A Comparison Study of the Compatibility Approaches for SGX Enclaves

Jinhua Cui[†], Yiyun Yin[†], Zhiping Cai[‡] and Jiliang Zhang[†§*]
[†]College of Semiconductors (College of Integrated Circuits), Hunan University
[§]Innovation Institute of Industrial Design and Machine Intelligence Quanzhou-Hunan University
[‡]College of Computer, National University of Defense Technology

***Abstract*—Confidential computing technologies, such as that enabled by Intel SGX (Software Guard eXtensions), have been widely deployed in various commercial cloud platforms. Specifically, SGX uses hardware-isolated compartments named enclaves to shield user applications from Operating Systems (OSes) and hypervisors, thus providing confidentiality and integrity guarantees for code and data. However, some crucial problems are not fully analyzed yet, especially for the compatibility with binary applications. This work first delivers an overview of Intel SGX and reviews its five design constraints that may affect compatibility. Subsequently, we revisit three distinct compatibility solutions from the internals and analyze their impact on security, performance, and flexibility. At last, we lay out some fundamental lessons learned from prior SGX studies.**

***Index Terms*—TEEs, Intel SGX enclaves, compatibility approaches, confidential computing, design constraints**

I. Introduction

Confidential computing can leverage hardware-based Trusted Execution Environments (TEEs) to protect users' sensitive data and code. Unlike traditional encryption protection for both data at rest and in transit, TEEs are primarily used to protect data in use [1–4]. A prominent example is Intel SGX, which introduces a specialized hardware security extension to enable an isolated execution environment called *enclave*. SGX is able to offer guarantees of confidentiality and integrity of enclave-bounded data and code [5–8]. The original design of SGX provides a strong security threat model, where it distrusts cloud service providers, third-party services, system administrators, and other individuals who have access to the physical machine. Instead, it only trusts the Intel CPU's SGX implementation and the user's own program code. Compared to the traditional OS, the attack surface has been significantly reduced in the SGX trust model. As a relatively mature commercial product, SGX technology has been deployed in some cloud servers, such as Microsoft Azure [2] and Alibaba Cloud ECS [4]. Security-sensitive scenarios shielded by SGX include finance, healthcare, intellectual property, multi-party confidential data sharing, confidential machine learning, blockchain, and so on.

Although Intel SGX provides powerful hardware-level security protection for data in use, some constraints imposed by SGX design result in varying degrees of impact on the compatibility with unmodified applications. The specific issues can be summarized as follows: (1) Binary compatibility: due to the design constraints of Intel SGX, most SGX runtime frameworks [9–21], particularly including the official Intel SGX SDK, only provide source-level compatibility rather than binary-level compatibility. This leads to problems in usability and compatibility between SGX and unmodified or legacy applications, which is an open issue in SGX-enforced confidential computing. (2) Language runtime compatibility: since SGX only supports applications compiled with C/C++ by default, language runtimes (e.g., Python and R) cannot run directly in the SGX enclave. The compatibility between the language runtime and SGX enclave has become an independent research focus, as indicated by prior studies [22–25].

This paper first reviews five important design constraints of Intel SGX and systematically analyzes the compatibility impacts caused by each constraint. Then, it provides an overview of existing SGX compatibility solutions and discusses their advantages and disadvantages. Finally, it concludes with some empirical lessons from Intel SGX studies.

II. Overview of Intel SGX

Intel SGX [5–8] is a security extension of Intel CPU architecture, which introduces a set of new instruction codes and memory isolation mechanisms for protection of data in use. SGX allows userland applications to create a protected area in an isolated address space, known as an *enclave*. In this enclave, both user code and data are offered for guarantees of confidentiality and integrity, effectively preventing malicious privileged software from tampering with them. The implementation of SGX requires coordination among the processor, memory management components, BIOS, drivers, runtime software, and other software and hardware components. Each enclave memory region, also called Processor Reserved Memory (PRM), is a contiguous physical memory block pre-reserved from the entire Dynamic Random Access

*Jiliang Zhang is the corresponding author.

This work was funded by the National Key Research and Development Program of China under Grant No. 2022YFB3903800, the National Natural Science Foundation of China under Grant No. U20A20202, No.62122023, the Science and Technology Innovation Program of Hunan Province under Grant No. 2021RC4019, the Natural Science Foundation of Hunan Province (Grant No. 2023JJ40160), the Natural Science Foundation of Changsha City (Grant No. kq2208212), the Fundamental Research Funds for the Central Universities, and the National Key Research and Development Program of China (Grant No. 2022YFF1203001).

2023 IEEE 32nd Asian Test Symposium (ATS) | 979-8-3503-0310-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ATS59501.2023.10317965

Memory (DRAM). Its size is typically configurable through the BIOS (e.g., a default size of 128MB).

In the trust model of Intel SGX, only the SGX hardware and enclave software are considered trusted. All other software, including privileged software such as the OS, is considered untrusted. Enclave software may rely on services provided by non-enclave software, such as system calls and signal handling. SGX hardware provides two interfaces, synchronous and asynchronous, for switching between the OS and the enclave. In addition, SGX hardware restricts access to enclave memory (i.e., private memory) from untrusted software. However, enclave software can read from or write to memory regions outside the enclave boundary (i.e., public memory).

Intel SGX SDK officially provides a set of function calls for SGX applications through *ecall*s and *ocall*s. Specifically, the *ecall* is used to enter an enclave and invoke trusted functions, while the *ocall* is used to exit the enclave and invoke untrusted functions [6]. Therefore, a user application is able to call a *ecall* to execute code in the enclave and receive the return values. Likewise, the enclave can invoke a *ocall* to exit the enclave to use functions/services in the host process.

III. SGX Design constraints and the Impacts

SGX offers confidentiality and integrity protection for code and data in the enclave. All enclave memory is private and can only be accessed when executing in enclave mode. Data exchanged with the external world (e.g., the host application or the OS) must reside in unprotected public memory since external programs cannot access the enclave's private memory. During runtime, execution control can only synchronously enter the enclave through *ecall*s and exit the enclave through *ocall*s. The two interfaces are the primary means of implementing system calls (*syscall*s). Any exception in the enclave results in asynchronous entry-exit points, after which SGX relocates the current control flow to pre-defined points in the program. Additionally, if enclave execution is asynchronously interrupted, SGX saves the enclave execution context and restores it at a later entry point [5, 6].

A. Design constraints

Intel SGX enforces strict isolation strategies at certain interaction points between the OS and the enclave code to ensure enclave security. This subsection recalls five constraints (C1∼C5) imposed by SGX hardware design [15] in Table I.

B. Impacts on (in)compatibility

The aforementioned constraints are an important basis for understanding the incompatibility between SGX enclaves and the functionalities of untrusted software. The C1∼C5 affect SGX1-based runtime frameworks [6, 7] while C1∼C4 also apply to the latest version of SGX2. The C2 is relaxed much as Intel introduces a new dynamic memory allocation feature in SGX2, but this feature cannot address other constraints.

C1. Spatial partitioning. C1 ensures that any data located in the enclave's private memory, such as syscall parameters, cannot be accessed by the OS or host processes. To do this, the enclave explicitly manages copies of data in both public and private memory to enable external access while protecting it from malicious modifications. This operation is called a *two-copy mechanism* [15]. However, C1 disrupts functionalities such as syscalls, signal handling, futex for locking, and introduces non-transparency issues (e.g., synchronizing two data copies) and security risks (e.g., TOCTOU attacks [26]).

C2. Static partitioning. Applications may require dynamic adjustments to the size or permissions of enclave memory. These operations happen when dynamically loading library files (e.g., *dlopen*) or data files (e.g., *mmap*), executing dynamically generated code, creating a read-only all-zero data segment (e.g., *bss*), or implementing software-based isolation protection. However, constraint C2 is highly incompatible with these functionalities. To maintain compatibility with C2, applications need to carefully modify the corresponding semantics. This may involve weakening security protection (shifting from read or execute to read and execute), adopting a two-copy mechanism, or relying on alternative forms of isolation (e.g., software instrumentation).

C3. Non-shared enclave memory. SGX lacks a mechanism for sharing private memory across enclaves. This limitation poses compatibility issues with synchronization primitives, such as locks and shared memory, especially in cases where there is no trusted OS to facilitate synchronization. Maintaining two copies of shared locks may disrupt their intended semantics and create a challenge in synchronizing the data state between the two copies without relying on another trusted synchronization primitive.

C4. One-to-one enclave memory mapping. When an application needs to create a new virtual address mapping, such as using *malloc*, the OS assists in establishing the mapping. Typically, an application can ask the OS to map the same physical page to several different virtual addresses with either same or varying permissions. For example, within a process's memory space, the same file can be mapped at two different locations as read-only. However, on SGX, a single PA cannot be mapped to multiple enclave VAs. Any attempt to do so will trigger an exception of SGX memory protection.

C5. Fixed enclave entry point. SGX enforces that enclave entry and resumption only occur from pre-specified and statically identifiable entry points, which are determined at compile time. However, in unmodified enclave applications, unexpected entry points may occur in the event of exception instructions or illegal memory accesses. When re-entering the enclave, SGX requires that the program's execution context matches that at the time of exit. This differs from typical program behavior, where programs can be resumed in a signal handler with correct execution context set up by the OS. In the context of SGX enclaves, however, the program must be resumed at the exact instruction with the same enclave context where it exited. Otherwise, another exception is triggered.

IV. Compatibility approaches for SGX enclaves

In the aforementioned section, the five design constraints outlined affect the compatibility or usability of SGX enclaves.

TABLE I
CONSTRAINTS IMPOSED BY SGX DESIGN

	Design constraint	Description
C1	Spatial partitioning	SGX reserves a dedicated memory region as private for the enclave and the rest as public.
C2	Static partitioning	Enclave must statically specify the spatial partitioning.
C3	Non-shared enclave memory	An enclave has no support to share private memory across enclaves.
C4	One-to-one enclave memory mapping	The virtual addresses (VA) for enclaves is mapped one-to-one with the physical address (PA).
C5	Fixed enclave entry point	An enclave is allowed to resume execution only from its last exit point.

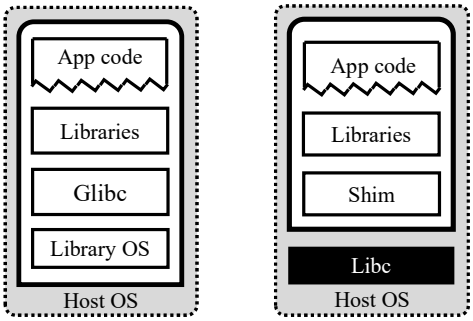


Fig. 1. Compatibility approaches for SGX enclaves. (a) Library OS-based design; (b) Library wrapper-based design

That means binary applications cannot be executed directly inside the enclave without modification or recompilation to the source code or binary file. To address the compatibility issue, researchers have proposed some approaches for running (unmodified or partially modified) applications within SGX enclaves. Among them, a prominent way is to have the application use prescribed program-level interfaces or APIs. The choice of interfaces varies, including specific programming languages [22–25], container interfaces [17], and specific implementations of standard libc interfaces [16, 18], among others. Figure 1 & 2 show three compatibility approaches, including library OS-based, library wrapper-based, and instruction wrapper-based. Although these approaches can enhance compatibility between SGX and applications, they may cause performance and security concerns. Next, we will discuss these compatibility-enhancing strategies individually and analyze their advantages and disadvantages, as well as their impact on performance, flexibility, and security.

A. Library OS-based Approach

The Library Operating System (i.e., libOS or library OS) is a special library that emulates the OS features and API interfaces relied upon by an application process. By porting the entire library OS into the enclave, application binaries can be executed directly in the same enclave. Since the binary program depends on the library OS completely, developers require relinking against specific versions of libraries (e.g., *musl*, *libc*, *glibc*). Then, the linked files are loaded into the enclave and run. The implementation of library OS includes rich functionalities such as those in a traditional OS, thus avoiding enclave applications frequently entering and exiting for the OS services. Frameworks of such library OS-based design include Haven [21], Graphene-SGX [16], and SGX-LKL [27].

Although, in this design (see Figure 1(a)), the library OS provides a user-space implementation for the majority of functionalities originally offered by the OS kernel, some privileged operations still need to be executed in the supervisor mode. For instance, both enforcement of protection and isolation to application code and page table switching are privileged operations. Thus, the library OS requires a small privileged software layer to help implement these operations. Since the library OS incorporates most of the functionalities of a traditional OS (e.g., file systems and network management), the interface between the library OS and the software layer is usually smaller than the system call layer exposed between the OS and the application. For example, the Graphene-SGX interface includes 38 different operations while Haven includes only 24. When the library OS wants to execute certain special instructions/operations, such as *cpuid* or *getsec*, it transfers control flow to the privileged software layer to exit the enclave and complete these operations with the assistance of the OS.

Advantages. The main advantage of the library OS-based approach is a simpler OS-enclave interaction interface. This is primarily attributed to the fact that the library OS itself implements numerous functionalities of a conventional OS, greatly simplifying the enclave interface. For example, the library OS may implement a significant portion of the file system in the enclave, allowing fine-grained control over read and write operations when the control flow reaches the enclave interface. For many system calls (e.g., *fcntl*), which require interaction between user space and kernel space in a traditional OS, the library OS running in the enclave does not require cross-boundary operations or even touch the enclave interface. These system calls can simply be mapped to read, write, and modification operations on the file system-related data structures in the library OS. The function calls also do not modify any security-sensitive states of other applications and do not require assistance from privileged system software.

> The library OS-based approach has a small OS-enclave interaction interface and gains better performance and compatibility, because the library OS in the enclave incorporates the majority of functionalities of a traditional OS relied on by any applications.

Disadvantages. An important drawback of this approach is that the entire library OS needs to run in a single enclave.

This means that the Trusted Computing Base (TCB) becomes significantly large. For example, the TCB of Graphene-SGX can be as high as $\sim$ 1.2MB. Consequently, the attack surface of the entire enclave also increases, providing attackers with more opportunities to exploit and compromise the isolated execution environment. One typical example is a buffer overflow vulnerability in the library OS, which can be leveraged to launch code-reuse attacks. Lack of flexibility is also one of the main drawbacks of the library OS-based approach. As mentioned in the literature of library OS-based prototypes, the entire application runs in the enclave. However, application developers likely put only a portion of the security-sensitive code in the enclave, rather than the entire program code.

Another drawback is the need for complex engineering efforts. Different enclave processes may need to communicate with each other, which involves multiple layers of communication mechanisms across library OSes. This design makes communication between enclaves more complicated. Whenever there is inter-enclave communication, the transmitted data needs to pass through the software layer of each library OS to be processed. Since the two enclaves do not trust each other, they cannot perform related operations as applications running in the same library OS (meaning in the same enclave).

> The library OS-based approach introduces a large TCB size that likely reduces the trustworthiness guaranteed by SGX hardware. Meanwhile, this design incurs high complexity and low flexibility.

B. Library Wrapper-based Approach

The representative work of the library wrapper-based approach, such as Panoply [19], assumes that applications invoke system services through library functions (e.g., the standard C library or *libc*), as shown in Figure 1(b). Typically, these library files contain low-level system call functions and other sensitive instruction operations that cannot be executed in the enclave. Panoply provides a library wrapper-based interface, allowing enclave applications to link against them. The type of wrapper interface ensures that the library code is called from outside the enclave, and the enclave interface is standard C library functions.

Currently, Panoply is the only runtime system that utilizes the library wrapper-based approach. In contrast to the library OS-based approach, this design implements a set of *exit* interfaces from the enclave, but adds few functionalities inside the enclave. In Panoply, standard C library functions are executed outside the enclave, and it provides a library wrapper-based interface with which the enclaved application needs to be linked. These wrapper functions wrap the data and transfer it to the corresponding library function outside the enclave.

Advantages. Since the library function wrappers in the enclave perform few operations for marshaling/unmarshaling data, it effectively reduces the TCB of the enclave system. Application developers can flexibly decide which security-sensitive functions to be executed in the enclave. The task of partitioning the application involves executing all enclave code as a separate module and implementing inter-function calls through cross-module execution. Creating the enclave code for applications is straightforward, as it only requires linking the function modules into the Panoply library function wrappers.

> The library wrapper-based approach involves a quite small TCB size, which prioritizes security over binary compatibility. Meanwhile, this design brings high flexibility.

Disadvantages. The most obvious drawback of the library wrapping approach is that the library function code executed outside the enclave is untrusted and can be exploited by attackers to compromise the confidentiality and integrity of the enclave application (i.e., Iago-like attacks). Due to the large and complex library wrapper interfaces, defending against such attacks is difficult.

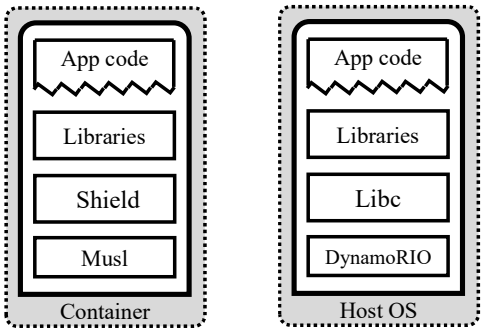
The standard C library contains thousands of function interfaces and some non-standard data structures, which may continuously vary (e.g., increase, modify, or delete). It is found that some APIs or data structures have been added or removed in different versions of libraries [28]. Therefore, SGX frameworks based on the library wrapper-based approach (e.g., Panoply) need huge engineering efforts for adaptation to different versions of library files.

In addition to the large wrapper interfaces, the Panoply system also requires modifications to the target application code to ensure that the corresponding function wrappers are executed correctly. In other words, the application's calls to functions in the library file must be adapted to call into the Panoply library file. According to the statistical data by the Panoply authors, approximately 1000 lines of code need to be modified for the test applications mentioned in the paper.

> The library wrapper-based approach contains large and complex library wrapper interfaces, which could potentially be exploited to launch Iago-like attacks. Further, this design involves huge porting efforts and manual modifications to application source codes, making it less compatible with SGX enclaves.

C. Instruction Wrapper-based Approach

The main purpose of the instruction wrapper-based approach is to provide wrapper functions for low-level instructions that are prohibited from execution in the enclave, such as $cpuid$, $rdtsc$, $syscall$, etc. These wrapper structures contain security rules that cross the enclave boundary to ensure the security of data. This approach can provide encryption services when data leaves the enclave, and perform decryption operations when entering the enclave. In this design, the enclave interface is a set of instruction wrapper-based interfaces used to intercept special instructions prohibited by SGX. It then transfers control and corresponding parameters to the outside of the enclave for further handling. SCONE [17] is an SGX runtime framework based on this approach, and Ratel [15] is another example where all the instructions of an application binary can be interposed in the enclave.



(a) (b)

Fig. 2. Two types of instruction wrapper-based design

The instruction wrapper-based approach provides finer-grained control over the enclave interface compared to the library OS-based and library wrapper-based approaches. Figure 2 illustrates two different design choices. In essence, not only can this approach intercept special instructions, but it can *transparently* monitor every instruction of the target program, such as Ratel (as shown in Figure 2(b)), to perform security checks, dynamic translation, profiling, etc. Compared to the library wrapper-based approach, the instruction wrapper-based occurs at a lower level of abstraction layer and is closer to the enclave boundary, such as operations on register parameters and enclave memory.

Advantages. Theoretically, the instruction wrapper-based approach can support the execution of arbitrary binary programs inside SGX enclaves by replacing the instruction invocation for out-enclave services with specified instruction wrapper-based interfaces. Implementing wrapper interfaces (e.g., wrapping instructions instead of giving a library call interfaces) at a lower level means that these wrapper interfaces work on a more stable interface. For example, the *syscall* instruction is commonly used to implement various system calls, and every instruction wrapper interface varies depending on syscall parameters and types. Research [28] has demonstrated that the Linux system call interface is more stable and undergoes fewer changes across different kernel versions than the *glibc* interface.

The instruction wrapper-based approach can offer higher compatibility and low-level control capabilities. This design brings high flexibility for implementing various in-line security monitors for security, profilers for performance, and other functionalities.

In this approach, the standard C library files that an application relies on run inside the enclave, while they run outside the enclave in the library wrapper-based approach. Therefore, the TCB of the former is larger and heavier than that of the latter. In addition, a thin layer of security checking programs can be flexibly added to prevent Iago-like attacks from malicious manipulation. In comparison, this approach requires little or no intrusive modifications to the application, making it easily adapt to SGX hardware enclaves. Importantly, controlling over the target binary is easy to achieve, which can be used for security introspection, monitoring, optimization, and so on.

Disadvantages. The primary overhead of the instruction wrapper-based approach is the TCB, which is larger than Panoply but much smaller than the library OS-based approach. Additionally, the application may need to relink against new instruction wrapper-based library files, such as those made by SCONE (see Figure 2(a)). Ratel provides an instruction-level dynamic translation engine in the enclave, which can dynamically translate and execute every instruction of the application binary. It replaces all instructions that are illegal in SGX with an appropriate external call, which is handled outside the enclave. Once done, it re-enters the enclave to continue execution. During this process, Ratel does not require dynamic or static link with any library files and enables the unmodified application to run directly in the enclave.

The instruction wrapper-based approach contains a relatively large TCB size. It may cause non-ignorable performance overhead, because frequently fine-grained control over the instruction of target binary requires plenty of extra handling.

V. Key Lessons Learned from SGX Studies

We have learned several valuable lessons from studies of Intel SGX, which involve aspects of performance and compatibility/usability.

Shared enclave memory. As described in Section II, Intel CPUs protect enclave memory from being accessed by all non-enclave software (e.g., the OS, hypervisor) that are restricted to only non-enclave memory regions. Enclave private memory is even not allowed to be shared with other enclaves. This memory isolation model enables strong security guarantees for each SGX enclave. This is why Intel SGX does not support shared memory by design across enclave boundaries. To initialize an enclave, for example, system software requires copying data from non-EPC pages to EPC pages to load the target application into the newly created enclave. Unfortunately, due to the lack of shared memory between the enclave and non-enclave, this operation incurs significant performance overhead. Another issue is that the OS- or host-enclave interactions (e.g., read/write syscalls) involve shallow or deep copies across enclave boundaries to exchange the data of syscall parameters, thus causing additional overhead. Furthermore, the absence of shared memory in SGX also leads to sweeping incompatibility with legacy applications.

Spatial isolation model. The core idea behind Intel SGX is to provide spatial memory isolation for enclaves from the external world and other enclaves. Memory can either be public or private, not both, on SGX-enabled systems. From a security perspective, this design works well, but it is inadequate for the program's expressiveness or usability. The next generation of TEEs should offer flexibility to a program where it can choose among strong security guarantees, better performance, and rich expressiveness (or better compatibility).

For example, programs should be allowed to convert private enclave pages to non-private ones on demand. Current SGX implementations do not permit such operations throughout the lifecycle of an enclave.

Context switch overhead. Many real-world applications are highly dependent on system services provided by the OS. SGX does not allow in-place execution of system calls in the enclave. Instead, system services can be called through the *ocall* interface pre-defined. When enclave codes require a system service, an *ocall* operation will be performed, and the *syscall* parameters will be copied from enclave private memory to non-enclave memory. Once the *syscall* is done, an *ecall* will be executed to re-enter the enclave. However, in some system calls, if the parameters include pointers and nested data structures, the user program may need to implement deep copy operations by itself. Although current newer versions of the SGX SDK already have support for deep copies, the performance overhead incurred is similar. As *ocall*s and *ecall*s need to switch back and forth between the enclave-OS, an amount of data copies and runtime software operations (e.g., save and restore enclave and non-enclave contexts) will cause significant impacts on the overall system performance. This issue may be mitigated to some extent by [29, 30].

VI. Conclusion

Intel SGX offers a hardware primitive for construction of isolated execution environments. However, due to certain design constraints in SGX, the problem of SGX compatibility/usability arises. This paper first recalls five design constraints of SGX and systematically analyzes the root causes of incompatibility (or poor usability) with binary software. Subsequently, we review three solutions to address SGX compatibility, and analyze their pros and cons from the perspectives of security, performance, and flexibility. Finally, we present some lessons learned from prior SGX studies.

References

[1] "Confidential computing consortium-open source community," http://confidentialcomputing.io/, Accessed May, 2023.

[2] "Microsoft azure confidential computing with intel software guard extensions (intel sgx)," https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/microsoft-confidential-computing-sgx-video.html, Accessed May, 2023.

[3] "The critical need for confidential computing," https://www.intel.com/content/www/us/en/security/critical-need-for-confidential-computing-report.html, Accessed May, 2023.

[4] "Alibaba cloud ecs bare metal instance supports intel sgx," https://www.alibabacloud.com/product/ebm?spm=a2c5t.10695662.1996646101.searchclickresult.539275ed2F9WWK, Accessed May, 2023.

[5] 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)*.

[6] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *HASP (2016)*.

[7] "Intel software guard extensions sdk-documentation — intel software," https://software.intel.com/en-us/sgx-sdk/documentation, Accessed May, 2023.

[8] C. V and D. S, "Intel sgx explained," http://eprint.iacr.org/2016/086, Accessed May, 2023.

[9] "Apache teaclave: a universal secure computing platform," https://teaclave.apache.org/, Accessed May, 2023.

[10] "Edgeless rt: an sdk and a runtime for intel sgx," https://github.com/edgelesssys/edgelessrt, Accessed May, 2023.

[11] "Enarx: an application deployment system enabling applications to runwithin tee," https://github.com/enarx/enarx/wiki/Enarx-Introduction, Accessed May, 2023.

[12] "A novel container runtime for cloud-native confidential computing and enclave runtime ecosystem," https://inclavare-containers.io/, Accessed May, 2023.

[13] "Veracruz: privacy-preserving collaborative compute," https://github.com/veracruz-project/veracruz, Accessed May, 2023.

[14] "Asylo: an open and flexible framework for enclave applications," https://asylo.dev/, Accessed May, 2023.

[15] J. Cui, S. Shinde, S. Sen, P. Saxena, and P. Yuan, "Dynamic binary translation for SGX enclaves," *ACM Trans. Priv. Secur.*, vol. 25, no. 4, pp. 32:1–32:40, 2022.

[16] C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *USENIX ATC (2017)*.

[17] 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 *OSDI (2016)*.

[18] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel SGX," in *ASPLOS (2020)*.

[19] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with SGX enclaves," in *NDSS (2017)*.

[20] "Open enclave sdk," https://openenclave.io/sdk/, 2022.

[21] A. Baumann, M. Peinado, and G. C. Hunt, "Shielding applications from an untrusted cloud with haven," in *OSDI (2014)*.

[22] H. Wang, E. Bauman, V. Karande, Z. Lin, Y. Cheng, and Y. Zhang, "Running language interpreters inside SGX: A lightweight, legacy-compatible script code hardening approach," in *AsiaCCS (2019)*.

[23] 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 *USENIX Security (2020)*.

[24] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured routines: Language-based construction of trusted execution environments," in *USENIX ATC (2019)*.

[25] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "Acctee: A webassembly-based two-way sandbox for trusted resource accounting," in *Middleware (2019)*.

[26] S. Checkoway and H. Shacham, "Iago attacks: why the system call API is a bad untrusted RPC interface," in *ASPLOS (2013)*.

[27] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: securing the host OS interface for trusted execution," *CoRR*, vol. abs/1908.11143, 2019. [Online]. Available: http://arxiv.org/abs/1908.11143

[28] K. Shanker, A. Joseph, and V. Ganapathy, "An evaluation of methods to port legacy code to SGX enclaves," in *ESEC/FSE (2020)*.

[29] O. Weisse, V. Bertacco, and T. M. Austin, "Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves," in *ISCA (2017)*.

[30] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *EuroSys (2017)*.