2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE) | 979-8-3503-1594-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSRE59848.2023.00016

SAPPX: Securing COTS Binaries with Automatic Program Partitioning for Intel SGX

Jiawei Huang*, Hao Han*, Fengyuan Xu†, Bing Chen*
*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
†State Key Laboratory of Novel Software Technology, Nanjing University, China
Email: {maxwell, hhan}@nuaa.edu.cn, fengyuan.xu@nju.edu.cn, cb_china@nuaa.edu.cn

***Abstract*—In the era of cloud computing, many applications are migrated to public servers not fully controlled by users who may fear their critical operations or data from being compromised by attackers. Previous studies have shown that Intel SGX enclaves can improve applications' security in many market products. Yet they mainly rely on developers to reprogram and recompile the application into an SGX-aware version. To address this problem, we propose SAPPX, an SGX-based program retrofitting method that can automatically partition COTS application binaries into two parts without breaking the original program semantics. The first part of the application runs in user space, while the second part is executed in an SGX enclave to protect the user's sensitive information. We have implemented a prototype of SAPPX on x86/Linux platforms and evaluated its performance using real-world applications and SPECCPU 2017 benchmarks. The experimental results show that the average overhead of the proposed approach is up to 19%.**

***Index Terms*—Partition, Intel SGX, Security enclave, COTS binary**

I. Introduction

Due to the significant benefits of cost efficiency, service agility, easy accessibility, and good scalability, offloading computation into public cloud/edge centers has become a prominent trend nowadays. Many government agencies and small business owners desire to migrate their applications to public cloud/edge platforms but still fear their critical operations or data from being compromised by attackers. A significant concern is that the infrastructure of the cloud/edge is not fully controlled by the customers who use the cloud/edge services. Hence, applications should be designed to protect sensitive data from being corrupted or stolen by privileged attackers. Although Fully Homomorphic Encryption (FHE) provides an excellent potential to secure applications running in untrusted environments as data are not required to be decrypted in memory to perform computation, speed and limited function support are major bottlenecks that prevent existing FHE schemes from being practical [1].

Intel Software Guard eXtensions (SGX) [2] is one of the promising solutions to secure applications in public cloud/edge centers. In 2017 Microsoft Azure [3] first used SGX to empower confidential computing services. Now SGX has become the mainstream solution for confidential computing services such as Aliyun, Baidu MesaTEE, and IBM Data Shield [4]. The core idea of SGX is creating an "enclave" which is a separated and encrypted region for running code and data. Intel SGX provides transparent memory encryption based on hardware to isolate the enclave from the external untrusted execution environment. The code/data located in the enclave are only decrypted inside the processor, so code/data integrity and confidentiality are ensured to the application even if the RAM is being read directly [5] or the privileged operating system and hypervisor outside the enclave are compromised [6].

Previous works have demonstrated the feasibility of executing ordinary applications for SGX within enclaves. For instance, Haven [7] intends to put the entire application into an enclave and has proposed a library-based approach to support unmodified binaries using SGX. This work predates the availability of SGX hardware, followed by SCONE [8], Graphene-SGX [9], and Panoply [10] seeking to improve the different aspects of efficiency on real SGX hardware.

Since not all library functions (e.g., file I/O operations: fopen) can be executed inside an SGX enclave, these approaches must load a series of supporting libraries into the enclave, resulting in *a huge trusted computing base (TCB)*. However, the resources of the Secure Space are limited and cannot be dynamically resized. In Intel SGX, the memory area used for storing code and data is called Enclave Page Cache (EPC), and the maximum size of an EPC is 128MB. Besides, the size of the Secure Memory of ARM TrustZone [11] and SEV Encrypted Memory of AMD SEV [12] need to be distributed in advance. Improper sizing could make the security containers inefficient.

To reduce the TCB size (i.e., security risks), Glamdring [13] splits applications' source code into different components and only loads the sensitive part into the enclave. To support legacy binary code, Wang et al. [14] extends an existing open-source binary reverse engineering platform (Uroboros [15]) to enable the SGX instrumentation. Yet it is still unclear what functions should be put into the enclave for protection given an application. In addition, not all applications can be reverse-engineered perfectly by existing "disassemble-reassemble" tools, thus significantly reducing the applicability of those approaches.

To solve these problems, we propose a new approach for <u>S</u>ecuring COTS application binaries with <u>A</u>utomatic <u>P</u>rogram <u>P</u>artitioning using Intel SG<u>X</u> technology, dubbed SAPPX. With the minimum human effort to mark sensitive input data to the program, such as parameters or authenticated files involved

Hao Han is the corresponding author.

in the program, SAPPX automatically determines a set of functions that should be placed inside an SGX enclave for protection. To reduce the runtime overhead of the partitioned program, SAPPX is responsible for adding or removing certain functions from the set. To overcome the drawback of binary rewriting, SAPPX leverages the idea of malware (e.g., virus) injection technique and creatively generates an SGX wrapper for the original application binary. We implemented a prototype of SAPPX and evaluated the system with real-world applications, including OpenSSL and thttpd, as well as SPECCPU 2017 benchmarks. The experimental results show that when the original application is retrofitted with security guarantees provided by SAPPX, the extra time consumed to execute the partitioned application is 0.1-0.3 times of the original version.

This work makes the following contributions:

- We designed a binary dependency analysis method that automatically determines a minimal set of sensitive functions to run inside an SGX enclave from user annotations. We also designed a boundary optimization algorithm to adjust the set size, making the partitioned program incur an affordable runtime overhead.
- We proposed a new approach to partitioning COTS application binaries into two parts: running in user space and an SGX enclave separately. A new binary infection technique is developed to fill the generated SGX library with original program binaries and preserve their semantics.
- We implemented and evaluated a prototype of SAPPX using benchmarks and real-world applications. Experiment results show our approach can achieve a small TCB at a reasonable overhead.

The rest of this paper is organized as follows. Section II describes the background of Intel SGX and how it protects applications from data-flow attacks with a motivating example. Section III presents the design of SAPPX and its security guarantees. Section IV presents the evaluations. Section V discusses the limitations in detail. Section VI reviews the related work, followed by conclusions in Section VII.

II. Background and Threat Model

A. Motivating Example

For ease of understanding, we use a motivating example throughout the paper. The following code snippet is modeled after a web server that loads the private key $privKey$ of the website through $loadPrivKey$ and uses it to establish a connection with the client by $GetConnection$. The user input $userInput$ is checked, and the corresponding file content will be retrieved by $getFile$ and sent back to the client. There is a stack overflow vulnerability in $getFile$, through which the $privKey$ may be stolen by maliciously designed $userInput$.

```
void getFile(char *reqFile, char *output){
    char fullPath[BUFSIZE] = "/path/to/root/", *
        result;
    strcat(fullPath, reqFile) ;
    // stack buffer overflow
    result = retrieve(fullPath) ;
    sprintf(output,"%s:%s",reqFile, result);
}
int server() {
    char *userInput, *privKey, output[BUFSIZE];
    privKey = loadPrivKey("/path/to/privKey");
    GetConnection(privKey, ...) ;
    // connection using privKey
    userInput = read_socket() ;
    if(checkInput(userInput)) {
        // user input OK , parse request
        getFile(getFileName(userInput), output);
        sendOut(output);
    }
}
```

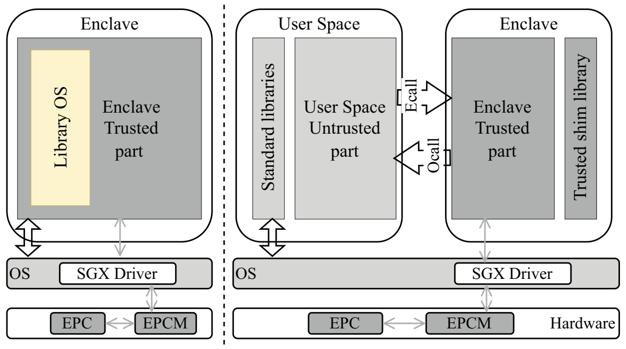


Fig. 1: Comparison between the non-partitioned model and SAPPX.

If putting $server$ into a trusted space, the stack of function $getFile$, where the overflow occurs, can never reach the area where the $privKey$ locates. Therefore, we believe that program partitioning is an effective way to avoid such an overflow and many other malicious attacks.

B. Trusted Execution with SGX

CPUs that support Intel SGX reserve a portion of memory space in DRAM, known as Processor Reserved Memory (PRM) [16]. The CPU protects this area from all instruction accesses from non-enclave memory, including kernel access, virtual machine manager, system management mode, and DMA accesses from peripherals. The most important part of the PRM area is the EPC, which consists of several 4 KB pages that store user-defined enclave code and data. If putting more code and data into an enclave (see left of Fig. 1), we will face a longer time to initialize the enclave and switch between user space and the enclave. That is why existing approaches that run an entire application inside an enclave often have unacceptable runtime performance. A better idea is to develop an SGX-secured application from scratch so that only critical or trusted components are executed inside an SGX enclave. As shown in the right of Fig. 1, the communication between user space and the enclave is achieved through Enclave Enter Call (Ecall) and Enclave Outer Call (Ocall) interfaces.

C. SGX-Secured Application

Programmers can use the Intel SGX SDK to develop an SGX-secured application by providing Enclave Definition Language (EDL) file(s), user space source file(s), and enclave source file(s). The EDL file contains the declarations of Ecalls and Ocalls, while their implementations are written in the enclave source and user-space source files, respectively. The `edge8r` tool – a text interpreter will generate user proxy functions for the programmer to invoke in the application's source code. For example, to secure the motivating example using SGX, an EDL file defines the function *server* in Enclave and the function *getFile* in user space. The local call of *getFile* at line 16 should be replaced by the generated proxy Ocall `sgx_status_t getFile(&ret, args...)`.

D. Threat Model and Assumptions

The goal of adversaries is to steal or modify users' confidential data without being noticed. An example adversary includes a malicious system administrator who controls the hardware and software of the machine on which the application is executed. We assume he can (i) access or modify data in memory or on disk (ii) monitor the execution path of the program; and (iii) modify the services provided by the operating system.

We assume that an attacker cannot exploit any function with security vulnerabilities isolated in an enclave. For example, if function $server$ of our motivating example is placed inside an enclave, attackers cannot peek into this execution stack and trigger the overflow attack. In addition, we do not consider malware running in the enclave to leak sensitive data or compromise the integrity of the code in the enclave.

We do not consider denial-of-service (DoS) attacks because there is no information leakage if the service is not provided. Side-channel attacks that use page faults are also out of the scope of this paper since enclaves cannot resist such attack alone [17]. We can rely on existing countermeasures [18], [19] for detection or mitigation.

III. System Design

To use the proposed SAPPX, users first annotate their sensitive data fed to the application, such as a key file for encryption or specific content in a configuration file. The output of the SAPPX is the retrofitted application with two partitions: the sensitive library running in an SGX enclave and an executable running in the user space. The SAPPX is designed to fulfill the following needs: 1) Finding a minimal set of sensitive functions that run in an SGX enclave; 2) Converting the original program to an SGX-secured version while preserving the program semantics; and 3) Imposing an acceptable performance overhead.

The workflow of SAPPX is shown in Fig. 2. Given the annotated input data, SAPPX uses symbolic execution with data flow analysis to determine a set of functions that will process those sensitive data. Simultaneously, each function's invocation cost (bytes of parameters and return) is estimated. Then, a boundary adjustment algorithm is proposed to move functions in/out of the above set to ensure compatibility and reduce the runtime overhead. Next, an SGX-secured version of the application is generated automatically. This application adds corresponding Ecall and Ocall interfaces and an SGX-compatible wrapper of the original program. Finally, the proposed binary infection method fills the generated application with the original binary code.

A. Binary Analysis Phase

SAPPX analyzes how the sensitive data are flowed among functions and determines a minimal set of relevant functions/data that should be placed in an SGX enclave. To support the analysis, we define a Colored Data Flow Graph (CDFG), where a vertex represents a function or a globally allocated memory object (e.g., global variables, program arguments, and syscall returns), and an edge indicates the existence of data flow between connected vertices. When a function reads or writes the sensitive data, the corresponding vertex in the graph is colored. Note that we do not consider the indirect data flow. For example, if x is sensitive, we will not mark y as sensitive given the statements if $x == a$ then $y = 1$ else $y = 0$.

The process of building the proposed CDFG is as follows: We first leverage existing static binary analysis methods [20]–[22] to compute the program's Call Graph (CG). However, existing static binary analysis tools can only produce an approximate CG due to the presence of indirect calls. Thus, the resulting CG may be incomplete. This problem is detected and fixed by dynamic analysis and runtime testing later. The detail is discussed in Section V. On top of CG, all global/static memory objects are added as new vertices, where all registers and memory addresses related to the annotated data are marked as initial tainted sources. Next, we use dependency analysis to mark other dependent tainted sources, including control and data dependency. To track the propagation of these tainted sources, we transform every source into a symbolic variable and use symbolic execution to cover as many branches as possible. This allows us to obtain a fine-grained data flow graph, including intra- and inter-functional data flows.

In the CDFG, all functions in the application are split into two sets: a tainted set S_e for an SGX enclave and an untainted set S_u running in the user space, respectively. All memory objects referenced by functions in S_e are denoted as the set G. Since they may exist in both user space and enclave, SAPPX synchronizes them as they enter and leave the enclave. In addition, we define an Ecall set F_e and an Ocall set F_o as follows:

$$F_e = \{f_j \in S_e : \exists f_i \in S_u, s.t.\ f_i \rightarrow f_j\}$$

$$F_o = \{f_i \in S_u : \exists f_j \in S_e, s.t.\ f_j \rightarrow f_i\}$$

Note that $F_e \subseteq S_e$ and $F_o \subseteq S_u$.

Fig. 3 shows an example of the CDFG. In this example, $data_1$ and $data_2$ are annotated as sensitive variables. Since f_1 and f_2 directly use $data_1$ and $data_2$, these two functions are also colored. Suppose f_1 passes $data_1$ to f_3 but not to f_4, thus f_3 is colored while f_4 is not. Additionally, we may color extra functions such as f_8 and f_9 for performance issues (see Section III-B for detail). Finally, the resulting sets are $S_e = \{f_1, f_3, f_2, f_8, f_9\}$, $S_u = \{f_0, f_4, f_5, f_6, f_7\}$, $F_e = \{f_1, f_2\}$, $F_o = \{f_4\}$, and $G = \{data_1, data_2\}$.

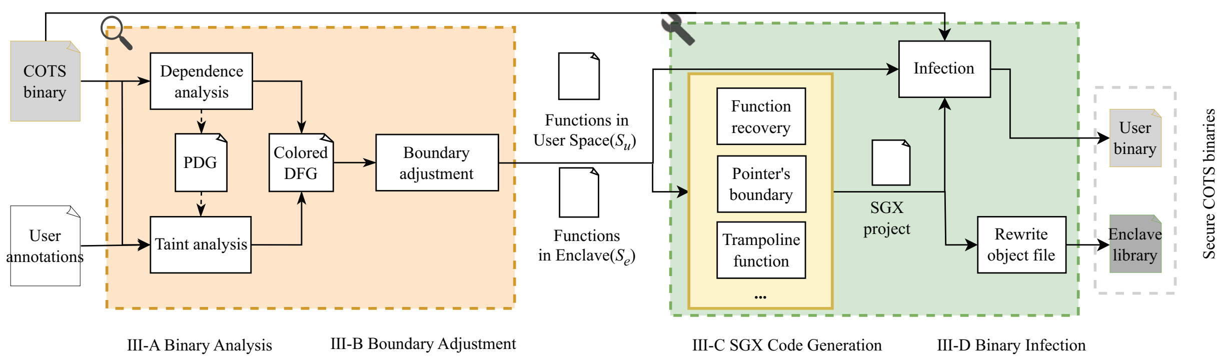


Fig. 2: Overview of the SAPPX workflow.

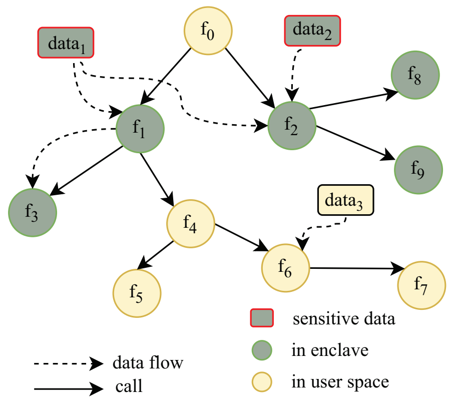


Fig. 3: Illustration of a colored data flow graph with sensitive data marked.

B. Boundary Adjustment Phase

In the boundary adjustment phase, we move functions from S_u to S_e or vice versa, adjusting F_e and F_o accordingly. The goals of this phase are twofold: 1) To reduce the frequency of enclave boundary crossings to reduce the performance overhead, and 2) to solve the incompatibility issue that some functions are not supported in an SGX enclave. It yields two constraints as follows:

- **Constraint 1: Incompatibility.** Since the SGX enclave does not support all x86 machine instructions [23], such as some privileged instructions and I/O instructions, the function containing these instructions must be removed from S_e to S_u. They are denoted as S_{limit}. Also, we need to exclude some complex functions from F_e and F_o to avoid potential runtime errors. For example, some functions with nested pointers as arguments are difficult to analyze their memory boundary correctly. These functions are denoted as F_{limit}.
- **Constraint 2: Performance.** We need to move some functions from S_u to S_e to avoid making E/Ocalls, since E/Ocalls often incur large runtime overhead [24]. To estimate the cost, we use the dynamic binary instrumentation tool Pin [25] to run the program under various test cases and record the number of bytes passed between caller and callee functions and their function size. For any $f_i \to f_j \in$ CDFG, we define m_{ij} and B_{ij} as the number of invocations and the average data size passed between the caller and the callee, respectively. If such a call were made through an O/Ecall interface, the cost could be estimated by:

$$cost_{E/Ocall}(f_i \to f_j) = \alpha \cdot m_{ij} \cdot B_{ij} + \beta, \qquad (1)$$

where α is a proportional factor and β is extra overhead for updating the SGX status. If the cost is too large, we consider moving f_j from S_u to S_e to reduce the overhead. Adding a function in S_e does not violate security properties but increases the enclave's TCB size. The formula for calculating the TCB size is as follows:

$$size(TCB) = \sum_{f_i \in S_e} size(f_i) + \sum_{g_i \in G} size(g_i) \qquad (2)$$

Given the above constraints, Algorithm 1 shows the proposed approach to find a quasi-optimal partition set in polynomial time. In the algorithm, lines 2-9 adjust the initial S_e and S_u within Constraint 1, and lines 11-22 search the space of partitioning strategies in the first fit order to handle Constraint 2. Since we want to exclude S_{limit} and F_{limit}, we assign the infinite cost to functions in F_{limit} and the infinite size to $size(f_i)$ where $f_i \in S_{limit}$.

In particular, S_{limit} are removed from the initial sensitive set S_e to S_u, and then F_e and F_o are re-calculated accordingly. Any function that makes an Ecall/Ocall to F_{limit} is moved to S_e to eliminate the incompatibility until there are no unsuitable functions in F_e and F_o. After this adjustment, the total cost is computed using Eq.(1) and Eq.(2) at line 10. Next, for each Ecall $f_j \to f_i$, we estimate if adding the inbound function f_j into an enclave could reduce the total cost. Similarly, for each Ocall $f_i \to f_j$, we search the outbound function f_j that could be merged into the enclave to avoid frequent switching. To balance the cost of passing parameters and the function size, the weight is heuristically set to 0.5. Note that we define only up to two layers of boundary adjustments to prevent moving all functions into the enclave. After the optimization,

Algorithm 1 Boundary Adjustment

```
Input: Initial $S_e, S_u$, $CDFG$, $S_{limit}$, $F_{limit}$, weight $\omega$
Output: Optimized sets $S'_e$ and $S'_u$.
 1: repeat
 2:     $S_e = S_e - S_{limit}; S_u = S_u + S_{limit}$;
 3:     Re-calculate $F_e$ and $F_o$ from $S_e$ and $S_u$;
 4:     while $\{f_j | f_j \in F_{limit} \cap (F_e \cup F_o)\} \neq \emptyset$ do
 5:         $S_e = S_e + \{f_x | \exists f_x \in S_u : f_x \to f_j \in F_e\}$;
 6:         $S_e = S_e + \{f_j\}$;
 7:         Update $S_u, F_e, F_o$ according to $S_e$;
 8:     end while
 9: until $(S_{limit} \cap S_e = \emptyset)$ & $(F_{limit} \cap (F_e \cup F_o) = \emptyset)$
10: $T_{cost} = \omega \cdot \sum cost_{E/Ocalls} + size(TCB)$;
11: for $f_i \in F_e$ do
12:     if $\exists f_j \in S_u, f_j \to f_i \in CDFG$ s.t. adding $f_j$ into $S_e$
        reduces $T_{cost}$ by $\Delta > 0$ then
13:         $S_e = S_e + \{f_j\}; S_u = S_u - \{f_j\}$;
14:         Update $F_e, F_o, T_{cost}$;
15:     end if
16: end for
17: for $f_i \in F_o$ do
18:     if $\exists f_j \in S_e, f_j \to f_i \in CDFG$ s.t. adding $f_i$ into $S_e$
        reduces $T_{cost}$ by $\Delta > 0$ then
19:         $S_e = S_e + \{f_i\}; S_u = S_u - \{f_i\}$;
20:         Update $F_e, F_o, T_{cost}$;
21:     end if
22: end for
23: $S'_e \leftarrow S_e; S'_u \leftarrow S_u$;
```

we determine what functions will run in enclave (S_u and S_e) and their boundary functions for Ecalls/Ocalls (F_e and F_o).

C. SGX Code Generation Phase

With the adjusted partitioning sets, the next step is to create an SGX project from which we can automatically compile and generate an SGX-secured version of the original program. The EDL file in the project includes the declaration of functions in F_e and F_o. We need to reconstruct the function type from the program binaries to declare those functions in a higher programming language, such as C/C++. If any function has a pointer-type parameter, the boundary information of such a pointer is also defined in the EDL file so that SGX knows the exact amount of memory to copy in and out of the enclave.

1) Recovery of the function: The function information that needs recovery includes the number of parameters, their types, and the return value type. The function name is not essential. The recovery of the exact function from the program binary is still an open problem that has never been fully solved. We do not derive the same type as declared in the source code. Instead, we attempt to reconstruct the number of parameters and their type widths such that the compiled binary uses the same registers and stack to pass parameters.

According to the calling convention of x86-64 [26], arguments may be passed through `REGISTER` and/or `MEMORY`. When the number of registers is not enough for storing

```
struct array{
    int arr[20];
};

typedef struct d{
    struct array a;
    float c, d;
    char e, f;
} data;

int test(data s,
        float c) {
    return
        s.a.arr[10] +
        (int)c;
}
```

(a) Source code

```
<test>:
push    %rbp
mov     %rsp,%rbp
movss   %xmm0,-0x4(%rbp)
mov     0x38(%rbp),%edx
movss   -0x4(%rbp),%xmm0
...
```

(b) Usage of arguments

```
struct stack_44{
        int arr[11];
};
ret_t test(stack_44 a0,
        float a1);
```

(c) Recovered parameters

Fig. 4: Example of recover parameter types

arguments (e.g., integer argument registers rdi, rsi, rdx, rcx, $r8$, $r9$, and floating point argument registers $xmm0$-$xmm7$) or the argument width is too long (e.g., a structure containing a data of 128bits or more), the caller will store the arguments on the stack every eight bytes. Similarly, the return value is passed using return registers or callee-saved registers like rax and rdx. Hence, how arguments are stored and used at call targets and call sites can be used to infer parameter count and types. We adopt the method proposed in τCFI [27].

By statically analyzing the use of parameter registers and the use of the stack from the called side, we can recover all the register parameters and stack references. In Fig. 4(b), we can notice that callee references the offset of *rbp* at 0x38, which means that there is a `MEMORY` type parameter access, so we calculate the maximum positive offset like this to get the size in bytes of the parameter passed using the stack. In this example, the bytes of stack parameters are (0x38 + 4(size of *edx*) - 0x10 (size of old *rbp* + return address) = 0x2C). The recovered argument type in Fig. 4(c) differs from the original source program in Fig. 4(a), but this is the maximum offset that can be used by callee.

2) Pointer's boundary: If a function has parameters of a pointer type, we need to tell SGX which parameters are pointers and their memory location/size through the EDL file to copy data into/out of the enclave. From program binaries, it is relatively easy to identify a pointer by searching for the offset and dereference instructions. However, it is difficult to determine the actual memory size a pointer points to because pointers do not carry any boundary information. We combine both static and dynamic binary analysis techniques to address this problem. In x86-64, pointers locate data by indirect addressing, the syntax of which can be simply summarized as `[base register + offset]`. In simple terms, as long as the base register holds the data to be analyzed, we consider the data as a pointer type.

To get the actual offset of a pointer, we rely on the dynamic binary analysis framework Triton [28], which can emulate the execution of machine codes and monitor the memory addresses where they are being written or read. We mark all memory addresses referenced by pointer parameters and set non-pointer parameters as symbolic values. By recording the reads and writes of the memory when a function is invoked dynamically (i.e., intraprocedural analysis), we obtain the offset of a pointer. The relationship between the pointer's boundary and the symbolic value is derived by changing the symbolic values of non-pointer arguments. How to handle nested pointers in a structure is discussed in Section V. SAPPX applies interprocedural analysis to obtain the maximum offset of pointer parameters in the interface functions of all S_e. Note that the pointer bounds derived from the analysis may not precisely reflect the true bounds. They just indicate the maximum memory offset to which the function(s) refer, and these bounds provide an estimate of the range of values that the pointers within the function(s) could take on.

3) Trampoline function: Firstly, the interface function declared in EDL generates the proxy function in a form inconsistent with the original function, as discussed in II-C. Secondly, since global variables may be used separately after the partition in both spaces, we need to synchronize them when switching spaces. Besides, trampoline functions can also be used to handle other concerns, such as function pointers as parameters and calls to standard library functions.

We define the trampoline functions of Ecalls as follows:

```
1  // in User Space
2  ret_type TE0_function(argi, ...){
3      ret_type res;
4      // use Ecall user interface to enter enclave
5      TE1_function(eid, &res, argi, ..., global);
6      // error collection
7      return res;
8  }
9  // in Enclave
10 ret_type TE1_function(argi, ..., global){
11     ret_type res;
12     // syn global variable to enclave;
13     res = function_name(argi, ...); // real Ecall
14     // syn global back to user space
15     return res;
16 }
```

The parameter *global* is a generic term for the contents of G, which needs to be synchronized into the enclave. The eid in line 5 is a global variable that is initialized at the enclave initialization phase and specifies the enclave's id. Besides, Ocalls also need trampoline functions in both spaces:

```
1  // in Enclave
2  ret_type TO1_function(argi, ...){
3      ret_type res;
4      TO0_function(&res, argi, ..., global);
5      // syn global variable to enclave;
6      return res;
7  }
8  // in User Space
9  ret_type TO0_function(argi, ..., global){
10     // syn global variable to user space;
11     return function_name(argi, ...);
12 }
```

We synchronize global variables before and after the execution of Ecalls and Ocalls, i.e., before entering and leaving the enclave. Over and above, we need to handle these issues:

Handle indirect calls. As discussed in III-A, existing call graph analysis software (e.g., Angr [29] and IDA Pro [30].) may not be able to analyze the complete call graph [31] because there may be some indirect calls in the programs [32]. In the x86-64 assembly programming language, we know that instruction `call *%rax` is the indirect call corresponding to the machine code `0xffd0`. Still, the value of the register rax, i.e., the address of the target function, is not necessarily in the specified space. The value of rax may come from the argument, constant, or relocation. So, we handled indirect calls in different cases:

- Arguments. For functions with a function pointer as a parameter, we used the trampoline function (inside enclave) to change the value of the argument by assigning the corresponding function pointer within the enclave to it or reporting `ERROR` to readjust the function sets.
- Constant and relocation. We applied backward slicing to analyze the possible values of the rax register and then use relocation items to fit them to III-D symbols of the corresponding function. The results can be used to fill CDFG, so the `ERROR` condition will not happen.

Handle memory allocation type library and other standard library functions. Enclave supports memory operation library functions such as *malloc* and *realloc*. The memory operation to allocate space for local variables and free them within the function body is enough to use the memory library functions inside the enclave. However, the memory allocated for global variables or returned values still needs to exist after the functions return to the user space, we use Ocall to allocate the same size of memory from user space and maintain a memory address correspondence table inside the enclave which can be used to release the space allocated inside the enclave and synchronize when we exit the enclave (at the end of Ecall trampoline function inside an enclave).

Standard library functions are not completely supported by SGX [23], and there are two cases to handle this difference: (a) Not supported by SGX. For functions such as file manipulations, we can use Ocalls to export the contents of the file we need to process in user space or input to the enclave. (b) Supported by SGX. We add the function directly to the symbol table and reference it in the relocation, see III-D.

Due to the usage of untrusted OS library functions, attackers who compromised the OS kernel can modify their return value to force enclave code to leak or modify sensitive enclave data. This type of attack is known as Iago attack [33]. These Ocalls that we generate for library functions are the Iago attack surface, so we perform a static derivation of the return value in the trampoline function using the function provided by SGX SDK: `sgx_is_within_enclave()` to determine whether the address points to the enclave address space.

After adding trampoline functions to the project, we can get the partitioned CDFG Fig. 5 obtained from part of Fig. 3 after partitioning. TE0 means the Trampoline of Ecall (E) in user space (0). The TE0_f$_1$ leads the partitioned application into an enclave and synchronizes global variable data$_1$ to the

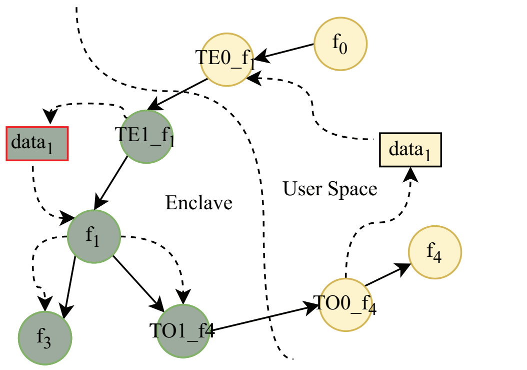


Fig. 5: Partitioned colored data flow graph with sensitive data marked.

```
1  ...
2  2d5f: 48 8d 35 ba 65 00 00 lea   0x65ba(%rip),%rsi
3  2d66: 48 89 c7              mov   %rax,%rdi
4  2d6e: e8 8d e3 ff ff        call  <function>
5  ...
```

```
1  ...
2  48 8d 35 00 00 00 00      lea   0x0(%rip),%rsi
3  48 89 c7                  mov   %rax,%rdi
4  e8 00 00 00 00            call  <function>
5  ...
```

Fig. 6: Rewrite example. Above is the original binary instruction content, and below is the binary content that should be filled in the enclave object file.

copies in the enclave. When switching to user space via Ocall, TO0_f_4 can synchronize $data_1$ back to user space.

D. Binary Infection Phase

We've gotten the SGX project with the required interface, the next step is to fill the raw binary content into the SGX project so that the working binaries preserve the functional integrity of the original application. For the enclave part, we rewrite the enclave relocation file, i.e. fill in the original code and data content, and then use the relocation table to link the bytes that need to be relocated with the data symbols. As for the contents of user space, in order to ensure the functional consistency of the partitioned program and the original program, we use the infection to join the original program with the SGX user program.

1) Rewrite enclave object file: Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Relocatable files have data that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. Relocation entries are these data.

For each relocation table entry, we need to obtain the subscript of the symbol in the symbol table to which the corresponding relocation refers, the offset of the byte to be relocated in that section, and the type of relocation (e.g., function call or symbol reference). For convenience, to facilitate the calls to some SGX-supported library functions, we added them to the symbol table and string table.

As shown in Fig. 6, the COTS binary has the relative addresses (line 2 and line 4) of all symbolic references in the original code. Through the register rip we can know where needs a relocation entry, and which symbol it links to. We choose to populate the object file with the contents of the original binary, so we need to collect all the relocation information in the function body (such as function calls and global symbol use) and the data or function to which the relocation refers, and set the bytes to be relocated to $0x0$. Besides machine code and relocation entry, we also need to put the data (including read-only data) to be used in the corresponding section and add the name of library functions into the symbol table and string table.

2) Infection: Based on the ELF binaries, we $deconstruct$ the generated SGX user binary and $reassemble$ it with the original executable file to get a working executable file that can interact with SGX enclave and has the same functionality as the original program. In Fig. 7, we infected the original binary file (code and data) to the end of the .bss segment. The main function of the original program is called after the enclave has been created and initialized, which means that the execution of the program follows the original logic completely. In addition, we should pay attention to the calls to shared library functions. Hence, the reassembled application needs to fit these requirements:

Lazy binding. For dynamically linked applications, ELF uses the Procedure Linkage Table (PLT) to implement lazy binding, i.e., binding a function (variable) only when a certain function (variable) is used, which can greatly speed up the start of the application. But after infection, the pushed content in line 7 may not be correct for the library functions. The function *_dl_runtime_resolve()* finds the address of the corresponding function in the dynamic link library based on the relocation information and puts it in the Global Offset Table (GOT), it has two arguments: module_ID, which is a fixed value as shown

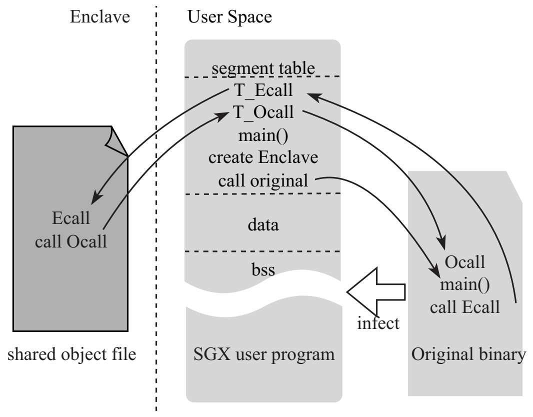


Fig. 7: Execution view of SAPPX project.

in Fig. 8(b) and the index of relocation table which is n in Fig. 8(a), so we need to modify n to a correct value.

Runtime check. For a dynamically linked executable file, the segments (program header, code, data, read-only data) of type `PT_LOAD` in the file are loaded into memory at a randomized base address; the kernel then maps the dynamic linker (DL) to the process address space and hands control over to DL, after completing its bootstrap, it performs explicit runtime linking, i.e., dynamically loading libraries needed by the execution: opening the shared library $dlopen$, finding the symbol $dlsym$, error handling $dlerror$, closing the dynamic library $dlclose$. The $dlsym$ is the core part, which checks all the contents of the symbol table (not only functions but also variables), so in order to ensure the partitioned program passes the check, we need to make appropriate changes to the symbol information and the corresponding shared library information of the binary after the infection.

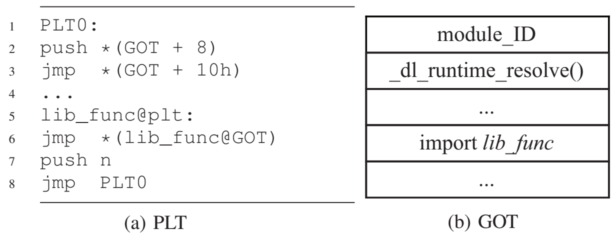
IV. Evaluation

We have implemented a prototype toolkit of SAPPX in C++, Python, and shell scripts, coupled with Pin, and Triton framework. We use Pin to record the sequence of function calls of the program during the trial run and functions' offset of read and write memory. Trition is used for emulating the executions of binary code and recording the maximum offset of pointer parameters.

To evaluate this prototype, we performed extensive experiments using real-world applications and benchmarks to answer the following questions:

1) **Correctness (Q1)**: Can binaries preserve their program semantics and function correctly after the partitioning?
2) **Effectiveness (Q2)**: Can SAPPX improve the security of COTS binaries?
3) **Efficiency (Q3)**: How much performance overhead does SAPPX impose on binaries for protection, compared to other related approaches?

Due to the page limit, we report the experimental results of some applications, including OpenSSL, thttpd, and SPECCPU 2017 benchmarks. All experiments were conducted on an Intel(R) Core(TM) i5-8400 CPU running at 2.80 GHz and 12 GB of RAM. The CPU supports Intel SGX hardware mode. SAPPX compiled the SGX projects with default optimization flags and debug symbols in `HW_PRERELEASE` mode using the G++ version 7.5.0 and Intel SGX SDK version 2.13. The ld.so program used for dynamic linking was version 2.27.



```
PLT0:
push *(GOT + 8)
jmp  *(GOT + 10h)
...
lib_func@plt:
jmp  *(lib_func@GOT)
push n
jmp  PLT0
```

Fig. 8: PLT modifying. (a) is PLT in the text segment, and (b) is the table to store the address of the corresponding library function.

A. Performance of Protecting OpenSSL (Q2 and Q3)

OpenSSL 1.1.1 [34] is a robust, commercial-grade, full-featured open-source toolkit for the transport layer security (TLS) protocol formerly known as the secure sockets layer (SSL) protocol. A stand-alone cryptographic library is provided for applications to use. We annotated that the calls to `memcpy` and the key in AES-CBC module were sensitive. For security, we annotated this function as sensitive and performed a binary analysis based on it. Since AES-CBC encrypts and decrypts 16 bytes of data as a block, we randomly generated binary data files of different numbers of blocks as input for OpenSSL and OpenSSL with SAPPX. By protecting the process of initializing and using keys, the risk of key compromise is greatly reduced.

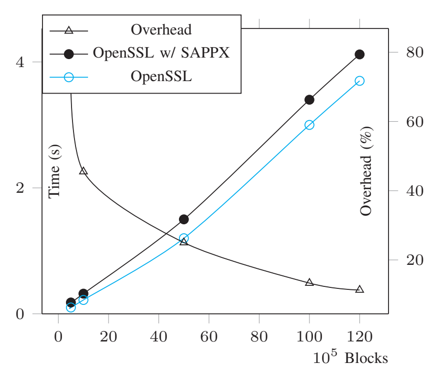
As shown in Fig. 9(a), the Overhead curve decreases because the enclave creation and destruction time is fixed (which is related positively to the size of the TCB), and it becomes smaller as the amount of data processed becomes larger and the processing time becomes longer. Eventually, the extra overhead of OpenSSL with SAPPX stays around 10% because of the additional time consumed by the memory operations that require copying the file contents to the enclave for processing and returning the results.

Besides, we used the `openssl speed` command with a specific number of bytes to measure the speed of AES-128,192,256-CBC algorithms. The *speed* utility performs a series of encryption and decryption operations using different sizes of randomly generated data. We take the average of the speed of these algorithms shown in Fig. 9(b). We found that the poor performance after the partition is due to the fact that the *speed* utility calls the AES-CBC (Ecall) frequently to encrypt a specific number of blocks in a fixed period of time. The fewer the number of blocks, the more frequently the space switches, resulting in encryption inefficiency.

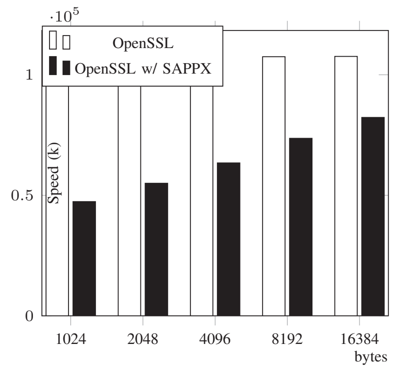
B. Performance of Protecting thttpd (Q2 and Q3)

Thttpd 2.29 [35] is an open-source software web server designed for simplicity, a small execution footprint and speed. It supports user authentication, i.e., the password file stored in the server is checked before each GET request. We used thttpd to build a local server and placed the different sizes of files in the corresponding html folder to provide download services. In terms of security, we annotated the server's local password file and the files to be downloaded by a client are security-sensitive. In this way, the authentication check of thttpd and the downloading of the files by the client will not be compromised by attackers. We used wget 1.91.4 [36] to record the request time of downloading different files and calculated the additional overhead under the SAPPX partition.

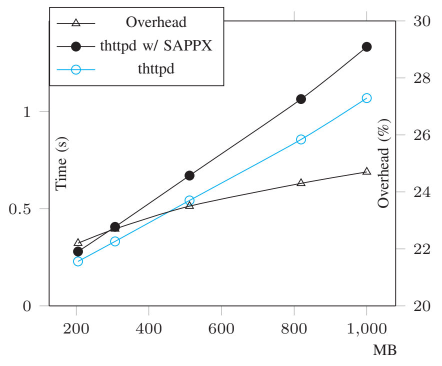
The overhead curve in Fig. 9(c) is unlike OpenSSL, because the creation and destruction time of the enclave is not recorded when downloading the file. Thttpd is a server program, and GET request is just plugged into the program's main loop that



(a) Partition OpenSSL stand-alone.



(b) OpenSSL speed benchmarks.



(c) Partition thttpd server to download files.

Fig. 9: Evaluations of SAPPX with real-world applications.

TABLE I
RESULTS FOR SPECCPU 2017 BENCHMARKS.

Benchmark	Binary Size	Sensitive Data	# of functions/in enclave	User Program Size[a]	TCB Size[b]	Overhead %
505.mcf_r	87.4KB	network_t net	40/5	78.2KB	162.8KB	**9.6**
519.lbm_r	67.0KB	LBM_Grid* srcGrid	20/3	85.3KB	174.2KB	**18.3**
525.x264_r	1.7MB	uint16_t cabac_size_5ones[]	837/30	927.7KB	318.7KB	**29.5**
538.imagick_r	4.4MB	UTFInfo utf_info[]	2960/11	3.1MB	276.1KB	**23.5**
544.nab_r	441.8KB	ATOM_T atab[]	343/7	323.9KB	162.8KB	**16.8**
557.xz_r	577.5KB	char bufs[]	543/13	329.8KB	243.7KB	**12.7**

[a]Information like debug section is not infected into the SGX user program, which makes the size may be smaller than the original binaries.
[b]TCB size is the size of the enclave-shared object file, utilities in EPC like enclave measurement and other security-critical functions are excluded.

does not participate in the server's initialization. The reason for the slow rise in the additional overhead of thttpd with SAPPX is the increase in the proportion of Ocalls. The writing of the contents of the file being downloaded is implemented using the library functions *write* and *writev*, which are Ocalled back to user space for execution under the partition of SAPPX. It should be mentioned that the speed of the network is the most critical part of the download latency in practice.

C. Experiment Results of SPECCPU 2017 (Q1 and Q3)

SPECCPU 2017 [37] benchmarks are designed to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications and producing deterministic results. Using them, we can well demonstrate the correctness as well as the efficiency of SAPPX. These benchmarks are computationally intensive applications, not security-sensitive ones. However, we believe it was important for SAPPX to use computationally intensive benchmarks because they stress-tested SAPPX's detection mechanism.

We randomly selected a global variable for each benchmark, annotated it as sensitive, and provided it to SAPPX. Table I presents the experimental results. The type of information comes from IDA Pro. The enclave-shared object file contains not only functions and data split from the original program but also many functions and data that are used to support the operations in the enclave, such as encryption and switching, therefore, the size of enclave shared object file may be larger or smaller than the original program.

We run these benchmarks in rate mode. The runtime overhead of SAPPX comes from the switching between user space and enclave, which means the frequency of switching determines the overhead. With the help of dynamic binary instrumentation, we avoid the high frequency of switching between these two spaces. The results presented in Table I show that the additional runtime overhead of the benchmarks with SAPPX is 18.4% on average, and the TCB is lightweight.

D. Comparison to Other Work

As shown in table II, Haven puts unmodified binaries into the enclave, while Graphene-SGX puts binaries generated by source code in a particular compilation environment into the enclave. They both need to load a series of trusted shared libraries into the enclave, which undoubtedly leads to huge TCB. It is a violation of the *principle of least privilege*: all enclave code executes at the level of privilege that allows it to access sensitive data [38].

The use of static linking causes SCONE and Panoply's TCB size to be not as small as they described and low code reusability. Panoply, still needs to make changes to the source code to fit their architecture, which greatly reduces the efficiency of development. Glamdring introduces an automatic partitioning method to obtain small TCB, but requires the user to annotate source and sink in source code, requiring the user to have an overall knowledge of the C program. Instead, SAPPX only needs the user to mark the sensitive data source, i.e., the input on which the program relies, it can be a file or other input as well. So, using our partitioning tool, the user

TABLE II
COMPARISON OF PROTECTING APPLICATIONS USING INTEL SGX.

Model Name	**Source Code**	**Modify Source**	**Link**	**TCB Size Level (to original)**	**Overhead**
Haven	✗	-	dynamic	huge	31%-54%
SCONE	✓	✗ (library)	static	60%–200%	0.6x–1.2x (throughput)
Panoply	✓	✓	static	105.8%	24%
Graphene-SGX	✓	✗	dynamic	huge	<200%
Glamdring	✓	✗ (annotations)	dynamic	optional	25%-200% (optional)
SAPPX	✗	-	dynamic	6.3%-260% (optional)	9.6%-29.5% (optional)

only needs to know the basic features of the program and the sensitive data it may deal with.

V. DISCUSSION

Nested pointer. As discussed in III-C, by assigning a fixed value to the memory to which pointed by the pointer and monitoring memory reads and writes during the dynamic execution of the function, we can know whether the pointer is referenced as a nested pointer or not. In addition to this, there is another usage of nested pointers that can be identified by static analysis: the memory to which the pointer offset points is assigned as the value of another pointer.

```
p, q; // Known: p, q are pointers
p->offset = q; // p is a nested pointer
```

Nested pointers bring uncertainties, such as a linked list which needs the exact memory layout of the structure, and the list pointer needs to be distinguished from the other pointers. PtrSplit [39] is a framework for partitioning pointers into remote-procedure calls (RPCs) for safe executions, which "solves" the pointer memory copy problem by defining the layers of deep copies of pointers.

Indirect call. As discussed in III-C, there is another situation: the value of rax comes from a variable. The value of the variable is changeable, so we cannot use relocation or trampoline functions to get to the function it points to. Previous work has proposed some solutions, but reliability cannot be guaranteed: Balakrishnan et al. [32] present value-set analysis (VSA) which determines an over-approximation of the set of addresses that each data object can hold at each program point, it can get all the possible value of register rax. DEEPVSA [40] uses deep neural networks to improve alias analysis for VSA to get the value more accurately.

Finally, SGX's interface functions do not support using class objects passed as parameters. But in our implementation of SAPPX, all object member functions passed `this` are treated as ordinary pointers, and all member data is available via its offset. But the virtual function table pointer `vptr` in it brings a lot of uncertainty, which is the reason why SGX does not allow passing objects at the interface.

VI. RELATED WORK

Secure application with TEE. Trust Execution Environment (TEE) is designed to protect sensitive data and code from malicious attacks. TEEs typically rely on hardware-based isolation mechanisms, such as Intel SGX or ARM TrustZone, to create a secure environment that is isolated from the rest of the system. Park et al. [41] combine a multi-level security model and hardware enclave to get a nested enclave, which provides a separate internal module for each user to handle privacy-sensitive data while sharing the same library. Ryoan [42] uses SGX to provide hardware enclaves, and each enclave contains a sandbox instance that loads and executes untrusted modules for protecting applications from untrusted 3rd party libraries. Rubinov et al. [43] use taint analysis handles of confidential data in Android applications to run in the "secure" world and reduce the overhead of transitions between the secure and normal worlds.

Partition for execution. Partitioning programs for execution are the most straightforward way to protect sensitive content. Previous studies of program partition execution have included differences in what to protect and how to partition, and even the granularity of the partition. RT-trust [44] checks the validity of the annotation, analyzes whether the converted system will continue to satisfy the original real-time constraints via custom static analysis, and with feedback loops suggests how to modify the code. Program-mandering [45] selects metrics to adjust the boundaries of the constructed PDG automatically to reach privilege separation and turn function callings into RPCs. Trapp et al. [46] divide programs into different processes based on privileges (e.g., socket, file) and improve inter-process communication. Civet [47] is a partitioning framework using Intel SGX, but for Java applications, it contributes a partitioned Java runtime design. Besides Java, Ghosn et al. [48] automatically extract the Go language's secure code and data to the enclave, and uses low-overhead channels to switch between the enclave and user space. These methods allow SGX to expand the choice of target programs without being limited to C/C++.

VII. CONCLUSION

SAPPX employs a combination of static and dynamic binary analysis techniques to determine the subset of the application's code and/or data that should be secured within an SGX enclave, reduces the frequency of E/Ocall by boundary adjustment, and offers guarantees that the integrity of application data and code by infection and synchronize. Our experimental evaluations demonstrate that SAPPX offers sufficient utility for partitioning unmodified legacy COTS binaries. Moreover, it introduces a reasonable additional overhead and maintains a small TCB.

ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their valuable comments helping us to improve this work. This work was supported in part by NSFC #61972200 and #62176122.

REFERENCES

[1] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully Homomorphic Encryption Compilers," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pp. 1092–1108, IEEE, 2021.

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, “Innovative instructions and software model for isolated execution,” in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013* (R. B. Lee and W. Shi, eds.), p. 10, ACM, 2013.

[3] C. T. O. Mark Russinovich and M. A. Technical Fellow, “Introducing Azure confidential computing.” https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[4] P. Karnati, “Data-in-use protection on ibm cloud using intel sgx,” *IBM, May*, 2018.

[5] S. Selvaraj, “Overview of Intel® Software Guard Extension Enclaves,” *Retrieved from Intel: https://software.intel.com/en-us/blogs/2016/06/06/overview-of-intel-software-guard-extension-enclave*, 2016.

[6] V. Costan and S. Devadas, “Intel SGX Explained.” Cryptology ePrint Archive, Paper 2016/086, 2016. https://eprint.iacr.org/2016/086.

[7] A. Baumann, M. Peinado, and G. C. Hunt, “Shielding Applications from an Untrusted Cloud with Haven,” *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 8:1–8:26, 2015.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, “SCONE: Secure Linux Containers with Intel SGX,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (K. Keeton and T. Roscoe, eds.), pp. 689–703, USENIX Association, 2016.

[9] C. Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX,” in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (D. D. Silva and B. Ford, eds.), pp. 645–658, USENIX Association, 2017.

[10] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, “Panoply: Low-TCB Linux Applications With SGX Enclaves,” in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, 2017.

[11] “ARM TrustZone.” http://www.arm.com/products/processors/technologies/trustzone/.

[12] A. Sev-Snp, “Strengthening vm isolation with integrity protection and more,” *White Paper, January*, vol. 53, pp. 1450–1465, 2020.

[13] J. Lind, C. Priebe, D. Muthukumaran, D. O’Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. M. Eyers, R. Kapitza, C. Fetzer, and P. R. Pietzuch, “Glamdring: Automatic Application Partitioning for Intel SGX,” in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017* (D. D. Silva and B. Ford, eds.), pp. 285–298, USENIX Association, 2017.

[14] S. Wang, W. Wang, Q. Bao, P. Wang, X. Wang, and D. Wu, “Binary Code Retrofitting and Hardening Using SGX,” in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017* (T. Kim, C. Wang, and D. Wu, eds.), pp. 43–49, ACM, 2017.

[15] S. Wang, P. Wang, and D. Wu, “UROBOROS: Instrumenting Stripped Binaries with Static Reassembling,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pp. 236–247, IEEE Computer Society, 2016.

[16] S. Gueron, “A Memory Encryption Engine Suitable for General Purpose Processors,” *IACR Cryptol. ePrint Arch.*, p. 204, 2016.

[17] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing,” in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017* (E. Kirda and T. Ristenpart, eds.), pp. 557–574, USENIX Association, 2017.

[18] Y. Xu, W. Cui, and M. Peinado, “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems,” in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pp. 640–656, IEEE Computer Society, 2015.

[19] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, “Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity,” in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[20] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “BAP: A Binary Analysis Platform,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 463–469, Springer, 2011.

[21] A. D. Federico, M. Payer, and G. Agosta, “rev.ng: a unified binary analysis framework to recover CFGs and function boundaries,” in *Proceedings of the 26th International Conference on Compiler Construction, Austin, TX, USA, February 5-6, 2017* (P. Wu and S. Hack, eds.), pp. 131–141, ACM, 2017.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pp. 138–157, IEEE Computer Society, 2016.

[23] “Intel® Software Guard Extensions (Intel® SGX) Developer Guide.” https://www.intel.com/content/www/us/en/content-details/671334/intel-software-guard-extensions-intel-sgx-developer-guide.html.

[24] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, “Eleos: Exitless OS services for SGX enclaves,” in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017* (G. Alonso, R. Bianchini, and M. Vukolic, eds.), pp. 238–253, ACM, 2017.

[25] “Pin.” https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[26] H. Lu, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, “System v application binary interface,” *AMD64 Architecture Processor Supplement*, pp. 588–601, 2018.

[27] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, “τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries,” in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 423–444, 2018.

[28] F. Saudel and J. Salwan, “Triton: A Dynamic Symbolic Execution Framework,” in *Symposium sur la sécurité des technologies de l’information et des communications*, SSTIC, (Rennes, France), pp. 31–54, June 2015.

[29] “angr.” https://angr.io/.

[30] “IDA Pro.” https://hex-rays.com/IDA-pro/.

[31] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, “An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries,” in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016* (T. Holz and S. Savage, eds.), pp. 583–600, USENIX Association, 2016.

[32] G. Balakrishnan and T. W. Reps, “Analyzing Memory Accesses in x86 Executables,” in *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings* (E. Duesterwald, ed.), vol. 2985 of *Lecture Notes in Computer Science*, pp. 5–23, Springer, 2004.

[33] S. Checkoway and H. Shacham, “Iago attacks: why the system call API is a bad untrusted RPC interface,” in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013* (V. Sarkar and R. Bodík, eds.), pp. 253–264, ACM, 2013.

[34] The OpenSSL Project, “OpenSSL: The Open Source toolkit for SSL/TLS.” www.openssl.org, April 2003.

[35] J. Poskanzer, “thttpd-tiny/turbo/throttling HTTP server,” *http://www. acme. com/software/thttpd*, 2000.

[36] “wget.” https://www.gnu.org/software/wget/.

[37] J. Bucek, K.-D. Lange, and J. v. Kistowski, “SPEC CPU2017: Next-generation compute benchmark,” in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 41–42, 2018.

[38] J. H. Saltzer and M. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[39] S. Liu, G. Tan, and T. Jaeger, “PtrSplit: Supporting General Pointers in Automatic Program Partitioning,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, eds.), pp. 2359–2371, ACM, 2017.

[40] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, “DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program

Analysis," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (N. Heninger and P. Traynor, eds.), pp. 1787–1804, USENIX Association, 2019.

[41] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pp. 776–789, IEEE, 2020.

[42] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 13:1–13:32, 2018.

[43] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, "Automated partitioning of android applications for trusted execution environments," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016* (L. K. Dillon, W. Visser, and L. A. Williams, eds.), pp. 923–934, ACM, 2016.

[44] Y. Liu, K. An, and E. Tilevich, "RT-trust: automated refactoring for trusted execution under real-time constraints," in *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018* (E. V. Wyk and T. Rompf, eds.), pp. 175–187, ACM, 2018.

[45] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative Privilege Separation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019* (L. Cavallaro, J. Kinder, X. Wang, and J. Katz, eds.), pp. 1023–1040, ACM, 2019.

[46] M. Trapp, M. Rossberg, and G. Schäfer, "Automatic source code decomposition for privilege separation," in *24th International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2016, Split, Croatia, September 22-24, 2016*, pp. 1–6, IEEE, 2016.

[47] C. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, "Civet: An Efficient Java Partitioning Framework for Hardware Enclaves," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (S. Capkun and F. Roesner, eds.), pp. 505–522, USENIX Association, 2020.

[48] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured Routines: Language-based Construction of Trusted Execution Environments," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019* (D. Malkhi and D. Tsafrir, eds.), pp. 571–586, USENIX Association, 2019.