecidable and unrecognized pointers may still exist when analyzing real-world complex programs. Hence, we further design

```
exception_exit:
  mov    %gs:0x0,%rax
  mov    %rax,%rbx
  call   update_ocall_lastsp
  mov    0x20(%rbx),%rdx
  mov    0x98(%rdx),%rbp
  mov    0x90(%rdx),%rsp
  mov    $target_addr,%rbx
  mov    $EEXIT, %rax
  enclu
```

Figure 3: Code snippet of exception_exit procedure.

the exception handling techniques to deliver a faithful execution during runtime (§4.3).

OCALL Trampoline As aforementioned, we analyze the assembly code of the target function and collect its inter-procedural control transfer destinations. SGX specifies that code inside an enclave needs to use OCALL routines to call functions in the untrusted part. Hence, these collected callee addresses are candidates for OCALL transformations and we create one OCALL routine for each control transfer. We rewrite the destination of such control transfer instructions to point to one corresponding OCALL routine code. The execution flow will be further forwarded to the untrusted world, where library or application functions are invoked.

Our instrumentation is conceptually similar to the "replica-based" instrumentation in terms of the organization of the instrumented components. That is, function call can eventually reach to the callee even if the callee function has been relocated inside an enclave, since its trampoline code in the instrumented binary would faithfully redirect the execution flow. On the other hand, considering the relatively high performance cost of inter-enclave control transfers, we perform optimizations at this step if both the caller and callee of a function call are in the same enclave; OCALL trampoline is omitted and caller destination is rewritten with callee's new address in the enclave.

4.3 Exception Handling

In this section we propose exception handling mechanisms to solve potential runtime exceptions (e.g., pointer dereference error). When such an exception occurs, the processor performs an AEX before invoking the system software's exception handler. The AEX saves the enclave state in the enclave's State Save Area (SSA) frame. The EXITINFO in the SSA frame contains the information used to report exit reasons to software inside the enclave. According to the SGX manual [1], a segmentation fault exception is not reported inside an enclave in the current implementation of SGX. As a result, the VALID bit of EXITINFO is not set. In order to support the recovery from runtime exceptions, we removed the examination of the VALID bit in the trts_handle_exception function (file trts_veh.cpp) to enable exception handling inside an enclave in our proof-of-concept implementation.

To perform a successful transition from the enclave state to the application state, the application stack pointers RBP and RSP need to be restored. Furthermore, the target address outside the enclave needs to be retrieved from the saved region and EEXIT leaf function needs to be executed to perform a synchronous exit. According to the SGX manual [1], upon an interrupt or exception the application's RBP and RSP are saved in the GPRSGX region of the current thread's SSA frame. We register our exception handler inside the enclave which is first invoked after the exceptions. The exception handler sets the RIP to a dedicated exception_exit procedure

(Figure 3). The `exception_exit` procedure restores the application stack by calling `update_ocall_lastsp` and then executes the `EEXIT` leaf function to exit the enclave. It also retrieves the target address[1] and puts the address to `RBX` before the `EEXIT` leaf function, so that the execution branches to the target address after enclave exits.

For each `call` from the enclave to the application that raises an exception, the exception handling process involves three transitions between enclave state and application state. Extra time is also spent on the processing inside the system software. As a result, the exception handling approach takes more time than the trampoline code based approach. However we believe the runtime exceptions are rare cases, and we propose possible methods which we leave as future work to reduce the overhead. Firstly, the runtime exception frequency could be reduced if we could update the pointer dereference information dynamically during runtime. Secondly, Intel Transactional Synchronization Extensions (TSX) could be used to transfer control to the enclave address specified in the `XBEGIN` instruction before trapping to the system software once runtime exceptions occur inside the enclave. Another side effect of the current exception handling mechanism is that it allows jumping to arbitrary code outside the enclave from within the enclave.

5 IMPLEMENTATION

We extend an existing open source binary reverse engineering platform (Uroboros [30, 31]) with the SGX instrumentation functionality described in this paper. Our prototype is implemented in Scala, with over 1,700 LOC. Our extension components perform the aforementioned instrumentation steps (§4), and also employ the core functionality of Uroboros to identify program relocation symbols (e.g., code pointers, global data pointers). The proof-of-concept implementation of the exception handling mechanism adds 56 lines of C code.

We leverage `hexedit` to edit the hex representation of the input binary code [21]. Also, although (stripped) binaries are lack of function information, existing research has presented promising results in this direction [3]. Hence as aforementioned, we assume the function information is available in this research. In addition, although "type" information is absent in disassembled outputs, there has been a lot of existing research and industrial tools performing type inference towards assembly code [7]. Without reinventing the wheel, we acquire the function prototype information (regarding the number of function arguments and whether an argument is pointer type or not) from the industrial strength reverse engineering tool IDA-Pro [12].

Symbol addresses in the SGX interface libraries need to be known during binary instrumentation (§4.1), which is contradict to the randomization agreement provided by ASLR. In our prototype implementation, we disable ASLR to fix the addresses of exported symbols in the interface libraries. Nevertheless, loading time instrumentation can be used to acquire those addresses without breaking the security guarantee of ASLR. We leave it as one future work to intercept the loading process and instrument application binaries with the runtime memory addresses of invoked functions in the interface libraries.

[1]The target address is obtained by subtracting the enclave base address from the `RIP` saved in the SSA.

Table 1: Functions placed in the SGX enclave for two performance evaluations. Function `enc` and `dec` are written by us to iteratively invoke the block-level processing code, while the other three are implemented in the OpenSSL library.

	Functions
Evaluation One	AES_decrypt, AES_encrypt, AES_ecb_encrypt, enc, dec
Evaluation Two	AES_decrypt, AES_encrypt

6 PRELIMINARY RESULTS

In this section we present the evaluations of this research. Our preliminary evaluation mainly focuses on understanding the feasibility and cost of the instrumentation products. All the experiments are launched on a machine with 3.40GHz i7-6700 CPU and 16GB memory. This machine is SGX enabled, with 64MB SGX enclave reserved memory. The operating system is 64-bit Ubuntu 16.04. The SGX interface and enclave instance libraries are all compiled by Intel SGX SDK with the pre-release mode (optimization `-O2`).

With the growing need to secure cryptosystems using SGX techniques [9], our preliminary evaluation instruments sensitive procedures provided by cryptographic libraries. In this research, we adopt the AES encryption and decryption procedures for evaluation. We wrote a sample code to perform AES encryption and decryption tasks. The AES implementation is from a commonly-used cryptosystem, OpenSSL (version 0.9.7c), and the key length is set as 256. We use the AES electronic codebook (ECB) mode to process the data. This mode divides the input data into fixed-length blocks, and perform encryption (decryption) towards each block, separately. As a result, the underlying data block processing code would be invoked for multiple times, depending on the length of the input data.

6.1 Performance Penalty Evaluation

In general, two major factors would contribute to the performance penalty of the SGX protected code: 1) execution slowdown of code components inside enclaves; 2) inter-enclave control flow transfers, e.g., enclave `ECALL`. To get a comprehensive understanding of the performance cost, we launched two evaluations to study both factors, respectively.

To measure the performance cost of the first factor, we put all the encryption and decryption functions into an enclave (referred as **Evaluation One** later in the paper). Data pointers on the secret key and input data blocks are passed in through the interface function. Hence, all the block-level encryption and decryption tasks are processed within an enclave. The second row of Table 1 presents the roster of functions in the enclave.

Moreover, by putting only the block-level encryption (decryption) functions into the enclave and changing the length of the input, we are able to control the number of inter-enclave control transfers, thus revealing how the inter-enclave transfers affect the overall execution slowdown (referred as **Evaluation Two**). Note that while enclave creations can cause even higher performance penalty, in our case they are only executed for once. That means, major performance penalty would come from the repeated SGX `ECALL`s, also the inserted trampoline code (§4). The third row of Table 1 shows two functions used for this evaluation.

Figure 4 presents the performance evaluation; we increase the number of processed data blocks, and record the execution CPU time. Besides two evaluations introduced before, we also present the performance data of the original input (the "Baseline" case).

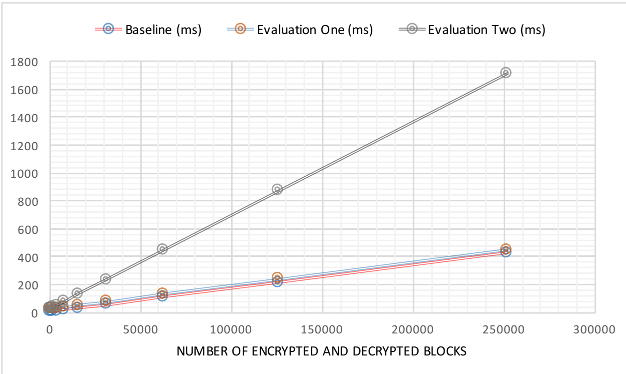


Figure 4: Performance evaluation of AES sample code. For evaluation two, processing of one data block (for encryption and decryption) leads to two SGX `ECALLs`.

For all evaluations, the overall processing time increase is roughly linear regarding the number of processed data blocks. We report that on average the instrumented binaries in **Evaluation One** are 23.7% slower than the baseline case. Note that such comparison indeed includes the processing time of enclave initialization routines, which should not change with the growing of the processed data blocks. Actually when processing over 100k data blocks (i.e., around 1.52M data), we report the normalized average overhead is 6.91%.

The second evaluation brings in notable execution slowdown. We report that on average the instrumented binary in this evaluation becomes 4.12 times slower than the baseline. We interpret the results are reasonable; frequent inter-enclave control transfers are major factors of the execution cost.

6.2 Size Increase Evaluation

Besides performance cost, SGX-based instrumentation would also bring in size increase of the instrumentation products. In this section we report the size increase of the launched experiments.

As shown in the motivating example (§4), outputs of our tool includes multiple components, i.e., the instrumented binary, the SGX interface libraries and corresponding enclave instance libraries. Also, SGX runtime environments (provided as multiple shared libraries) would also be linked into the process address space. In this evaluation, we measure the total size of the instrumented binaries and newly-created libraries by our tool, while ignoring the SGX standard runtime libraries since the later ones are not produced by our technique.

Table 2 presents the data for both evaluations. As previously discussed, our binary editing only performs in-place instrumentation, and the instrumented binaries would have identical size comparing with the input. On the other hand, both interface and enclave libraries bring in new code pieces of non-trivial size. In **Evaluation One** we put five functions into the enclave, and the size of the produced libraries grows slightly larger than the second evaluation. In sum, we interpret the size increase due to those libraries are mostly within a reasonable extent. Actually the enclave instance libraries are essential components of any SGX-based software protection (not only binary-related approaches), and we trade some memory space for flexible code re-use by compiling SGX routine code into interface libraries.

Table 2: Size increase of the instrumented outputs. Here we present the size of the original binary (second column), the instrumented binary (third column), SGX interface library (fourth column) and the enclave library (fifth column). The last column represents the total size of the instrumented outputs.

Case	Input Bin (KB)	Output Bin (KB)	Interface Libs (KB)	Enclaves (KB)	Output Total (KB)
Evaluation One	48	48	16	116	180
Evaluation Two	48	48	12	108	168

6.3 Processing Time

In this section we report the processing time of our preliminary evaluations. We report our tool takes 8.53 CPU seconds for the first evaluation while 9.12 CPU seconds for the second one. Although IDA-Pro is used to recover the function prototype information, we do not measure its processing time since it is running in a virtual machine. Typically IDA-Pro would take seconds to a few minutes to process real-world executable files, which shall be acceptable in general. Techniques proposed in our work are mostly efficient; we use Uroboros to recover the function relocation symbols, and it is reported that Uroboros takes less than one minute to analyze large size binary code (e.g., stripped GCC binary with over 3MB size). While the current processing time evaluation only delivers preliminary data, our estimation is that the proposed technique shall take less than 5 minutes for most real-world cases.

7 DISCUSSION

For code snippets running inside enclaves, performance overhead would mainly come from two aspects: 1) instruction execution overhead, 2) memory access overhead for data buffers allocated inside enclaves. Our evaluation in §6.1 has studied the first aspect, and we omit to evaluate the second factor since existing work has presented comprehensive study on this. As reported by previous research [2], random data write would lead to relatively high cost, especially when the accessed buffer size grows over the L3 cache size (usually 8MB), and further beyond the SGX EPC size.

Nevertheless, our experimental results (i.e. **Evaluation One** in §6.1) show that execution cost brought by SGX becomes negligible once the accessed data is allocated outside enclave. Hence, we interpret that while usually we shall need to trade certain efficiency for data security when designing SGX applications, sensitive code pieces can be mostly protected with relatively low cost.

8 CONCLUSION

Intel Software Guard Extensions (SGX) provides techniques to execute code and data in an isolated environment. In this work, we have presented techniques for hardening binary components (i.e., functions) through SGX enclaves. Our technique can be directly used to protect legacy binary applications with a small size of TCB. Our preliminary evaluations on hardening AES encryption and decryption procedures demonstrate the practicability of the proposed technique.

ACKNOWLEDGMENT

We thank the FEAST anonymous reviewers for their valuable feedback. This research was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894.

REFERENCES

[1] 2014. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf/. (2014). Order Number: 329298-002, October 2014.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI' 16),*. 689–703.

[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium.*

[4] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with Haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 8.

[5] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT' 17).*

[6] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. of High Performance Computing Applications* 14, 4 (????), 317–329.

[7] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4, Article 65 (May 2016), 65:1–65:35 pages.

[8] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with Déjá Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.* ACM, 7–18.

[9] Mohammad H. Mofrad, Adam Lee, and Spencer L. Gray. 2017. Leveraging Intel SGX to Create a Nondisclosure Cryptographic library. (2017). http://arxiv.org/abs/1705.04706

[10] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17).* 299–312.

[11] Laune C. Harris and Barton P. Miller. 2005. Practical Analysis of Stripped Binary Code. *SIGARCH Comput. Archit. News* 33, 5 (2005), 63–68.

[12] Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler. (2014).

[13] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 2016 USENIX Symposium on Operating Systems Design and Implementation (OSDI' 16),*. 533–549.

[14] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 17).* 145–161.

[15] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium.*

[16] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC' 17).*

[17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES' 17).*

[18] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of 2016 USENIX Security Symposium.* 619–636.

[19] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 2017 European Conference on Computer Systems.* 238–253.

[20] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P '12).* 601–615.

[21] Pixel (Pascal Rigaux). 2017. HexEdit. (2017). http://rigaux.org/hexedit.html

[22] J. H. Saltzer and M. D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (Sept 1975), 1278–1308.

[23] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *Proceedings of 2015 IEEE Symposium on Security and Privacy (S&P' 15).* IEEE, 38–54.

[24] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. 2016. S-NFV: Securing NFV states by using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization.* ACM, 45–48.

[25] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS' 17).*

[26] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC' 15).* 611–626.

[27] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications With SGX Enclaves. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium.*

[28] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference.*

[29] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the 2017 Symposium on Network and Distributed System Security (NDSS '17).*

[30] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15).*

[31] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. Uroboros: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16).*

[32] Shuai Wang, Pei Wang, and Dinghao Wu. 2017. Semantics-Aware Machine Learning for Function Recognition in Binary Code. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME '17).*

[33] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.*

[34] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of 2015 IEEE Symposium on Security and Privacy (S&P' 15).* IEEE, 640–656.