SymGX: Detecting Cross-boundary Pointer Vulnerabilities of SGX Applications via Static Symbolic Execution

Yuanpeng Wang
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
yuanpeng_wang@pku.edu.cn

Ziqi Zhang
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
ziqi_zhang@pku.edu.cn

Ningyu He
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
ningyu.he@pku.edu.cn

Zhineng Zhong
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
zhongzhineng@pku.edu.cn

Shengjian Guo
Independent Researcher
United States
guosj@vt.edu

Qinkun Bao
Independent Researcher
United States
qinkun@apache.org

Ding Li
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
ding_li@pku.edu.cn

Yao Guo
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
yaoguo@pku.edu.cn

Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
cherry@sei.pku.edu.cn

ABSTRACT

Intel Security Guard Extensions (SGX) have shown effectiveness in critical data protection. Recent symbolic execution-based techniques reveal that SGX applications are susceptible to memory corruption vulnerabilities. While existing approaches focus on conventional memory corruption in ECalls of SGX applications, they overlook an important type of SGX dedicated vulnerability: cross-boundary pointer vulnerabilities. This vulnerability is critical for SGX applications since they heavily utilize pointers to exchange data between secure enclaves and untrusted environments. Unfortunately, none of the existing symbolic execution approaches can effectively detect cross-boundary pointer vulnerabilities due to the lack of an SGX-specific analysis model that properly handles three unique features of SGX applications: Multi-entry Arbitrary-order Execution, Stateful Execution, and Context-aware Pointers. To address such problems, we propose a new analysis model named Global State Transition Graph with Context Aware Pointers (GSTG-CAP) that simulates properties-preserving execution behaviors for SGX applications and drives symbolic execution for vulnerability detection. Based on GSTG-CAP, we build a novel symbolic execution-based vulnerability detector named SymGX to detect cross-boundary pointer vulnerabilities. According to our evaluation, SymGX can find 30 0-DAY vulnerabilities in 14 open-source projects, three of which have been confirmed by developers. SymGX also outperforms two state-of-the-art tools, COIN and TeeRex, in terms of effectiveness, efficiency, and accuracy.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Vulnerability scanners**; **Trust frameworks**.

KEYWORDS

Intel SGX; symbolic execution; vulnerability detection

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623213

1 INTRODUCTION

Today, many projects have utilized Intel SGX to raise the security bars on data confidentiality [7, 13, 38, 50, 56, 60, 67, 69, 76, 81]. Intel SGX enables developers to create enclaves that protect sensitive data in an isolated environment [30]. However, recent studies have shown that ECalls, the interfaces for code from the untrusted world to communicate with code inside SGX enclaves, are also vulnerable to software vulnerabilities [11, 46, 71]. Symbolic execution has been widely adopted in software security analysis [17, 39–42]. To

ensure the security of SGX applications, researchers have developed several symbolic execution-based vulnerability detectors. For example, TeeRex [18], Guradian [5], and COIN [48] utilize symbolic execution to detect memory corruption inside enclaves. Although these approaches have been effective in discovering multiple 0-DAY vulnerabilities, they still miss critical types of vulnerabilities in SGX applications. In particular, existing approaches are ineffective in detecting *cross boundary pointer vulnerabilities*, which allow adversaries to access SGX-protected memory through a pointer deliberately created in untrusted memory.

Since SGX applications heavily use pointers to exchange data across the boundary of enclaves, *cross boundary pointer vulnerabilities* are common in practice. In fact, many incidents related to *cross boundary pointer vulnerabilities* have been reported [46, 61, 71]. However, effectively detecting *cross boundary pointer vulnerabilities* remains a challenging problem. According to our evaluation, none of the existing state-of-the-art solutions can effectively detect *cross boundary pointer vulnerabilities*.

The main reason is that existing symbolic execution techniques try to **apply the execution and analysis model for conventional programs to SGX applications**, without following the execution models of SGX applications. Compared to conventional programs, SGX applications have a new execution model with three unique properties as follows:

- Multi-entry Arbitrary-order Execution (P1): SGX applications assume that the adversary has full control of the untrusted world. Thus, unlike conventional applications, we cannot consider SGX applications as single-entry programs. Each ECall is an entry and the adversary can call multiple ECalls in any order, leading to arbitrary-order execution.
- Stateful Execution (P2): ECalls may not be independent from each other. They can share the same set of global states (global variables and heap variables) in the enclave. Thus, the execution of an ECall might show different execution behaviors before and after running another ECall if the latter modifies the global states.
- Context-aware Pointers (P3): the untrusted environment often communicates with enclaves through pointer parameters of ECalls. However, to ensure the security of enclaves, SGX should carefully check if pointers originating from the untrusted world indeed point to any addresses in enclaves. Thus, the pointers in SGX applications are context-aware.

Unfortunately, existing approaches cannot properly address the three unique properties, leading to ineffective and inefficient detection of *cross-boundary pointer vulnerabilities*. Specifically, we observe that they suffer from the following limitations.

First, the absence of multi-entry arbitrary-order execution modeling leads to ineffective detection of complex vulnerabilities. For example, one of the state-of-the-art tools, TeeRex [18], assumes that ECalls are independent, which is not accurate for SGX applications. Therefore, TeeRex cannot capture complex vulnerabilities that involve multiple ECalls. Another recent tool, COIN [48], bruteforcibly enumerates ECall sequences to simulate a multi-entry SGX application. However, this approach has low efficiency due to quick path explosion. In our evaluation, COIN failed to find vulnerabilities within the time budget for 64% of times.

Second, without tracking how ECalls modify global states, existing approaches may introduce a great many false positives. TeeRex assumes that all values from global states are free symbolic values. However, since a global state can only be modified from a preceding state by ECalls, the conservative assumption made by TeeRex can cause many false positives. COIN creates ECall sequences and retains successive global states in executing the sequenced ECalls. However, since COIN enumerates ECall sequences rather than tracking modifications to global states, COIN is incapable of leveraging internal information flow for effective sequence generation.

Third, ignoring context-aware pointers makes existing approaches fail to accurately model the semantics of the SGX sanitizers like *sgx_is_within_enclave* and *sgx_is_outside_enclave*, which are critical to ensure the security of SGX applications. TeeRex handles the semantics of the sanitizers by tracking the origins of pointers. However, TeeRex does not have the capability to reason about relations between the enclave bounds and pointers with complex computation. Thus, given a pointer that points to a segment of memory, TeeRex cannot tell whether the memory segment can cross the boundary of enclaves, leading to a substantial amount of false positives. Meanwhile, COIN does not model the semantics of the intrinsic functions at all.

To address these problems, we propose a new tool SymGX that detects *cross-boundary pointer vulnerabilities* in SGX applications. Our key insight is that we need a specific analysis model for SGX applications, instead of applying a conventional analysis model directly. However, building such a specific analysis model is challenging, because there are no effective methods to simulate the three unique properties of SGX. To this end, we propose the *GSTG-CAP* model that handles the three unique properties of SGX applications effectively. Our model is based on a Global State Transition Graph (GSTG), which is a graph that models how ECalls interact with each other by tracking changes to global states. Walking on the graph model helps to model the multi-entry arbitrary-order execution and the stateful execution properties. In addition to the GSTG, our model also includes a Symbolic Execution with Context-Aware Pointers (SECAP) that processes context-aware pointers properly. In this way, our GSTG-CAP models the behavior of an SGX application accurately, enabling effective detection of *cross-boundary pointer vulnerabilities*.

We evaluated SymGX against TeeRex [18], and COIN [48], with real-world open-source projects and known CVE vulnerabilities. Our evaluation shows that SymGX can discover 30 new vulnerabilities in 14 open-source projects from GitHub, while TeeRex and COIN detected only ten and six vulnerabilities, respectively. We have reported the discovered vulnerabilities to the project developers and have already received confirmations for three 0-DAY vulnerabilities in two open-source projects. For known vulnerabilities in the Microsoft Open Enclave SDK and Google Asylo, SymGX detected all 13 vulnerabilities within 22 minutes. In contrast, TeeRex detected only six vulnerabilities, and COIN detected only one vulnerability. Our evaluation also shows that SymGX achieved 12.28% and 19.28% higher code coverage than TeeRex and COIN, respectively, thanks to the code-coverage-guided search strategy in ECall sequence generation. Overall, our evaluation demonstrates that SymGX is superior in detecting *cross boundary pointer vulnerabilities* than state-of-the-art solutions.

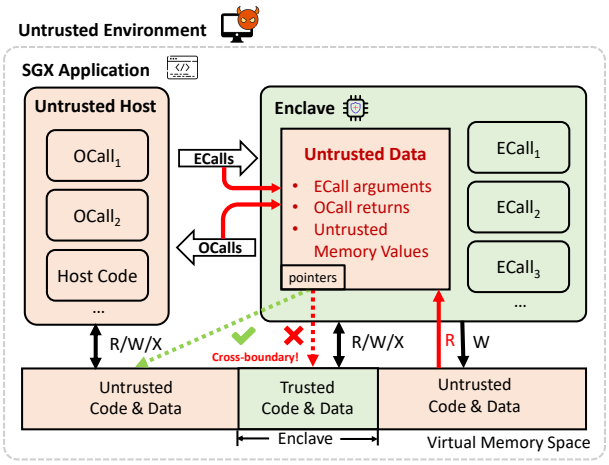

Figure 1: The structure of an SGX application.

To conclude, we make the following contributions in this paper:

- We have developed a new detector SymGX based on symbolic execution for the *cross boundary pointer vulnerabilities*, a critical security problem that occurs at the data-exchange boundary of SGX applications.
- We build a novel analysis model *GSTG-CAP* for SGX applications that accurately handles their three unique properties: Multi-entry Arbitrary-order execution, Stateful Execution, and Context-aware Pointers.
- We have evaluated SymGX on open-source projects and known CVEs in industry-class projects. SymGX discovered 30 new 0-Day vulnerabilities and three have been confirmed by developers.

SymGX is available at: https://github.com/PKU-ASAL/WASEM.

2 BACKGROUND AND MOTIVATION

In this section, we provide the background of Intel Software Guard Extensions (SGX) [30–32, 45]. Next, we use sample code snippets to illustrate insecure memory accesses in an SGX application. Moreover, we explain the limitations of existing sanitizers in SGX SDK and the limitations of other symbolic execution techniques.

2.1 Software Guard Extensions (SGX)

Intel SGX is a hardware-based solution that protects sensitive data in critical applications from strong adversaries, such as the host OS or DevOps staff [4, 55]. Figure 1 illustrates the programming model of an SGX software application, which consists of two worlds: a *trusted enclave* inside a protected memory region and an *untrusted environment* on the host machine. The untrusted environment cannot access the memory region in the enclave directly, ensuring the isolation of the enclave from the untrusted world.

The Intel SGX SDK [78] provides software interfaces for data-exchange between the enclave and the untrusted world: *Enclave Call* (*ECall*) and *Outside Call* (*OCall*). These interfaces allow the untrusted world to invoke ECalls to communicate with the enclave, and the enclave to issue OCalls to the untrusted world. Developers must declare all ECalls and OCalls in the *Enclave Description Language* (*EDL*) file. The SGX SDK then generates a *wrap function* for each ECall and OCall to verify the arguments and return values, transfer data, and switch the control flow.

```
1  enclave {
2      ...
3      // define ECALLs
4      trusted {
5          public int ecall_create_wallet(
6              [in, string]const char* master_password
7          );
8          public int ecall_show_wallet(
9              [in, string]const char* master_password,
10             [out, size=wallet_size] wallet_t* wallet,
11             size_t wallet_size
12         );
13         ...
14     };
15 };
16
```

Listing 1: A sample EDL file from sgx_wallet.

We present a sample EDL file from an application *sgx_wallet* in Listing 1. Each variable in the file begins with a pair of brackets that contain several *describers*, which have the following meanings:

- [*in*/*out*] indicates the direction of data transfer between the caller and the callee.
- [*size* = *cnt*] specifies the number of elements in a buffer.
- [*string*] denotes that the content is a string.
- [*user_check*] implies that developers are fully responsible for managing the pointer.

For example, the argument *master_password* of *ecall_create_wallet* at line 5 has the describers [*in*, *string*]. This means that the SGX SDK will verify if the corresponding string is outside the enclave, allocate a buffer inside the enclave, copy the string into the buffer before ECall execution, and assign the buffer address to ECall argument.

2.2 Cross-Boundary Pointer Vulnerabilities

Previous studies have witnessed a growing number of *cross-boundary pointer vulnerabilities* that enable adversaries to compromise SGX-protected memory by crafting a malicious pointer in the untrusted memory [46, 61, 71]. Despite their importance, effectively detecting *cross-boundary pointer vulnerabilities* with symbolic analysis techniques is still very difficult. State-of-the-art techniques, such as TeeRex [18], COIN [48], sgxfuzz [19], and Guardian [5], are more effective in detecting target conventional memory-safety vulnerabilities in enclaves. However, these recent techniques are insufficient for detecting *cross-boundary pointer vulnerabilities*.

We classify *cross-boundary pointer vulnerabilities* into two categories: the direct cross-boundary pointers (V_1) and the indirect cross-boundary pointers (V_2). The former (V_1) occurs when the attacker passes a pointer from the untrusted world that points to an address inside the enclave. The latter (V_2) occurs when the attacker indirectly manipulates a pointer within the enclave to access the SGX-protected memory.

Listing 2 illustrates an example of V_1, which enables the adversary to write to any address in the enclave. The ECall function *simple_encrypt*, defined at line 6, has one parameter *encrypted* marked by *user_check*, indicating that the SGX SDK will not verify the validity of the pointer *encrypted*. Therefore, in this case, the adversary can assign any memory address in the enclave to *encrypted*. As a result, the code at line 18 will write a char that is declared in the global variable *book* at line 24 to the address pointed by *encrypted[i]*.

```
1  //Enclave.edl
2  // msg is a safe pointer, SGX SDK will automatically
3  // check msg and move it to enclaves. encrypted is an
4  // unsafe pointer (decorated by user_check]) that the
5  // adversary controls
6  void simple_encrypt([in, size=len] char *msg,
7     [user_check] char* encrypted, unsigned len);
8
9  //Enclave.cpp
10 char book[10] = {52, 48, 55, 51, 56, 54, 50, 49, 57, 53};
11 //Attackers set encrypted to an enclave memory address.
12 void simple_encrypt(char *msg, char* encrypted,
13         unsigned len) {
14   if (!msg || !encrypted || len<=0 || len>10)
15     return;
16
17   for (unsigned i = 0; i < len; ++i) {
18     if (msg[i]<48 || msg[i]>57)
19         return;
20     // if (!sgx_is_outside_enclave(encrypted + i,
21     //  sizeof(char)))
22     //       exit();
23     // a possible fix with intrinsic functions
24     encrypted[i] = book[msg[i]-48];
25     // arbitrary write to addresses in enclaves
26   }
27 }
```

Listing 2: The sample code of unprotected pointers (V_1).

Listing 3 shows an example of V_2, which enables the adversary to access confidential data from within enclaves. In this scenario, the adversary has manipulated the variable *cnt* so that he can provide a large *cnt* that surpasses the array boundary of *meta* at line 9, and exposes the data in *secrete* through *memcpy*.

2.3 Inadequacy of Sanitizers and Analyzers

The SGX SDK provides two functions, $sgx\_is\_within\_enclave(p, size)$ and $sgx\_is\_outside\_enclave(p, size)$, to help developers verify the location of a pointer p. These functions return true if the memory region starting at p and spanning *size* bytes is entirely inside or outside the enclave, respectively. Figure 2 illustrates how these functions operate on an example. Using these functions can prevent some vulnerabilities. For example, by checking lines 20-22 in Listing 2, developers can avoid the V_1 vulnerability.

2.3.1 Inadequacy of Sanitizers. Unfortunately, the two sanitizers are prone to get misused, impeding the effective prevention of *cross-boundary pointer vulnerabilities*. Listing 4 shows an example

```
1  //Enclave.edl
2  // cnt is an offset controlled by the attacker
3  void ecall_copy_information([out, count=cnt] int* ptr,
4          unsigned cnt);
5
6  //Enclave.cpp
7  void ecall_copy_information(int *ptr, unsigned cnt) {
8      int meta[8] = {0, 1, 2, 3, 4, 5, 6, 7};
9      int secret[8] = {1, 2, 3, 4, 5, 6, 7, 8};
10     //Attacker set cnt > meta size to steal secret
11     memcpy(ptr, meta, sizeof(int) * cnt);
12 }
```

Listing 3: The sample code of unprotected array indices (V_2).

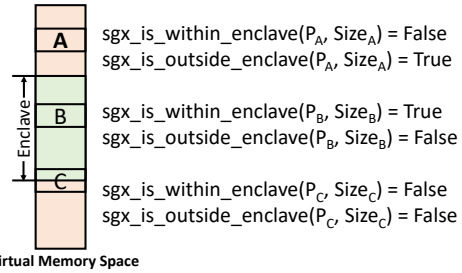

Figure 2: The usage of SGX SDK sanitizer functions, *sgx_is_within_enclave* and *sgx_is_outside_enclave*.

of how inappropriate usage of sanitizers can lead to information leakage. There are two ECalls *ecall_store_msg* and *ecall_load_msg* in Listing 4. The former allows developers to save a message to the enclave and the latter loads messages out from the enclave. In this case, since the pointer of the message p is from the untrusted world, it should not point to addresses in enclaves. Therefore, in *ecall_store_msg* and *ecall_load_msg*, the developer adds the sanitizers at line 11 and line 19 to ensure the security of p. However, the developer mistakenly writes the *!(sgx_is_outside_enclave(p, len* sizeof(int)))* at line 11 into *sgx_is_within_enclave(p, len* sizeof(int))*, allowing adversaries to bypass the check. An authentic example of this vulnerability can be discerned in *Microsoft's Confidential Consortium Framework (CCF)* [1]. Specifically, the adversary can set *len* as a large number that makes $p[len]$ exceed the memory bound of the target enclave. This setting makes p points to a memory segment like C in Figure 2 that lies across the boundary of an enclave. Since *sgx_is_within_enclave* only returns true if the checked memory segment is *fully* in enclaves, the check at line 11 will be bypassed. To this end, the adversary can specify p to an address within the enclave and copy it to the global variable *msg* at line 16. Finally, the adversary can steal the information out from the enclave by calling *ecall_load_msg*, a fully correct program.

2.3.2 Existing Analyzers for Listing 4. The vulnerability in Listing 4 is extremely challenging for existing symbolic execution-based analyzers. For example, TeeRex does not support context-aware pointers. Therefore, its analysis engine cannot accurately simulate the behaviors of both *sgx_is_within_enclave* and *sgx_is_outside_enclave*. The consequence of lacking context-aware pointers support is that,

```
1  //Enclave.edl
2  void ecall_store_msg([user_check]int *p, int len);
3  void ecall_load_msg([user_check]int *p, int len);
4  //Enclave.c
5  int *g1 = NULL, *g2 = NULL;
6  int l = 0, msg_len = 0;
7  int keys = [0,1,2,3,4];
8  char *msg = NULL;
9
10 void ecall_store_msg(char *p, int len){
11     if(len <= 0 || (sgx_is_within_enclave(p,
12        len * sizeof(int))))
13                 exit();
14     msg =  malloc(len+1);
15     msg_len = len;
16     memcpy (msg, p, len);
17 }
18 void ecall_load_msg(char *p, int len){
19     if(len <= 0 || (!sgx_is_outside_enclave(p,
20        len * sizeof(int))))
21         exit();
22     if (len < msg_len + 1)
23         exit();
24     memcpy(p, msg, msg_len);
25 }
```

Listing 4: An example to show the research challenges.

at line 24 in Listing 4, TeeRex cannot determine whether the memory segment starting at p with a size of *len* is completely outside of the enclave. Therefore, it conservatively assumes that the *memcpy* at line 24 is vulnerable since p is a pointer from the untrusted world. However, this is a false positive because the developer has already ensured that p is safe at line 20 by calling *sgx_is_outside_enclave*. COIN also suffers from the lack of context-aware pointers. For instance, it will miss all V_1 vulnerabilities (e.g., the one at line 24 in Listing 2) due to its inherent design that prevents its analyzer from distinguishing where a pointer originates from.

We then summarize the application scope of existing tools w.r.t V_1 and V_2. TeeRex cannot detect V_2 because it doesn't have information about buffer size. COIN cannot detect V_1 because it does not analyze untrusted data flow and SGX-specific memory. Guardian misses most of V_1 and all the V_2 because it only checks the violation of memory range in SDKs. Guardian does not perform taint analysis or buffer analysis.

3 OUR APPROACH

This section presents the core components of SymGX. We start by introducing the threat model and the challenges that SymGX tries to resolve. Then, we explain the underlying GSTG-CAP model that handles the execution properties of SGX applications. Next, we discuss how SymGX walks the graph and performs fine-grained taint analysis on data flows. Finally, we describe how SymGX identifies vulnerabilities in SGX applications.

3.1 Threat Model and Research Challenges

We adopt the same threat model as TeeRex [18], which assumes that enclaves protect data from adversaries. We also assume that an adversary can access arbitrary memory addresses and execute any combination of ECalls in the untrusted world. Our approach focuses on *cross-boundary pointer vulnerabilities* that are specific to SGX applications. We do not address other memory safety-related vulnerabilities that may affect SGX applications, as they are orthogonal to the interest of this work.

Based on the threat model, we aim at addressing the following research challenges in SymGX:

- **Challenge 1 (C1)**: How to model the multi-entry arbitrary-order execution behaviors of SGX applications?
- **Challenge 2 (C2)**: How to track inter-ECall information flows through global states?
- **Challenge 3 (C3)**: How to handle context-aware pointers in symbolic execution?

3.2 The GSTG-CAP Model

Overall, we address the challenges by constructing an analysis model named Global State Transition Graph with Context Aware Pointers (GSTG-CAP). For C1, we represent ECalls as edges in the Global State Transition Graph (GSTG). This design enables SymGX to generate diverse ECalls sequences by performing random walks with restarts on the graph, simulating the multi-entry and arbitrary-order ECall executions. For C2, we let each node in GSTG act as a unique SGX program global state. Thus, while randomly walking on the GSTG, the updates to the relevant nodes naturally track the inter-ECall information flow through global states. For C3, we develop a Boundary-Aware Memory Model (BAMM) inside the symbolic executor to handle context-aware pointers during ECall sequence execution and perform vulnerability detection.

Figure 3 illustrates GSTG-CAP that consists of two components, GSTG and the SECAP. The nodes in GSTG represent states of global and heap variables, and the edges are ECalls that get symbolically executed by the SECAP. We analyze the execution of an SGX application with random walks from the initial state $\eth_0$ in GSTG.

3.2.1 Global State Transition Graph. To formally define a GSTG, we assume, without loss of generality, that there are n global or heap variables in total, denoted by a vector $V =< v_0, v_1 ..., v_n >$. A global state $\eth_i$ is then a vector of symbolic expressions over V, i.e., $\eth_i =< v_0^e, v_1^e, ..v_n^e >$, where v_i^e is the symbolic value of v_i. Given a global state $\eth_i$, an ECall e_i is a function that transforms it into a new global state $\eth_i'$.

$$\mathbb{G} = (N, \eth_0, E) \tag{1}$$

We leverage the above definitions to formulate GSTG, as shown in Equation 1. In this formulation, $\eth_i \in N$ denotes a node in the graph that corresponds to a global state and $\eth_0$ is a special node that presents the initial global state, which is statically determined when an SGX application is initialized. Note that a GSTG might be an infinite graph and we strategically explore a subset of the graph. A transition $e_i \in E$ in the GSTG represents an ECall executed symbolically with context-aware pointers.

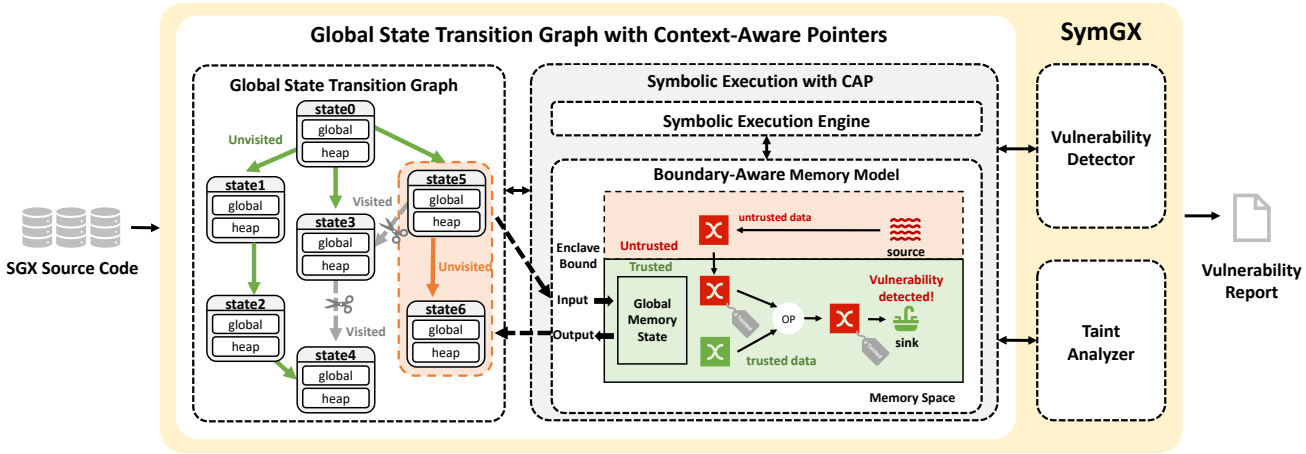

Figure 3: SymGX – The architecture, the main components, and the workflow

3.2.2 Symbolic Execution with Context-Aware Pointers. SECAP is responsible for executing ECalls and its execution forms directional edges between GSTG nodes. The new component in SECAP is the BAMM memory model, which enables SymGX to reason SGX context-aware pointers. We formalize BAMM model as a tuple $MEM = (U, P)$ that consists of a set of untrusted addresses U and a set of trusted addresses P. Each element in U or P is a range of memory addresses. Regarding the programming model of SGX, there is no overlap between the untrusted and the SGX-protected memory. Moreover, the union of U and P covers the whole supported memory address space. In other words, we partition the memory space of an SGX application under symbolic execution into two disjoint sets of U and P.

We define a set of basic memory operations, namely +, −, *load*, *store*, *malloc*, *free* over the BAMM. In particular, + and − are pointer arithmetic operators that increment or decrement a pointer. The standard semantics of the memory operations apply to memory addresses in S. For addresses in U, we assume that an adversary can arbitrarily manipulate the memory in the untrusted world. Therefore, we let addresses in U always be fresh symbolic values. This simplification for U can improve the efficiency of SymGX. For instance, a load operation on untrusted memory returns a free symbolic variable with a taint indicating it is insecure. We present the details of assigning taints to symbolic variables in Section 3.5.

One of the main challenges in modeling memory operations is to accurately model pointer arithmetic operations, + and −. These operations must not result in buffer overflows that could lead to V_2. For instance, the pointer arithmetic at line 6 in Listing 5 may cause a buffer overflow and expose sensitive information. The BAMM ensures that the buffer overflow can be detected once it happens.

To model + and − operations accurately, our BAMM keeps track of the *original base address and buffer size* of a pointer when it is allocated. This helps us detect potential buffer overflows if pointer arithmetic modifies the pointer. Specifically, for each pointer p, we associate it with two attributes, *o_base* and *o_len*, representing its original base address and size, respectively. Whenever pointer arithmetic changes p, SymGX compares the new address of the pointer to its original base address and size to detect buffer overflow. An SMT solver performs this comparison.

```
1 void ecall_get_by_index(int *ptr, int index) {
2     int meta[8] = {0, 1, 2, 3, 4, 5, 6, 7};
3     int secret[8] = {1, 2, 3, 4, 5, 6, 7, 8};
4     int *tmp = meta;
5     for(i = 0; i < index; ++i)
6         ptr[i] = meta++;
7 }
```

Listing 5: The sample code of pointer arithmetic.

3.2.3 Intrinsic Function. We design two new intrinsic functions in our Boundary-Aware Memory Model memory model. The first function *symgx_is_within_trusted*(P, *size*) checks whether a memory object P with size *size* is inside the SGX-protected memory. Likewise, the second function *symgx_is_outside_trusted*(P, *size*) checks if the memory object sits outside the enclave memory. The semantics of these functions are consistent with their counterparts in the SGX SDKs. However, we extend their functionality to let them fit BAMM and accept symbolic parameters.

Note that we borrow the name of the two intrinsic functions from the Intel SGX SDK [78]. However, they are essential for all SDKs of SGX. For Asylo [20], they are *IsInsideEnclave* and *IsOutsideEnclave*. For OpenEnclave [29], these two funcions are *oe_is_inside_enclave* and *oe_is_outside_enclave*. These functions have similar parameters and return values. Thus, they can map to *symgx_is_within_trusted* and *symgx_is_outside_trusted*.

Function *symgx_is_within_trusted*(P, *Size*) can return one of three possible values: *True*, *False*, or *Unknown*. Returning *True* means that the memory segment $[P, P + size]$ is entirely contained within an enclave. Return value *False* means that the memory segment $[P, P + size]$ is entirely outside of any enclave. The value *Unknown* means that *symgx_is_within_trusted* cannot determine whether the memory segment $[P, P + size]$ is fully inside or outside of an enclave, implying both cases are possible. Function *symgx_is_outside_trusted* has a similar usage except that it checks whether the memory segment $[P, P + size]$ is completely outside of any enclave.

We use SMT solver as the backend reasoner for both functions. Given a symbolic variable P, a symbolic variable $Size$, and an enclave ranges from b_i^p to e_i^p, we first compute two clauses for each predicate: $C_1 : (P \geq b_i^p) \wedge (P + size \leq e_i^p)$ and $C_2 : (P < b_i^p) \vee (P + size > e_i^p)$ for *symgx_is_within_trusted*; and $C_3 : (P + size \leq b_i^p) \vee (P \geq e_i^p)$ and $C_4 : (P + size > b_i^p) \wedge (P < e_i^p)$ for *symgx_is_outside_trusted*. Then we check the satisfiability of these clauses as follows: (1) if C_1 is SAT and C_2 is UNSAT then *symgx_is_within_trusted* returns *True*; it returns *False* if C_1 is UNSAT and C_2 is SAT; and it returns *Unknown* if both C_1 and C_2 are SAT. (2) Similarly, *symgx_is_outside_trusted* returns *True* if C_3 is SAT and C_4 is UNSAT; it returns *False* if C_3 is UNSAT or Unknown and C_4 is SAT; and it returns *Unknown* if both C_3 and C_4 are SAT[1].

3.3 The Architecture of SymGX

We developed a new symbolic executor based on the GSTG-CAP model to detect cross-boundary pointer vulnerabilities, as shown in Figure 3. SymGX takes in the source code of an SGX app and outputs a vulnerability report using the GSTG-CAP analysis model. It builds a GSTG to model ECall interactions and simulates execution as random walks with restarts on the GSTG. Our SymGX framework uses SECAP to analyze transitions between nodes in GSTG. SECAP builds on classic symbolic executors but with the new memory model of BAMM for SMT-based reasoning of context-aware pointers. It interprets instructions with symbolic values and maintains program symbolic states. Forked states may have constraints added, and executed instructions are recorded for analysis. Program paths are viewed as sets of visited basic block pairs.

Based on the GSTG-CAP model, SymGX incorporates a coverage-guided random-walk strategy on GSTG to simulate the generation of sequences of ECalls to an SGX application. Moreover, SymGX owns a fine-grained taint dataflow analyzer that can differentiate between the two types of *cross-boundary pointer vulnerabilities* and a specialized detector for effective vulnerability identification. Note that we will focus on the new features of SymGX and omit the normal behavior of SymGX as a symbolic executor.

3.4 Code Coverage-guided Graph Random Walk with Restart

SymGX starts its analysis by doing random walks from the initial state $\eth_0$ in GSTG-CAP. A purely random approach is inefficient for two reasons: (1) it may explore limited paths in GSTG and (2) GSTG may have infinitely many nodes. To address the first issue, we assign a restart probability p^r to the random walk. Therefore, in each step, SymGX has the potential to return to the initial state. For the second issue, we design a code coverage-guided algorithm to improve the efficiency of SymGX.

Algorithm 1 presents the code coverage-guided random walk algorithm. SymGX starts by creating a priority queue *prio_queue* that contains the initial state $\eth_0$ and a priority function $\mathcal{A}_r$ (lines 1-2). Then SymGX explores the graph in a while loop (lines 4-16). At each iteration of the loop, SymGX constructs the path that reaches the current node. This path consists of a sequence of edges that

[1]the SMT solver may not finish solving the constraints within time limits. In this case, we consider the constraints UNSAT

Algorithm 1: Code coverage Guided Random Walk with Restart

```
   Input  : 𝔾_state
   Output : ECall Sequences
1  ð_0.path = {ð_0}
2  prio_queue = init(A_r, ð_0)
3  visited = ∅
4  while prio_queue ≠ ∅ do
5      current=prio_queue.pop()
6      visited = visited ∪ {current}
7      yield current.path
8      for each ECall e ∈ E do
9          next=e(current)
10         next.path=current.path ∪ {next}
11         if not A_p(next) then
12             prio_queue.push(next)
13         end
14     end
15     Restart with probability p^r
16 end
17 return 𝔾_state
```

corresponds to a valid ECall sequence (line 7). To enable yielding paths, we store a field *path* for each node. $n.path$ records all the nodes on the path from $\eth_0$ to n. We update the field of *path* for each node at line 1 and line 10. When SymGX examines the successors of the current node, it eliminates redundant successors with a pruning function $\mathcal{A}_p$(line 11).

We now describe the priority function $\mathcal{A}_r$ and the pruning function $\mathcal{A}_p$ in Algorithm 1. To do so, we introduce three essential concepts. First, for a node n, we define $\mathbb{B}_n^c$ as the set of code basic blocks that are covered by any ECall on the edges of $n.path$. Next, we define $\mathbb{P}_e = \bigcup_{n_j \in visited} \mathbb{B}_{n_j}^c$ as the set of code basic blocks that are covered by any node in the *visited* set (see Algorithm 1) during the current graph traversal iteration. Finally, we define $\mathbb{P}_e/\mathbb{B}_{n_j}^c$ as the set of code basic blocks that are newly covered by symbolically executing the ECalls on $n_j.path$.

$\mathcal{A}_r$ is a scoring function that takes a node n_j from the *global transition graph* and assigns it a value for the priority queue. The idea is to prioritize the ECalls that are likely to explore new code basic blocks and discourage the ones that are too long. SymGX calculates the score for $\mathcal{A}_r$ using Equation 2. In particular, $|\mathbb{P}_e/\mathbb{B}_{n_j}^c|$ is the number of newly discovered code basic blocks. $len(n_j.path)$ calculates the length of $n_j.path$ and we penalize longer ECall sequences to avoid inefficient path exploration.

$$score = |\mathbb{P}_e/\mathbb{B}_{n_j}^c| - len(n_j.path). \quad (2)$$

The function $\mathcal{A}_p$ takes a node n_j as input and returns true if and only if $\mathbb{P}_e/\mathbb{B}_{n_j}^c = \varnothing$. This function is used to eliminate the ECall paths that do not cover any new code basic blocks.

3.5 Fine-grained Taint Dataflow Analyzer

This component distinguishes the vulnerability V_1 from V_2. We make this distinction based on the observation that V_1 poses more

Table 1: Description of Attributes

Attributes	Description
`taint`	whether the variable may be controlled by adversary
`pointer`	whether the variable is a memory address
`taint_base`	whether the base pointer is controlled by adversary

Table 2: Initial attributes for inner values

Element	Taint	Pointer	Base_taint
Constant Number	F	F	Null
Local Pointer	F	T	F
Global Pointer	F	T	F
ECallArguments	T	T/F	T/Null
OCallReturns	T	T/F	T/Null
Null Pointers	F	T	F
Untrusted Memory Values	T	T/F	T/Null
Lib function Returns	F	T/F	F/Null

risks than V_2. V_1 enables the attacker to manipulate a base pointer, resulting in arbitrary read/write in enclaves. In contrast, V_2 only permits attacker-controlled indexes. As long as there is no buffer overflow, the damage of V_2 is limited. By differentiating V_1 from V_2, we can reduce false positives by reasoning about whether the array index may cause overflow for V_2.

To conduct a fine-grained analysis, we assign a set of attributes to each variable that capture different properties. In particular, we devise three attributes: `taint`, `pointer` and `base_taint`, as shown in Table 1. The `taint` attribute indicates whether the variable can be influenced by the adversary (either by manipulating the base pointer or the offset). The `pointer` attribute indicates whether the variable is a memory address. The `base_taint` attribute indicates whether the base pointer is under the adversary's control. If `taint` and `base_taint` are both `True`, the attacker can control the base pointer (V_1). If `taint` is `True` and `base_taint` is `False`, the attacker can control the address offset (V_2).

We now explain how each attribute is assigned a value. There are eight kinds of immediate sources of values in SymGX, as shown in Table 2. Lib functions are the standard library functions in SGX SDK that we emulate. The `taint` attribute is set to be `True` only for variables that come from one of the three taint sources, namely, ECall arguments, OCall return values, and untrusted memory values. Otherwise, `taint` is set to be `False`. The `pointer` attribute is set to be `True` for the following cases: local pointers, global pointers, Null pointers, lib function returns of pointer type, ECall arguments of pointer type, OCall returns of pointer type, and untrusted memory addresses of pointer type. The type information can be determined at compile time. We do not consider fixed-address pointers because we assume that SGX programs follow the pointer rules of C standard. The `pointer` attribute is set to be `False` for other variables. For `base` and `base_taint` attributes, they are set to `NULL` when `pointer` is `False`, because the variable is not a memory address. If `pointer` is `True`, the `base_taint` attribute is assigned the same value as the `taint` attribute.

Algorithm 2 presents the taint propagation algorithm. We consider the general form of a binary operation that generates a new variable (x_{res}) from two operands (x_1 and x_2). The $x_{res}.taint$ attribute is set to `True` if either $x_1.taint$ or x_2.`taint` is `True`. When both operands are address variables, we assume that the operation can only be a comparison operation (== or !=) and $x_{res}.pointer$ is set to `False`. If one of the operands is a pointer and the other is not, we assume that the operation can only be addition or subtraction. We set $x_{res}.pointer$ to `True` and assign the attributes of the pointer operand to $x_{res}.base\_taint$. This way, we can ensure that, for each address variable, `taint` is `True` whenever its data-flow is influenced by attackers. The `base` and `base_taint` attributes will not change when the offset is modified. These two attributes will always record the buffer base. Note that the propagation algorithm can be easily extended to other forms of operations since they can be converted to binary forms accordingly.

Algorithm 2: Taint Propagation

```
   input  : x1, x2, op
   output : x_res
1  x_res = new_variable()
2  x_res.value = op(x1.value, x2.value)
3  taint = x1.taint ∨ x2.taint
4  if x1.pointer ∨ x2.pointer then
5      if x1.pointer ∧ x2.pointer then
6          assert(op is '==' or '!=')
7          pointer = False
8          base_taint = Null
9      else
10         assert(operation is '+' or '-')
11         pointer = True
12         if x1.pointer then
13             base_taint = x1.base_taint
14         else
15             base_taint = x2.base_taint
16 else
17     pointer = False
18     base_taint = Null
19 x_res.attributes = <taint, pointer, base_taint>
20 return x_res
```

3.6 Vulnerability Detection

As discussed in Sec. 2.2, memory instructions with unsafe address arguments caused vulnerabilities V_1 and V_2. So for each memory operation in the SGX program, we define two checking rules to detect V_1 and V_2, respectively. Algorithm 3 presents the detailed algorithm we use to identify vulnerabilities.

SymGX identifies V_1 by applying two conditions in Algorithm 3: (1) the pointer originates from the untrusted world (lines 1-2) and (2) it points to an address within enclaves (line 3). For (1), we check the `taint` and `base_taint` attributes of the target address. If both attributes are `True`, it indicates that the pointer is under the attacker's control. For (2), we use the methods discussed in Section 3.2.3 to check whether the address belongs to the enclave. If both conditions are satisfied, SymGX treats it as a V_1 instance.

Algorithm 3: Vulnerability Detection

```
Input  : addr, length
Output : vulnerability_reports
1 if addr.taint then
2     if addr.base_taint then
3         if sgx_outside_enclave(addr, length) == 0 then
4             return V1
5     else
6         if check_inside_buffer(addr, length) == 0 then
7             return V2
8 return Null
```

To detect vulnerability V_2, SymGX applies two conditions: (1) the offset comes from the untrusted world (line 1) and (2) the address goes beyond the buffer size (line 6). For (1), if `taint` is `True` and `base_taint` is `False`, it indicates that the attacker can manipulate the offset. For (2), we use an SMT solver to find possible solutions that show if the address can exceed the buffer bound. We obtain the base address from the `base` attribute and the buffer size from the compilation information and memory. The compilation information includes the size of local and global variables, and our memory model records the size of dynamic variables that are created by `malloc` functions. We formulate a set of checking expressions to represent whether the address can cross the buffer bound and use an SMT solver to solve them. If a solution exists, it means the address may go beyond the buffer, and we report a V_2 vulnerability.

4 EVALUATION

To evaluate SymGX, we create a comprehensive list of following research questions.

RQ 1: How effective is SymGX's vulnerability detection ability?
RQ 2: How precise is SymGX's vulnerability report?
RQ 3: Can SymGX achieve higher code coverage than baselines?
RQ 4: How does GSTG-CAP perform without code coverage-guided random walks with restarts?
RQ 5: What is the runtime memory consumption of SymGX?

4.1 Benchmark

We constructed a benchmark set of 27 applications that exhibit cross-boundary pointer vulnerabilities. The set includes 14 public GitHub applications and 13 real-world CVEs. We collected 22 open-source SGX applications from GitHub and successfully compiled 14 of them. The remaining 8 failed to compile. The real-world CVEs were found in popular confidential computing frameworks such as Microsoft Open Enclave SDK [29] and Google Asylo [20], which are widely used for SGX programming. The CVEs demonstrate typical memory vulnerabilities in SGX memory.

We present the details of the open-source GitHub applications in Table 3 and the details of the CVEs in Table 5. For each GitHub application, we report the number of basic blocks (`#BasicBlocks`), OCall (`#OCall`), and ECall (`#ECall`), following previous studies [18, 48]. These metrics indicate the complexity of SGX applications. For each real-world CVE, we provide the CVE number and the vulnerability type in Table 5.

Table 3: The information of GitHub projects we use in evaluation

Projects	Version	#.Basicblocks	#.ECall	#.OCall
sgx-wallet [6]	8d15df1	312	4	7
SGXCryptoFile [63]	92f3cd6	4130	38	12
sgx-dnet [62]	0fe09cc	4910	3	4
verifiable-election [36]	5f0f9f1	1145	7	2
sgx-log [47]	7b5530e	1080	4	6
sgx-kmeans [37]	8ab6035	185	22	8
sgx-reencrypt [51]	6f06591	904	4	4
CryptoEnclave [57]	5c60615	2983	17	11
sgx-pwenclave [33]	b81eace	1032	3	3
sgx-deep-learning [52]	2a6c3b7	3192	9	2
sgx-biniax2 [9]	35aaa1e	291	3	4
sgx-rsa [82]	ed099d4	731	7	5
sgx_protect_file [59]	5f2e64e	609	5	4
SGXSSE [3]	0520695	489	6	3

4.2 Implementation and Experiment Setup

We used Python and SeeWasm [44] to create SymGX with 4.5k lines of code. Our method sets p^r to 0.2 for optimal efficiency, but other values were not significantly different in effectiveness.

We conducted our experiments on a server running Ubuntu 20.04 with 8 Intel® Xeon® Gold 5218R CPUs @ 2.10GHz and 200 GB of RAM. It is worth noting that our implementation is purely based on software simulation of SGX functions. We need no hardware support of SGX to run SymGX. We used Wllvm [70] to compile all the benchmarks into dynamic link libraries and generate LLVM bitcode files. Then, we used LLVM to compile the benchmarks from LLVM bitcodes to WASM bitcode.

We compared our tool with two state-of-the-art tools, TeeRex [18] and COIN [48], as baselines. We obtained the COIN code from its official implementation [48], and we only focus on memory vulnerabilities that are relevant to *cross-boundary vulnerabilities*. For TeeRex, since its code is not available, we implemented its framework based on the design and algorithm in its paper [18]. The original TeeRex was based on angr [68] and applied to executed programs, while our framework works on bytecodes. Therefore, there exist differences between our implementation and the original one. To ensure a fair comparison between TeeRex and our tool, we kept all factors the same except for the core algorithm, including the instruction emulators and the state scheduling algorithm. Following the experimental settings in previous works, we set the time limit to 12 hours to avoid any bias due to different tools.

Note that we decided not to evaluate two other recent works sgxfuzz [19] and Guardian [5] after a preliminary study. sgxfuzz [19] is a fuzzing tool that fundamentally differs from SymGX while Guardian [5] adopts a technique fundamentally similar to TeeRex but has inferior performance.

4.3 Effectiveness

To answer RQ1, we compare the vulnerability detection performance of SymGX with baseline tools TeeRex and COIN. We conduct two tasks: (1) 0-DAY vulnerability hunting and (2) known vulnerability validation. For 0-DAY vulnerability hunting, we apply SymGX to 14 open-source GitHub projects and search for 0-DAY vulnerabilities. We manually verify the reported vulnerabilities by

Table 4: The effectiveness of SymGX and baselines in finding cross-boundary vulnerabilities. #Alerts is the number of alerts reported by different tools. #Vul is the number of vulnerabilities we confirmed in the reports. SymGX$^{\#}$ is SymGX that only uses the BAMM without $\mathcal{A}_r$ and $\mathcal{A}_p$.

Projects	TeeRex				COIN				SymGX$^{\#}$				SymGX			
	#Alerts		#Vul		#Alerts		#Vul		#Alerts		#Vul		#Alerts		#Vul	
	V_1	V_2	V_1	V_2	V_1	V_2	V_1	V_2	V_1	V_2	V_1	V_2	V_1	V_2	V_1	V_2
sgx-wallet [6]	273	0	1	0	0	0	0	0	4	5	0	1	6	9	0	1
SGXCryptoFile [63]	744	0	0	0	0	3	0	0	3	9	0	0	5	13	0	0
sgx-dnet [62]	321	0	1	0	0	0	0	0	2	4	1	1	4	8	1	1
verifiable-election [36]	410	0	0	0	0	5	0	2	4	8	1	2	7	11	1	4
sgx-log [47]	282	0	0	0	0	0	0	0	6	2	0	0	11	4	0	0
sgx-kmeans [37]	465	0	0	0	0	0	0	0	4	15	0	1	5	8	0	2
sgx-reencrypt [51]	92	0	0	0	0	0	0	0	0	5	0	0	0	9	0	0
CryptoEnclave [57]	876	0	1	0	0	0	0	0	10	3	1	0	16	5	1	0
SGX-pwenclave [33]	401	0	1	0	0	3	0	0	16	8	1	0	18	9	1	0
intel-sgx-deep-learning [52]	529	0	0	0	0	0	0	0	5	7	2	1	7	12	3	1
sgx-biniax [9]	691	0	2	0	0	0	0	0	4	4	0	0	9	6	0	2
sgx_rsa [82]	732	0	0	0	0	9	0	3	2	11	2	1	5	15	2	3
sgx_protect_file [59]	822	0	4	0	0	0	0	0	13	4	3	0	17	9	3	1
SGXSSE [3]	752	0	0	0	0	5	0	1	5	19	1	1	6	21	1	2
total	7390	0	10	0	0	25	0	6	78	104	12	8	116	139	13	17

following the methodology of previous literature [18]. For each vulnerability, we try to craft an exploit. For a V_1 vulnerability, we aim to create a pointer from the untrusted world that points to an enclave address. For V_2, we create an array offset to cause a buffer overflow inside enclaves. We classify reports that can be successfully exploited as true positives. The exploit construction procedure is consistent with previous work [18]. We report the number of true 0-DAY vulnerabilities and false positives for the 14 projects. For known vulnerability validation, we measure the time taken by SymGX to detect the 13 real-world CVEs in Microsoft Open Enclave SDK [29] and Google Asylo [20]. Overall, these vulnerabilities involve 1 - 3 Ecalls and 2 - 15 parameters. We list the details in Table 6 in Appendix A.

We present our findings in Table 4. The column labeled "#Vul" displays the number of vulnerabilities detected in GitHub applications (we will discuss the other parts of Table 4 in Section 4.4). In total, our tool SymGX identified 13 V_1 vulnerabilities and 17 V_2 vulnerabilities. Three of these vulnerabilities have been verified and confirmed by the developers. We compare our tool with two baselines: TeeRex and COIN. TeeRex reported 10 V_1 vulnerabilities, while COIN detected 6 V_2 vulnerabilities and none of the V_1 vulnerabilities. Interestingly, COIN did not find any issues in 11 applications. In summary, our tool discovered more vulnerabilities (13+17=30) than the combined results of the baselines (10 for TeeRex and 6 for COIN), demonstrating its effectiveness. We present the types, depths of ECalls, and the numbers of ECall parameters of each vulnerability in Appendix A.

We present the results of real-world CVEs in Table 5. The first six rows (CVE-2020-8904 to CVE-2020-8944) correspond to V_1 vulnerability, while the last seven rows show the V_2 vulnerability (CVE-2020-15224). We report the time taken by each tool to detect the vulnerability or *fail* if it exceeded the time limit. Our tool SymGX can identify all V_1 and V_2 vulnerabilities within 30 minutes and the average detection time is about 12.9 minutes.

SymGX outperforms both baselines in detecting V_1 and V_2 vulnerabilities. COIN fails to identify any V_1 vulnerabilities and only finds one V_2 vulnerability (CVE-2020-15224) in 62 minutes, which is about seven times slower than our tool's 9 minutes. TeeRex, on the other hand, can only detect V_1 vulnerabilities and takes much longer than our tool. On average, TeeRex takes 31.17 minutes, which is more than 2.41 times the 12.92 minutes taken by our tool.

Response from Developers: We reported the 30 vulnerabilities that SymGX detected to the developers of the affected projects. By the time we submitted this paper, we had received feedback from two projects about three vulnerabilities. The developers confirmed that these were **0-DAY** vulnerabilities. The first two vulnerabilities, a V_1 and a V_2 vulnerability, respectively, affected SGX-dnet [62]. The third vulnerability was a V_2 vulnerability in sgxwallet [6]. We are collaborating with the developers to create patches for these three vulnerabilities.

Answer to RQ 1: Within a limited time, SymGX, can detect more vulnerabilities than the state-of-the-art baselines.

4.4 Precision

Precision is an important metric for evaluating bug detectors. A higher precision, or a lower false alarm rate, can help developers save a lot of time and effort in investigating vulnerabilities. Table 4 shows the bugs reported in GitHub applications in the "#Alerts" columns. Precision can be computed as $\#Vul/\#Alerts$.

As shown in Table 4, TeeRex has a very low precision of only 0.14% (10/7390), which leads to a huge number of false positives. For instance, in CryptoEnclave, out of 876 reports, only one contains a real vulnerability. On the other hand, COIN has a much higher precision of 24.00% (6/25) because it only raises alerts when it is confident. However, this cautious approach also fails to catch many

[1]There are multiple vulnerabilities sharing the same CVE-number according to the commit of developers [21] and we detect the vulnerabilities separately.

Table 5: The time consumed to detect real vulnerabilities for SymGX and baselines. Fail means the tool cannot find the vulnerability within 12 hours. The column CVE contains the link to the CVE reports.

Project	Version	CVE	Type	SymGX	TeeRex	COIN
Asylo	0.5.3	[24]	V_1	6min	23min	Fail
		[23]	V_1	11min	17min	Fail
	0.6.0	[28]	V_1	7min	8min	Fail
		[27]	V_1	2min	21 min	Fail
		[25]	V_1	9min	7min	Fail
		[26]	V_1	12min	12min	Fail
OpenEclave[1]	0.10.0	[22]	V_2	22min	Fail	Fail
		[22]	V_2	19min	Fail	Fail
		[22]	V_2	17min	Fail	Fail
		[22]	V_2	12min	Fail	Fail
		[22]	V_2	9min	Fail	62min
		[22]	V_2	24min	Fail	Fail
		[22]	V_2	18min	Fail	Fail

true vulnerabilities. As we discussed in **RQ 1**, COIN can detect only 6 V_2 vulnerabilities and none of the V_1 vulnerabilities, limiting its usefulness in real-world applications.

SymGX achieves a precision of 11.76% (30/255) in detecting real vulnerabilities. This is a significant improvement over TeeRex, which has a precision of only 0.2%. COIN has a higher precision than SymGX, but it also misses 5× more real vulnerabilities (30 vs. 6). Therefore, SymGX balances precision and recall better than the existing tools.

4.4.1 False Positives: After analyzing false alarms, we have identified three main sources. Firstly, SMT solvers may produce inconsistent results in certain cases, which is unavoidable in practice. In situations like those detailed in Section 3.2.3, if the SMT solver fails to provide an accurate result, SymGX may explore branches that are not reachable, ultimately leading to false alarms. Secondly, SymGX may return an unconstrained symbol whenever it encounters functions that are not emulated. In reality, the return value may be constrained, causing false alarms. Lastly, the taint and pointer analysis is not 100% accurate and may introduce false alarms.

We conducted a quantitative analysis to determine the sources of the false alarms. Out of the 225 false alarms, 102 were due to the SMT solver's inaccurate results, 71 were due to undefined functions, and 52 were due to errors in taint and pointer analysis. Based on this analysis, reducing false alarms can be achieved by emulating more functions and enhancing the accuracy of static analysis.

4.4.2 Effectiveness of the Two Conditions of V_1: In Algorithm 3, there are two conditions used to detect V_1. Line 2 detects if the adversary provides the address base pointer, and Line 3 detects if the pointer points to an address inside the enclave. The second condition is necessary because our threat model assumes that all data outside of enclaves are dangerous. To quantitatively study the effectiveness of the second condition of V_1, we conducted the experiment from Section 4.4 again and found 893 false alerts. In comparison, with both conditions enabled, SymGX had 225 false positives. This result shows that the second condition of V_1 effectively reduces false positives.

Answer to RQ 2: SymGX achieves a precision of 11.8%, which is nearly 59× better than the state-of-the-art TeeRex. Moreover, SymGX identifies 5× more vulnerabilities than COIN, even though it has a lower precision.

4.5 Coverage

We conduct a comparison of the code coverage obtained by SymGX and two baselines. Due to the space constraint, the complete results are provided in Appendix A.

SymGX achieves an average coverage of 48.21%, which exceeds TeeRex by 12.28% and COIN by 19.28%. SymGX consistently surpasses the other two baselines for all applications (details in Appendix A). This superior performance results from our ECall sequence generation algorithm and optimizations, namely, the pruning and ranking approaches. Unlike TeeRex, which overlooks information flows across ECalls, SymGX can reduce execution time by eliminating unreachable paths. In contrast to COIN, which performs brute-force and random emulations of ECall sequences, the optimization developed in SymGX can selectively generate more high-quality ECall sequences.

Answer to RQ 3: One of the factors that makes SymGX more effective is its higher code coverage. Compared to TeeRex and COIN, SymGX achieves 12% and 20% higher code coverage on average, respectively.

4.6 Case Study

In this section, we explain how SymGX can detect realistic cross-boundary pointer vulnerabilities with a case study. The vulnerability is from sgx-biniax2 [9]. We show the main codes in Listing 6 [2].

In the code snippet, there are two ECalls: `add_to_store` (Line 15) and `get_from_store` (Line 23). The function `add_to_store` takes an array (`in_bytes`) and its length (`in_len`) as input and stores the input to the global variable store (line 15) in the enclave. The content of `in_bytes` is stored in `store->bytes` and the length `in_len` is stored in `store->lengths`. `get_from_store` receives an index, looks up `store->bytes`, and moves `store->bytes[index]` back to the untrusted world.

There's a V_2 vulnerability in the code where `get_from_store` doesn't verify `out_len`. An attacker can use `add_to_store` to insert an array, and then calling `get_from_store` with a `out_len` smaller than `in_len` to cause a buffer overflow. Note that on line 5, the developer added the annotator `[out_bytes, size=out_len]`. This indicates that the pointer `out_bytes` is shared by both the enclave and the untrusted world. To make this possible, the SGX SDK will allocate a temporary buffer inside the enclave with the size of `out_len` as the target of `memcpy`. It will then copy the content from the temporary buffer to an array outside the enclave. Unfortunately, an adversary can cause a buffer overflow at line 33 by carefully crafting `index`, `in_len`, `in_bytes`, and `out_len`, allowing arbitrary modifications to memory inside the enclave.

SymGX detects the vulnerability as follows: SymGX explores the GSTG and finds a Ecall chain of `add_to_store` → `get_from_store`. During the exploration, SymGX symbolically execute `add_to_store`,

[2]We simplified the code for clarity.

```
//Enclave.edl
void add_to_store([in,size=in_len]const void *in_bytes,
 size_t in_len);
void get_from_store([out,size=out_len]void *out_bytes,
 size_t out_len, size_t index);
...
#define MAX_ARRAY_NUM 100
typedef struct Store {
        int array_num;
        int bytes[MAX_ARRAY_NUM];
        int lengths[MAX_ARRAY_NUM];
} Store;
//Enclave.c
sgx_CryptStore_Store *store;
void add_to_store(const void *in_bytes, size_t in_len){
 ... // checks on in_len and in_bytes
 store->bytes[store->array_num] = malloc(in_len);
 memcpy(store->bytes[store->array_num],in_bytes,in_len);
 store->lengths[store->array_num] = in_len;
 store->array_num++;
 ...
}
void get_from_store(void *out_bytes, size_t out_len,
  size_t index){
 ... //checks on out_bytes and index but not on out_len
 int size = store->lengths[index];
 memcpy(out_bytes, store->bytes[index], size);
 ...
}
```

Listing 6: A V_2 vulnerability caused by a global variable.

which adds a `in_bytes` and `in_len` as symbolic variables to the arrays in `store`. In this step, SymGX marks the `in_bytes` and `in_len` as tainted because they are from the untrusted world. Here, the `index` is concretized. Lastly, SymGX symbolically executes `get_from_store`. In this step, SymGX enumerates possible values of `index` and retrieves the corresponding symbolic values from the array. Then, it detects whether the `size` variable at line 32 is tainted. If so, SymGX calls the SMT solver to check whether the `size` can be larger than `out_len`, which leads to V_2 in our case.

4.7 Ablation Study

In this section, we assess the effectiveness of GSTG-CAP and two optimization functions, $\mathcal{A}_r$ and $\mathcal{A}_p$. We compare the performance of our tool with and without these optimization functions.

We show the results in Table 4 and Figure 5. Due to the space limit, we put the complete result of the coverage data in Figure 5 in the Appendix. In our results, SymGX # represents the performance of SymGX that only uses GSTG-CAP without $\mathcal{A}_r$ and $\mathcal{A}_p$. As shown in Table 4, SymGX # still finds 20 vulnerabilities and outperform the baselines. This result demonstrates the effectiveness of our memory model. We also collect thr code coverage of SymGX without $\mathcal{A}_r$ and $\mathcal{A}_p$ and show the results in Figure 5 represented by SymGX #. Without $\mathcal{A}_r$ nor $\mathcal{A}_p$, the average coverage is 39.61%. $\mathcal{A}_r$ and $\mathcal{A}_p$ can increase the coverage by 8.60%.

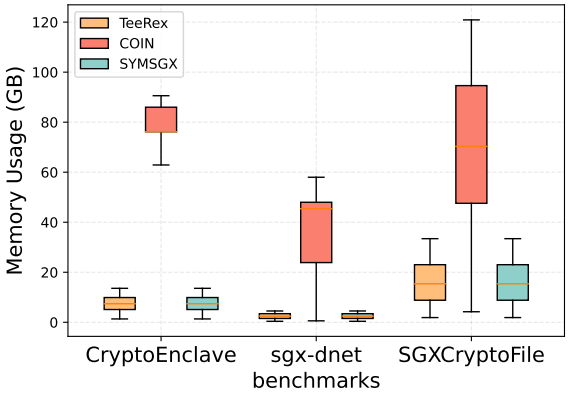

Figure 4: Memory usage for SymGX and baselines

As discussed in Section 4.3, SymGX discovered 30 vulnerabilities by combining $\mathcal{A}_r$ and $\mathcal{A}_p$. This is because $\mathcal{A}_p$ can save resources by skipping the set of ECall sequences that revisit the same code blocks, while $\mathcal{A}_r$ can focus on more important ECall sequences that explore more new code blocks. Hence, we infer that the optimization functions $\mathcal{A}_r$ and $\mathcal{A}_p$ can enhance the effectiveness of SymGX.

Answer to RQ 4: SymGX achieves superior performance over the baselines only when using the GSTG-CAP. The effectiveness of SymGX can be further enhanced by applying $\mathcal{A}_r$ and $\mathcal{A}_p$.

4.8 Memory Consumption

We measured the memory usage of our tool and its baselines. We collected the memory usage data every minute during the execution process and displayed the box-plot of the memory usage samples in Figure 6 in Appendix A. The average usage for TeeRex, COIN, and our tool was 8.43GB, 50.71GB, and 8.94GB, respectively. We noticed that the memory usage of COIN was 5.67 times higher than SymGX. This is because COIN generates all possible ECall sequences that its memory usage grows exponentially with the number of ECalls. The memory usage of TeeRex was comparable to that of our tool.

Answer to RQ 5: SymGX has a much lower memory consumption than COIN.

5 RELATED WORK

SGX has been widely used in many applications [10, 15, 66, 73–75]. Recent works have intensively studied side-channel vulnerabilities in SGX [2, 8, 12, 14, 16, 34, 43, 49, 53, 58, 72, 77, 79], while memory attacks have also received increasing attention [11, 46, 65, 71]. ASLR-based memory protection methods also appeared for SGX [54, 64, 80]. However, they cannot fully mitigate cross-boundary pointer vulnerabilities and are orthogonal to SymGX. The closest work to SymGX is vulnerability detectors like COIN [48], TeeRex [18] and other similar approaches [19, 35]. Nevertheless, none of these approaches can handle cross-boundary pointer vulnerabilities or capture the three unique properties of SGX applications.

6 DISCUSSION

Handling OCalls. SymGX analyzes ECalls from the untrusted world, skipping detailed analysis of OCalls for efficiency. Any values returned from OCalls are assumed to be tainted symbolic variables. Symbolically analyzing functions in the untrusted world is impractical due to complexity. Thus, only handling ECalls does not harm the soundness of SymGX, given our threat model.

Exploit Generation. SymGX identifies the location and type of cross-boundary vulnerabilities in SGX applications. It also forms a valid ECall sequence and parameters to trigger the vulnerability. As a result, SymGX can provide experts with above details to construct exploits. However, experts still need to manually verify the reports. In the future, we plan to invest new methods for automation.

7 CONCLUSION

Cross-boundary pointer vulnerability is a critial problem in SGX applications, due to the pointer-based data exchange between enclaves and untrusted environments. However, current analysis methods are ineffective in detecting such vulnerability since they cannot model three distinct features of SGX applications: Multi-entry Arbitrary-order Execution, Stateful Execution, and Context-aware Pointers. We propose a new symbolic execution technique called SymGX. Our tool builds upon a novel analysis model called GSTG-CAP that handles the above three properties. According to our evaluation, SymGX detected 30 new vulnerabilities in 14 open-source projects and found all 13 known vulnerabilities in Microsoft Open Enclave SDK and Google Asylo. In comparison, existing techniques only reported a small subset of the new and known vulnerabilities. Overall, SymGX is superior to existing solutions in detecting cross-boundary pointer vulnerabilities.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers. Ding Li and Yao Guo are corresponding authors. This work was partly supported by the National Key R&D of China (2022ZD0119103) and the National Natural Science Foundation of China (62172009,62141208).

REFERENCES

[1] 2023. Fix memory range check on ecall #1553. https://github.com/microsoft/CCF/pull/1553.
[2] Fritz Alder, Jo Van Bulck, Jesse Spielman, David Oswald, and Frank Piessens. 2022. Faulty point unit: ABI poisoning attacks on trusted execution environments. *Digital Threats: Research and Practice (DTRAP)* 3, 2 (2022), 1–26.
[3] Ghous Amjad, Seny Kamara, and Tarik Moataz. 2019. Forward and backward private searchable encryption with SGX. In *Proceedings of the 12th European Workshop on Systems Security*. 1–6.
[4] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA.
[5] Pedro Antonino, Wojciech Aleksander Woloszyn, and AW Roscoe. 2021. Guardian: Symbolic validation of orderliness in SGX enclaves. In *Proceedings of the 2021 on Cloud Computing Security Workshop*. 111–123.
[6] asonnino. 2023. Github page of sgx-wallet. https://github.com/asonnino/sgx-wallet.
[7] Gbadebo Ayoade, Vishal Karande, Latifur Khan, and Kevin Hamlen. 2018. Decentralized IoT data management using blockchain and trusted execution environment. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 15–22.
[8] Qinkun Bao, Zihao Wang, James R Larus, and Dinghao Wu. 2021. Abacus: Precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 797–809.
[9] Erick Bauman and Zhiqiang Lin. 2016. A Case for Protecting Computer Games With SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution, SysTEX@Middleware 2016, Trento, Italy, December 12, 2016*. ACM, 4:1–4:6.
[10] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. 2018. SGXElide: Enabling Enclave Code Secrecy via Self-Modification. In *Proceedings of International Symposium on Code Generation and Optimization*. Vienna, Austria.
[11] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The guard's dilemma: efficient code-reuse attacks against intel {SGX}. In *27th {USENIX} Security Symposium*. 1213–1227.
[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical.. In *WOOT*. 11–11.
[13] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*. 1–13.
[14] Sébastien Carré, Adrien Facon, Sylvain Guilley, Sofiane Takarabt, Alexander Schaub, and Youssef Souissi. 2019. Cache-Timing Attack Detection and Prevention: Application to Crypto Libs and PQC. In *Constructive Side-Channel Analysis and Secure Design: 10th International Workshop, COSADE 2019, Darmstadt, Germany, April 3–5, 2019, Proceedings 10*. Springer, 13–21.
[15] Swarup Chandra, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani Thuraisingham. 2017. Securing Data Analytics on SGX With Randomization. In *Proceedings of the 22nd European Symposium on Research in Computer Security*. Oslo, Norway.
[16] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy*.
[17] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1580–1596.
[18] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 841–858.
[19] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. 2022. {SGXFuzz}: Efficiently Synthesizing Nested Structures for {SGX} Enclave Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3147–3164.
[20] The Google Corporation. 2023. Asylo. https://asylo.dev.
[21] The Micosoft Corporation. [n. d.]. Merge pull request from GHSA-525h-wxcc-f66m. https://github.com/openenclave/openenclave/commit/bcac8e7acb514429fee9e0b5d0c7a0308fd4d76b.
[22] The MITRE Corporation. 2020. CVE-2020-15224. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15224
[23] The MITRE Corporation. 2020. CVE-2020-8904. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8904
[24] The MITRE Corporation. 2020. CVE-2020-8905. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8905
[25] The MITRE Corporation. 2020. CVE-2020-8935. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8935
[26] The MITRE Corporation. 2020. CVE-2020-8936. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8936
[27] The MITRE Corporation. 2020. CVE-2020-8937. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8937
[28] The MITRE Corporation. 2020. CVE-2020-8944. Retrieved April 30, 2023 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8944
[29] The Micosoft Corporation. 2023. Open Enclave SDK. https://openenclave.io/sdk.
[30] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).
[31] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. 2017. Secure processors part I: background, taxonomy for secure enclaves and Intel SGX architecture. *Foundations and Trends® in Electronic Design Automation* 11, 1-2 (2017), 1–248.
[32] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. 2017. Secure processors part II: Intel SGX security analysis and MIT sanctum architecture. *Foundations and Trends® in Electronic Design Automation* 11, 3 (2017), 249–361.
[33] ctz. 2023. Github page of sgx-pwenclave. https://github.com/ctz/sgx-pwenclave.
[34] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. 2021. Smashex: Smashing sgx enclaves using exceptions. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 779–793.
[35] Rongzhen Cui, Lianying Zhao, and David Lie. 2021. Emilia: Catching Iago in Legacy Code.. In *NDSS*.
[36] davidgmorais. 2023. Github page of varifiable-election. https://github.com/davidgmorais/verifiable-election.
[37] dsc-sgx. 2023. Github page of sgx-kmeans. https://github.com/dsc-sgx/sgx-kmeans.

[38] Benny Fuhry, Lina Hirschoff, Samuel Koesnadi, and Florian Kerschbaum. 2020. SeGShare: Secure group file sharing in the cloud using enclaves. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 476–488.
[39] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: speculative symbolic execution for cache timing leak detection. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 1235–1247.
[40] Shengjian Guo, Yueqi Chen, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang. 2020. Exposing cache timing side-channel leaks through out-of-order symbolic execution. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 147:1–147:32.
[41] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 377–388.
[42] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. 2023. Eunomia: Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, July 17-21, 2023*. ACM, 385–397.
[43] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. 2018. Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution. In *2018 IEEE 36th International Conference on Computer Design*. IEEE, 108–114.
[44] HNYuuu. 2023. Github page of SeeWasm. https://github.com/HNYuuu/SeeWasm.
[45] PLC INTEL. 2014. Intel software guard extensions programming reference.
[46] Jaehyuk Lee Jinsoo Jang Yeongjin Jang, Nohyun Kwak Yeseul Choi Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. (2017).
[47] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing system logs with SGX. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 19–30.
[48] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN attacks: On insecurity of enclave untrusted interfaces in SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 971–985.
[49] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. 2019. SGX-LEGO: Fine-grained SGX controlled-channel attack and its countermeasure. *computers & security* 82 (2019), 118–139.
[50] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.
[51] kudelskisecurity. 2023. Github page of sgx-reencrypt. https://github.com/kudelskisecurity/sgx-reencrypt.
[52] landoxy. 2023. Github page of intel-sgx-deep-learning. https://github.com/landoxy/intel-sgx-deep-learning.
[53] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. 2022. MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 978–988.
[54] Weijie Liu, Wenhao Wang, Hongbo Chen, XiaoFeng Wang, Yaosong Lu, Kai Chen, Xinyu Wang, Qintao Shen, Yi Chen, and Haixu Tang. 2021. Practical and efficient in-enclave verification of privacy compliance. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 413–425.
[55] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).
[56] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. 2016. Proof of luck: An efficient blockchain consensus protocol. In *proceedings of the 1st Workshop on System Software for Trusted Execution*. 1–6.
[57] Mohammad Hasanzadeh Mofrad and Adam Lee. 2017. Leveraging Intel SGX to create a nondisclosure cryptographic library. *arXiv preprint arXiv:1705.04706* (2017).
[58] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.
[59] occlum. 2023. Github page of sgx_protect_file. https://github.com/occlum/sgx_protect_file.
[60] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 264–278.
[61] Jaak Randmets. 2021. An overview of vulnerabilities and mitigations of intel sgx applications. *URL: https://cyber. ee/research/reports/D-2-116-An-Overview-of-Vulnerabilities-and-Mitigations-of-Inte l-SGX-Applications. pdf* (2021).
[62] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.
[63] rscosta. 2023. Github page of SGXCryptoFile. https://github.com/rscosta/SGXCryptoFile.
[64] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *NDSS*.
[65] Hovav Shacham, E Buchanan, R Roemer, and S Savage. 2008. Return-oriented programming: Exploits without code injection. *Black Hat USA Briefings (August 2008)* (2008).
[66] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. 2017. A Practical Encrypted Data Analytic Framework With Trusted Processors. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*. Dallas, TX.
[67] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *NDSS*.
[68] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
[69] Claudio Soriente, Ghassan Karame, Wenting Li, and Sergey Fedorov. 2019. Replicatee: Enabling seamless replication of sgx enclaves in the cloud. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 158–171.
[70] SRI-CSL. 2023. Github page of wllvm. https://github.com/SRI-CSL/whole-program-llvm.
[71] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D Garcia, and Frank Piessens. 2019. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1741–1758.
[72] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.
[73] Huibo Wang, Erick Bauman, Vishal Karande, Zhiqiang Lin, Yueqiang Cheng, and Yinqian Zhang. 2019. Running Language Interpreters Inside SGX: A Lightweight,Legacy-Compatible Script Code Hardening Approach. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. 114–121.
[74] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. 2019. Towards Memory Safe Enclave Programming with Rust-SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.
[75] Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. 2017. Binary code retrofitting and hardening using SGX. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. 43–49.
[76] Yongzhi Wang, Lingtong Liu, Cuicui Su, Jiawen Ma, Lei Wang, Yibo Yang, Yulong Shen, Guangxia Li, Tao Zhang, and Xuewen Dong. 2017. CryptSQLite: Protecting data confidentiality of SQLite with intel SGX. In *2017 International Conference on Networking and Network Applications (NaNA)*. IEEE, 303–308.
[77] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*. Springer, 440–457.
[78] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel® software guard extensions (Intel® SGX) software support for dynamic memory allocation inside an enclave. *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016* (2016), 1–9.
[79] Shixuan Zhao, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2022. vSGX: Virtualizing SGX Enclaves on AMD SEV. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy*. San Francisco, CA.
[80] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. 2020. Mptee: Bringing flexible and efficient memory protection to intel sgx. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.
[81] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.
[82] zyktrcn. 2023. Github page of sgx-rsa. https://github.com/zyktrcn/sgx_rsa.

A FULL EXPERIMENT RESULTS

In this section, we show the complete results of our evaluation. Figure 5 and Figure 6 show the coverage and memory consumption on the whole benchmark. Table 6 shows the details of each vulnerability we detect, including the number of ECalls (#ECalls) and the number of ECall parameters (#ECall Param.).

Table 6: Details of vulnerabilities

ID	Benchmark	Type	#ECall	#ECall Param.	ID	Benchmark	Type	#ECall	#ECall Param.
1	sgx-wallet	V2	2	4	16	SGX-pwenclave	V1	1	3
2	sgx-dnet	V1	1	3	17	sgx-biniax	V2	3	5
3		V2	1	3	18		V2	3	5
4	verifiable-election	V1	1	2	19	sgx_rsa	V1	1	6
5		V2	1	2	20		V1	1	6
6		V2	1	2	21		V2	2	10
7		V2	2	4	22		V2	3	14
8		V2	2	4	23		V2	3	15
9	sgx-kmenas	V2	2	8	24	sgx_protect_file	V1	1	3
10		V2	3	12	25		V1	1	3
11	CryptoEnclave	V1	1	4	26		V1	1	3
12	sgx-deep-learning	V1	1	4	27		V2	2	6
13		V1	1	4	28	SGXSSE	V1	1	4
14		V1	2	7	29		V2	1	4
15		V2	2	8	30		V2	3	12

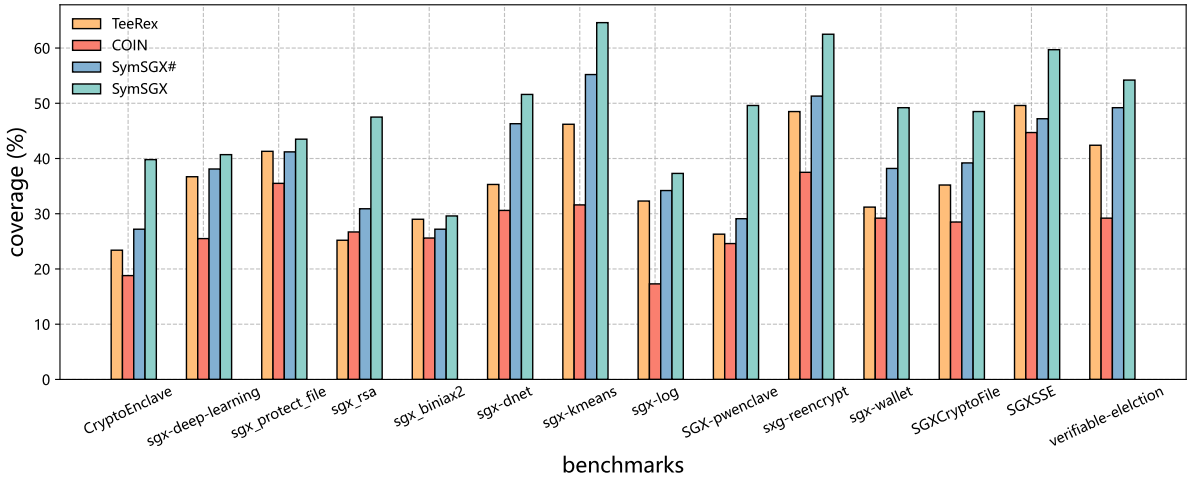

Figure 5: The complete coverage result of SymGX on the entire benchmarks

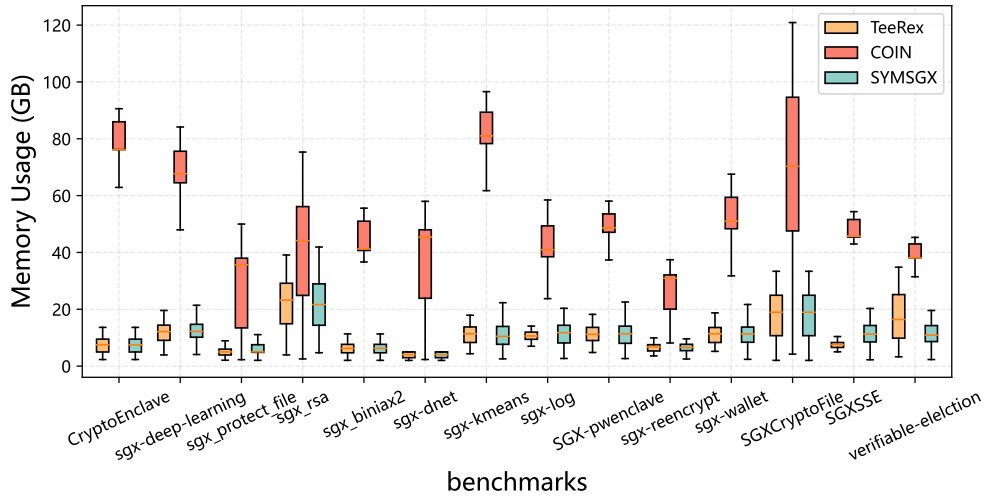

Figure 6: The complete memory consumption result of SymGX on the entire benchmarks