Practical and Efficient in-Enclave Verification of Privacy Compliance

Anonymous authors

***Abstract*—A trusted execution environment (TEE) such as Intel Software Guard Extension (SGX) runs attestation to prove to a data owner the integrity of the initial state of an enclave, including the program to operate on her data. For this purpose, the data-processing program is supposed to be open to the owner or a trusted third party, so its functionality can be evaluated before trust being established. In the real world, however, increasingly there are application scenarios in which the program itself needs to be protected (e.g., proprietary algorithm). So its compliance with privacy policies as expected by the data owner should be verified without exposing its code.**

To this end, this paper presents DEFLECTION, a new model for TEE-based delegated and flexible in-enclave code verification. Given that the conventional solutions do not work well under the resource-limited and TCB-frugal TEE, we come up with a new design inspired by Proof-Carrying Code that allows an untrusted out-enclave generator to analyze and instrument the source code of a program when compiling it into binary and a trusted in-enclave consumer efficiently verifies the correctness of the instrumentation and the presence of other protection before running the binary. Our design strategically moves most of the workload to the code generator, which is responsible for producing easy-to-check code, while keeping the consumer simple. Also, the whole consumer can be made public and verified through a conventional attestation. We implemented this model on Intel SGX and demonstrate that it introduces a very small part of TCB. We also thoroughly evaluated its performance on micro- and macro- benchmarks and real-world applications, showing that the design only incurs a small overhead when enforcing several categories of security policies.

I. INTRODUCTION

Recent years have witnessed the emergence of hardware trusted execution environments (TEEs) that enable efficient computation on untrusted platforms. A prominent example such as Intel SGX [1] has already been supported by major cloud providers today, including Microsoft Azure and Google Cloud [2], [3], and its further adoption has been facilitated by the Confidential Computing Consortium [4], a Linux Foundation project that brings together the biggest technical companies such as Intel, Google, Microsoft and IBM etc. However, before TEEs can see truly wide deployment for real-world confidential computing, key technical barriers still need to be overcome, *remote attestation* in particular.

Remote attestation. At the center of a TEE's trust model is remote attestation (RA), which allows the user of confidential computing to verify that the enclave code processing her sensitive data is correctly built and operates on a genuine TEE platform [5], so her data is well protected. This is done on SGX through establishing a chain of trust rooted at a platform attestation key which is used to generate a *Quote* – a signed report that contains the measurement of the code and data in an enclave; the Quote is delivered to the data owner and checked against the signature and the expected measurement hash. This trust building process is contingent upon the availability of the measurement, which is calculated from the enclave program either by the data owner when the program is publicly available or by a trusted third party working on the owner's behalf. This becomes problematic when the program itself is private and cannot be exposed. Programs may have exploitable bugs or they may write information out of the enclave through corrupted pointers easily. For example, pharmaceutical companies want to search for suitable candidates for their drug trial without directly getting access to plaintext patient records or exposing their algorithm (carrying sensitive genetic markers discovered with million dollar investments) to the hospital. With applications of this kind on the rise, new techniques for protecting both data and code privacy are in great demand.

Challenges. To address this problem, we present in this paper a novel *Delegated and flexible in-enclave code verification* (DEFLECTION) model to enable verification of an enclave program's compliance with user-defined security policies without exposing its source or binary code to unauthorized parties involved. Under the DEFLECTION model, a *bootstrap enclave* whose code is public and verifiable through the Intel's remote attestation, is responsible for performing the compliance check on behalf of the participating parties, who even without access to the code or data to be attested, can be convinced that desired policies are faithfully enforced. However, building a system to support this model turns out to be nontrivial, due to the complexity in static analysis of enclave binary for policy compliance, the need to keep the verification mechanism, which is inside the enclave's *trusted computing base* (*TCB*), small, the demand for a quick turnaround from the enclave user, and the limited computing resources today's SGX provides (about 93 MB physical memory on most commercial hardware [6]). Although the shielding runtimes such as Library OSes [7], [8], SCONE container [9], Ryoan sandbox [10] and the interpreters/compilers built for SGX [11], [12] enable confinement of unmodified binary in SGX enclaves, they all rely on a heavy interface layer for in-enclave service code to interact with the OS/Hypervisor [13], which introduces performance overhead. More importantly, the confinement mechanisms (sometimes including a whole interpreter) significantly increase the TCB, leading to the challenge in ensuring its security [14]. In our

This is a regular paper.

research, we studied popular shielding runtimes, Graphene-SGX even includes more than 100 kLoCs, more than 50 MB in binary, based upon analysis of its code.

A promising direction we envision that could lead to a practical solution is *proof-carry code* (*PCC*) [15], [16], a technique that enables a *verification condition generator* (*VC-Gen*) [17]–[19] to analyze a program and create a proof that attests the program's adherence to policies, and a *proof checker* to verify the proof and the code. The hope is to keep the VCGen outside the enclave while keeping the proof checker inside the enclave small and efficient. The problem is that this *cannot* be achieved by existing approaches, which utilize formal verification (such as [19], [20]) to produce a proof that could be considerably larger than the original code. Actually, today's formal verification techniques, theorem proving in particular, are still less scalable, difficult to handle large code blocks when constructing a security proof [21].

Our solution. In our research, we developed a new technique to instantiate the DEFLECTION model on SGX. Our approach, has been inspired by PCC, but relies on program analysis and SFI techniques, particularly out-of-enclave targeted instrumentation for lightweight in-enclave information-flow confinement, instead of heavyweight theorem proving to ensure policy compliance of enclave code. More specifically, DEFLECTION operates an untrusted *code producer* as a compiler to build the binary code for a data-processing program (called *target program*) and instrument it with a set of *security annotations* for enforcing desired policies at runtime, together with a lightweight trusted *code consumer* running in the bootstrap enclave to statically verify whether the target code indeed carries properly implanted security annotations.

To reduce the TCB and in-enclave computation, DEFLECTION is designed to simplify the verification step by pushing out most computing burden to the code producer running outside the enclave. More specifically, the target binary is expected to be well formatted by the producer, with all its indirect control flows resolved, all possible jump target addresses specified on a list and enforced by security annotations. In this way, the code consumer can check the target binary's policy compliance through lightweight *Recursive Descent Disassembly* to inspect its complete control flow (Section V-B), so as to ensure the presence of correctly constructed security annotations in front of each critical operation, such as load, store, enclave operations like OCall, and stack management (through a shadow stack). Any failure in such an inspection causes the rejection of the program. Also, since most code instrumentation (for injecting security annotations) is tasked to the producer, the code consumer does not need to make any change to the binary except relocating it inside the enclave. As a result, we only need a verifier with a vastly simplified disassembler, instead of a full-fledged, complicated binary analysis toolkit, to support categories of security policies, including data leak control, control-transfer management, self-modifying code block and side/covert channel mitigation in a small-size machine-language format (Section IV-B); in further work, other proofs could be extended given a formal model of the x64 instruction set (e.g., as in [22]). A wider spectrum of policies can also be upheld by an extension of DEFLECTION, as discussed in the paper (Section VII).

We implemented DEFLECTION in our research, building the code producer on top of the LLVM compiler infrastructure and the code consumer based upon the Capstone disassembly framework [23] and the core disassembling engine for x86 architecture. Using this unbalanced design, our in-enclave program has only 2000 lines of source code, which is significantly smaller than other shileding runtimes. Theorem prover Z3, with 26 MB used in Moat [21], is hard to be ported into SGX as well, not only because of its size but also its inability to be statically linked against an enclave. We further evaluated our implementation on micro-benchmarks (nBench), as well as macro-benchmarks, including credit scoring, HTTPS server, and also basic biomedical analysis algorithms.

DEFLECTION incurs on average (calculated by geometric mean) 20% performance overhead enforcing all the proposed security policies, and leads to around 10% performance overhead without side/covert channel mitigation. We have released our code on Github [24].

Contributions. The contributions are outlined as follows:

•*New model*. We propose DEFLECTION, a portable framework that extends today's TEE to maintain the data owner's trust in protection of her enclave data, without exposing the code of the data-processing program. This is achieved through enforcing a set of security policies through a publicly verifiable bootstrap enclave. This new attestation model enables a wide spectrum of applications with great real-world demand in the confidential computing era.

• *New techniques*. We present the design for instantiating DEFLECTION on SGX, following the idea of PCC. Our approach utilizes out-of-enclave code analysis and instrumentation to minimize the workload for in-enclave policy compliance check, which just involves a quick run of a well-formatted target binary for inspecting correct instrumentation. This simple design offers a much smaller TCB compared with existing solutions.

• *Implementation and evaluation*. We implemented our design of DEFLECTION and extensively evaluated our prototype on micro- and macro- benchmarks, together with popular biomedical algorithms. Our experiments show that DEFLECTION effectively enforces various security policies at small cost, with multiple level protections.

II. BACKGROUND

Intel SGX. Intel SGX [1] is a user-space TEE, which is characterized by flexible process-level isolation: a program component can get into an enclave mode and be protected by execution isolation, memory encryption and data sealing, against the threats from the untrusted OS and processes running in other enclaves. Such protection, however, comes with in-enclave resource constraints. Particularly, only 128 MB (256 MB for some new processors) encryption protected

memory is reserved. Although the virtual memory support is available, it incurs significant overheads in paging.

Another problem caused by SGX's flexibility design is a large attack surface. When an enclave program contains memory vulnerabilities, attacks can happen to compromise enclave's privacy protection. Prior research demonstrates that a ROP attack can succeed in injecting malicious code inside an enclave, which can be launched by the OS, Hypervisor, or BIOS [25]–[27]. Another security risk is side-channel leak [28]–[30], caused by the thin software stack inside an enclave (for reducing TCB), which often has to resort to the OS for resource management (e.g., paging, I/O control). Particularly, an OS-level adversary can perform a controlled side channel attack (e.g., [31]).

PCC. PCC is a mechanism that allows a host system to verify an application's properties with a proof accompanying the application's executable code. Using PCC, the host system is expected to check the validity of the proof, and compare the conclusions of the proof to its own security policies to determine whether the application is safe to run. Traditional PCC schemes tend to utilize formal verification for proof generation and validation. Techniques for this purpose includes verification condition generation [17], [32], theorem proving [33]–[35], and proof checking [36], which typically work on type-safe intermediate languages (IL) or higher level languages. A problem here is that up to our knowledge, no formal tool today can automatically transform a binary to IL for in-enclave verification. BAP [37] disassembles binaries and lifts x86 instructions to a formal format, but it does not have a runtime C/C++ library for static linking, as required for an enclave program.

Moreover, the PCC architecture relies on the correctness of the VCGen and the proof checker, so a direct application of PCC to confidential computing needs to include both in TCB. This is problematic due to their complicated designs and implementations, which are known to be error-prone [20]. Particularly, today's VCGens are built on interpreter/compiler or even virtual machine [18], and therefore will lead to a huge TCB. Prior attempts [38] to move VCGen out of TCB are found to have serious performance impacts, due to the significantly increased proof size growing exponentially with the size of the program that needs certified [15]. Although techniques are there to reduce the proof size [19], [38], they are complicated and increase the TCB size [36].

III. DEFLECTION

Consider an organization that provides data-processing services, such as image editing, tax preparation, personal health analysis and deep learning inference as a service. To use the services, their customers need to upload their sensitive data, such as images, tax documents, and health data, to the hosts operated by the organization. To avoid exposing the data, the services run inside SGX enclaves and need to prove to the customers that they are accessing to attested service programs. However, the organization may not want to release the proprietary programs to protect its intellectual property. This problem cannot be addressed by today's TEE design.

In this section, we present the *DElegated and FLexible in-Enclave Code verification* (DEFLECTION) model to allow the data owner to verify that the enclave code satisfies predefined security policy requirements without undermining the privacy of the enclave code.

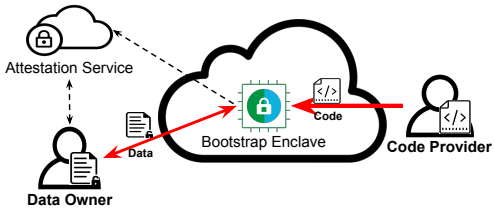

Fig. 1: The DEFLECTION model

A. The Delegation Model

Attestation service. Attestation service assists in the remote attestation process by helping the data owner verify the quote generated by an enclave, as performed by the Intel attestation service for SGX.

Bootstrap enclave. The bootstrap enclave is a built-in control layer on the software stack, hosted by the code provider or a third party cloud (see Figure 1). Its code is public and initial state is measured by hardware for generating an attestation quote, which is later verified by the data owner with the help of the AS. This software layer is responsible for establishing security channels with enclave users, authenticating and dynamically loading the binary of the target program from the code provider and data from its owner. Further it verifies the code to ensure its compliance with predefined security policies before bootstrapping the computation. During the computing, it also controls the data entering or exiting the enclave, e.g., through SGX ECalls/OCalls to perform data sanitization.

Data owner. The data owner uploads sensitive data (e.g., personal images) to use in-enclave services (e.g., an image classifier) and intends to keep her data secret during the computation. To this end, the owner runs a remote attestation with the enclave to verify the code of the bootstrap enclave, and sends in data through a secure channel only when convinced that the enclave is in the right state so expected policy compliance check will be properly performed on the target binary from the code provider.

Code provider. The code provider (owner) can be the service provider, and in this case, her target binary (the service code) can be directly handed over to the bootstrap enclave for compliance check. So, similar to the data owner, she can also request a flexible/portable remote attestation to verify the bootstrap enclave before delivering her binary to the enclave for a compliance check.

B. Guidelines

To instantiate a DEFLECTION System on a real-world TEE such as SGX, we expect the following requirements to be met by the design:

Minimizing TCB and resource consumption. Today's TEEs operate under resource constraints. Particularly, SGX is characterized by limited EPC. To maintain reasonable performance, we expect the software stack of the DEFLECTION model controls its resource use. The bootstrap enclave is responsible for enforcing security and privacy policies and for controlling the interfaces that import and export code/data for the enclave. So it is critical for trust establishment and needs to be kept as compact as possible for code inspection or verification [39].

Controlling portable code loading. The target binary is dynamically loaded and inspected by the bootstrap enclave. However, the binary may further sideload other code during its runtime. So the target binary, itself loaded dynamically, is executed on the enclave's heap space. Preventing it from sideloading requires a data execution prevention (DEP) scheme to guarantee the W $\oplus$ X privilege.

Preventing malicious control flows. Software stack should be designed to prevent the code from escaping policy enforcement by redirecting its control flow or tampering with the bootstrap enclave's critical data structures. Particularly, previous work shows that special instructions like ENCLU could form unique gadgets for control flow redirecting [26], which therefore need proper protection.

Minimizing performance impact. In all application scenarios, the data owner and the code provider expect a quick turnaround from code verification. Also the target binary's performance should not be significantly undermined by the runtime compliance check.

C. Threat Model

The delegation model is meant to establish trust between the enclave and the code provider, as well as the data owner, under the following assumptions:

• We do not trust the service code (target binary) and the platform hosting the enclave. In CCaaS, the platform may deliberately run vulnerable code to exfiltrate sensitive data, by exploiting the known vulnerabilities during computation.

• We assume that the code of the bootstrap enclave can be inspected to verify its functionalities and correctness. Also we consider the TEE hardware, its attestation protocol, and all underlying cryptographic primitives to be trusted.

• Our model is meant to protect data and code against different kinds of information leaks, not only explicit but also implicit. However, side channel in a user-land TEE (like SGX) is known to be hard to eliminate, which all existing SGX runtimes (including container [9], sandbox [10], interpreter [11] and others) fail to address. So our design for instantiating the model on SGX (Section IV-B) just aims to make the first step towards closing of this attack surface, mitigating some types of side-channel threats.

IV. DESIGN

In this section we present our design, which elevates the SGX platform with the support for the DEFLECTION model. This is done using an in-enclave software layer – the bootstrap enclave running the code consumer and an out-enclave auxiliary – the code generator. Following we first describe the general idea behind our design and then elaborate the policies it supports, its individual components and potential extension.

A. Overview

Idea. Behind the design of DEFLECTION is the idea of PCC, which enables efficient in-enclave verification of mobile code with proof. A direct application of the existing PCC techniques, however, fails to serve our purpose, as mentioned earlier, due to the huge TCB introduced, the large proof size and the exponential time with regards to the code size for proof generation. To address these issues, we design a lightweight approach with an untrusted code producer and a trusted code consumer running inside the bootstrap enclave. The producer compiles the source code of the target program with necessary libraries (for service providing), generates a list of its indirect jump targets, and instruments it with security annotations (in assembly with structured inline guards) for runtime mediation of its control flow and key operations, in compliance with security policies. The list and security annotations constitute a "proof", which is verified by the consumer (according to certain formats) after loading the code into the enclave and before the target binary is activated.

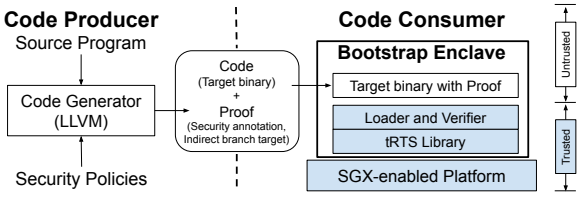

Fig. 2: System overview

Architecture. The architecture of is illustrated in Figure 2. The code generator and the binary and proof it produced are all considered untrusted. Only in the TCB is the code consumer with two components: a dynamic-loader operating a rewriter for re-locating the target binary, and a proof verifier running a disassembler for checking the correct instrumentation of security annotations. These components are all made public and can therefore be measured for a remote attestation (Section V-B). They are designed to minimize their code size, by moving most workload to the code producer.

We present the workflow of DEFLECTION in Figure 3. The target program (the service code) is first instrumented by the code producer, which runs a customized LLVM-based compiler (step 1). Then the target binary with the proof are delivered to the enclave. The code is first parsed (step 2) and then disassembled from the binary's entry along with its control flow traces. After that, the proof with the assembly is relocated and activated by the dynamic loader (step 3), further inspected by the verifier and if correct (step 4) before some immdiates being rewriten (step 5). Finally, after the bootstrap transfers the execution to the target program, the service begins and policies are checked at runtime.

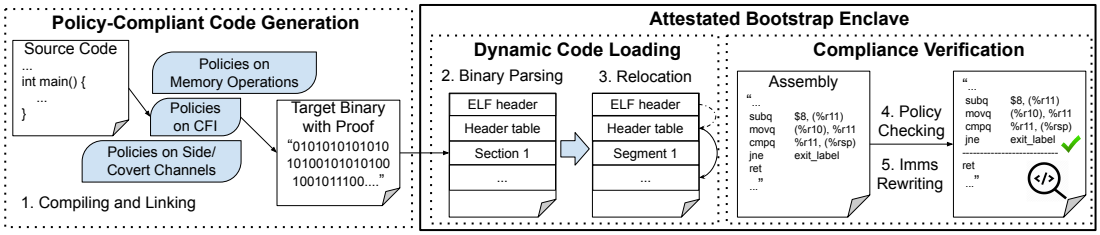

Fig. 3: Detailed framework and workflow

B. Security Policies

Without exposing its code for verification, the target binary needs to be inspected for compliance with security policies by the bootstrap enclave. These policies are meant to protect the privacy of sensitive data, to prevent its unauthorized disclosure. The current design supports following categories.

Enclave entry and exit control. DEFLECTION can mediate the content imported to or exported from the enclave, through the ECall and OCall interfaces, for the purposes of reducing the attack surface and controlling information leaks. Another objective here is to mitigate covert channel leaks through the interface between the enclave and the OS, making the attempt to covertly using users data to modulate events (e.g., system call arguments, I/O traffic statistics) hard to succeed.

• *P0: Input constraint, output encryption and entropy control*. We restrict the ECall interfaces to just serving the purposes of uploading data and code, which perform authentication, decryption and optionally input sanitization (or a simple length check). Also only some types of system calls are allowed through OCalls. Particularly, all network communication through OCalls should be encrypted with proper session keys (for the data owner or the code provider). In CCaaS, the data owner can demand that only one OCall (for sending back results to the owner) be allowed. Note that under such control and verification on explicit I/O channels, as well as system call traces and output data sizes, it becomes more difficult for untrusted code to utilize covert channels via software interfaces, such as system call arguments, for exfiltrating content of users input outside the enclave.

Memory leak control. Information leak can happen through unauthorized write to the memory outside the enclave, which should be prohibited through the code inspection.

• *P1: Preventing explicit out-enclave memory stores*. This policy prevents the target binary from explicit memory writes. It can be enforced by security annotations through mediation on the destination addresses of memory store instructions (such as `MOV`) to ensure that they are within the enclave address range `ELRANGE`).

• *P2: Preventing implicit out-enclave memory stores*. Illicit RSP register save/spill operations can also leak sensitive information to the out-enclave memory by pushing a register value to the address specified by the stack pointer, which is prohibited through inspecting the RSP content [40].

• *P3: Preventing unauthorized change to security-critical data within the bootstrap enclave*. This policy ensures that the security-critical data would never be tampered with by the untrusted code.

• *P4: Preventing runtime code modification*. Since the target code is untrusted and loaded into the enclave during its operation, under SGXv1, the code can only be relocated to the pages with `RWX` properties. So DEP protection should be in place to prevent the target binary from changing itself or uploading other code at runtime.

Control-flow management. To ensure that security annotations and other protection cannot be circumvented at runtime, the control flow of the target binary should not be manipulated. For this purpose, the following policy should be enforced:

• *P5: Preventing manipulation of indirect branches to violate policies P1 to P4*. This policy is to protect the integrity of the target binary's control flow, so security annotations cannot be bypassed. To this end, we need to mediate all indirect control transfer instructions, including indirect calls and jumps, and return instructions.

AEX based side/covert channel mitigation. In addition to the covert channel through software interfaces like system calls, we further studied the potential to mitigate the covert channel threat through SGX hardware interfaces. It is well known that SGX's user-land TEE design exposes a large side-channel surface, which cannot be easily eliminated. In the meantime, prior research focuses on the side-channel attacks causing Asynchronous Enclave Exits (AEXs). Examples include the controlled side channel attack [31] that relies on triggering page faults, and the attacks on L1/L2 caches [41], which requires context switches to schedule between the attack thread and the enclave thread, when Hyper-threading is turned off or a co-location test is performed before running the binary [42]. Such protection can be integrated into DEFLECTION to mitigate the side- or covert-channel attacks in this category, closing an important attack surface.

• *P6: Controlling the AEX frequency*. The policy requires the total number of the AEX concurrences to keep below a threshold during the whole computation. Once the AEX is found to be too frequent, above the threshold, the execution is terminated to prevent further information leak.

C. Policy-Compliant Code Generation

As mentioned earlier, the design of DEFLECTION is to move the workload from in-enclave verification to out-enclave generation of policy-compliant binary and its proof. In this section we describe the design of the code generator, particularly how it analyzes and instruments the target program so that security

policies (P1-P6, see Section IV-B) can be enforced during the program's runtime. Customized policies for purposes other than privacy can also be translated into proof and be enforced flexibly, e.g., to verify code logic and its functionalities.

Enforcing P1. The code generator is built on top of the LLVM compiler framework (Section V-A). When compiling the target program (in C) into binary, the code generator identifies (through the LLVM API `MachineInstr::mayStore()`) all memory storing operation instructions (e.g., `MOV`, Scale-Index-Base (SIB) instructions) and further inserts annotation code before each instruction to check its destination address and ensure that it does not write outside the enclave at runtime. The boundaries of the enclave address space can be obtained during dynamic code loading, which is provided by the loader (Section IV-D). The correct instrumentation of the annotation is later verified by the code consumer inside the enclave.

Enforcing P2. The generator locates all instructions that explicitly modify the stack pointer (the RSP in x86 arch) from the binary (e.g., a `MOV` changing its content) and inserts annotations to check the validity of the stack pointer after them. This protection, including the content of the annotations and their placement, is verified by the code consummer (Section IV-D). Note that RSP can also be changed implicitly, e.g., through pushing oversized objects onto the stack. This violation is prevented by the loader (Section IV-D), which adds guard pages (pages without permission) around the stack.

Enforcing P3. Similar to the enforcement of P1 and P2, the code generator inserts security annotations to prevent (both explicit and implicit) memory write operations on security-critical enclave data (e.g., SSA/TLS/TCS) once the untrusted code is loaded and verified.

Enforcing P4. To prevent the target binary from changing its own code at runtime, the code generator instruments all its write operations (as identified by the APIs `readsWritesVirtualRegister()` and `mayStore()`) with the annotations that disallow alternation of code pages. Note that the code of the target binary has to be placed on `RWX` pages by the loader under SGXv1 and its stack and heap are assigned to `RW` pages (see Sec. IV-D), so runtime code modification cannot be stopped solely by page-level protection (though code execution from the data region is defeated by the page permissions).

Enforcing P5. To control indirect calls or indirect jumps in the target program, the code generator extracts all labels from its binary during compilation and instruments security annotations before related instructions to ensure that only these labels can serve as legitimate jump targets. The locations of these labels should not allow an instrumented security annotations to be bypassed. Also to prevent the backward-edge control flow manipulation (through `RET`), the generator injects annotations after entry into and before return from every function call to operate on a shadow stack, which is allocated during code loading. Also all the legitimate labels are replaced by the loader when relocating the target binary. Such annotations are then inspected by the verifier when disassembling the binary to ensure that protection will not be circumvented by control-flow manipulation (Section IV-D).

Enforcing P6 with SSA inspection. When an exception or interrupt take place during enclave execution, an AEX is triggered by the hardware to save the enclave context (such as general registers) to the state saving area (SSA). This makes the occurrence of the AEX visible [42], [43]. Specifically, the code generator enforce the policy by instrumenting every basic block with an annotation that sets a marker in the SSA and monitors whether the marker is overwritten, which happens when the enclave context in the area has been changed, indicating that an AEX has occurred. Through counting the number of consecutive AEXes, the protected target binary can be aborted in the presence of anomalously frequent interrupts. This protection is verified by the code consumer before the binary is allowed to run inside the enclave.

Code loading support. Loading the binary is a procedure that links the binary to external libraries and relocates the code. For a self-contained function (i.e., one does not use external elements), compiling and sending the bytes of the assembled code is enough. However, if the function wants to use external elements but not supported inside an enclave (e.g., a system call), a distributed code loading support mechanism is needed. In our design, the loading procedure is divided into two parts, one (linking) outside and the other (relocation) inside the enclave. Our code generator assembles all the symbols of the entire code (including necessary libraries and dependencies) into one relocatable file via static linking. While linking all object files generated by the LLVM, it keeps all symbols and relocation information held in relocatable entries. The relocatable file, as above-mentioned target binary, is expected to be loaded for being relocated later (Section IV-D).

D. Configuration, Loading and Verification

With annotations instrumented and legitimate jump targets identified, the in-enclave workload undertaken by the bootstrap enclave side has been significantly reduced. Still, it needs to be properly configured to enforce the policy (P0) that cannot be implemented by the code generator. Following we elaborate how these critical operations are supported by our design.

Enclave configuration to enforce P0. To enforce the input constraint, we need to configure the enclave by defining certain public ECalls in Enclave Definition Language (EDL) files for data and code secure delivery. Note such a configuration, together with other security settings, can be attested to the remote data owner or code provider. The computation result of the in-enclave service is encrypted using a shared session key after the remote attestation and is sent out through a customized OCall. For this purpose, DEFLECTION only defines allowed system calls (e.g., `send/recv`) in the EDL file, together with their wrappers for security control (e.g., verifying the system call arguments). To support the basic CCaaS setting, `send` and `recv` need to be communicated to the data owner. Specially, the wrapper for `send` encrypts

the message to be delivered and pads it to a fixed length. For more complicated settings, we only permit a list of OCalls to be invoked once to deliver the computing result to the code provider, which can be enforced by the wrapper of the function. Further the wrapper can put a constraint on the length of the result to control the amount of information disclosed to the code provider: e.g., only 8 bits can be sent out.

Dynamic code loading and unloading. The target binary is delivered into the enclave as data through an ECall, processed by the wrapper placed by DEFLECTION, which authenticates the sender and then decrypts the code before handing it over to the dynamic loader. The primary task of the loader is to rebase all symbols of the binary according to its relocation information (Section IV-C). For this purpose, the loader first parses the binary to retrieve its relocation tables, then updates symbol offsets, and further reloads the symbols to designated addresses. During this loading procedure, the indirect branch label list is "translated" to in-enclave addresses, which are considered to be legitimate branch targets and later used for policy compliance verification.

As mentioned earlier (Section IV-C), the code section of the target binary is placed on pages with `RWX` privileges, since under SGXv1, the page permissions cannot be changed during an enclave's operation, while the data sessions (stack, heap) are assigned to the pages with `RW` privileges. These code pages for the binary are guarded against any write operation by the annotations for enforcing P4. Other enclave code, including that of the code consumer, is under the `RX` protection through enclave configuration. Further the loader assigns two non-writable blank guard pages right before and after the target binary's stack for enforcing P2, and also reserves pages for hosting the list of legitimate branch targets and the shadow stack for enforcing P5.

Just-enough disassembling and verification. After loading and relocating, the target binary is passed to the verifier for a policy compliance check. Such a verification is meant to be highly efficient, together with a lightweight disassembler. Specifically, our disassembler is designed to leverage the assistance provided by the code generator. It starts from the program entry discovered by the parser and follows its control flow until an indirect control flow transfer, such as indirect jump or call, is encountered. Then, it utilizes all the legitimate target addresses on the list to continue the disassembly and control-flow inspection. In this way, the whole program will be quickly and comprehensively examined.

For each indirect branch, the verifier checks the annotation code (Figure IV-C) right before the branch operation, which ensures that the target is always on the list at runtime. Also, these target addresses, together with direct branch targets, are compared with all guarded operations in the code to detect any attempt to evade security annotations. To simplify the verification of the CFI policy compliance, the verifier utilizes *hints* (i.e., the symbol name on the list) to identify the set of possible targets for calls/jumps. For this purpose, the verifier scans the machine code to ensure that these identifiers appear only at the beginning of basic blocks. The verification of P6 for covert channel mitigation is done one basic block at a time, and on the basic-block exit the verifier checks whether all policy-compliance instrumentations are in position at the entries to all possible successor blocks.

With such verification, no hidden control transfers will be performed by the binary, allowing further inspection of other instrumented annotations. These annotations are expected to be well formatted and located around the critical operations as described in Section IV-C. More details are given in Section V-A and our Github repository [24].

V. IMPLEMENTATION

We implemented the prototype on Linux/x86 arch. Specifically, we implemented the code generator with LLVM 9.0.0, and built other parts on an SGX environment. The LLVM passes consist of several types of instrumentations for the code generator. Besides, we implemented the bootstrap enclave based on Capstone [23] as the disassembler.

A. Multi-level Instrumentation

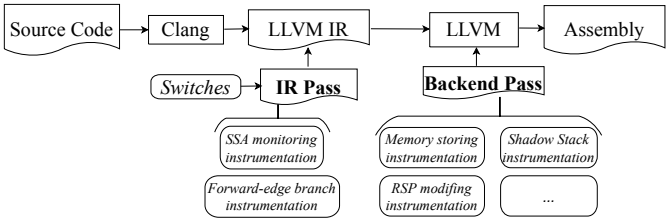

Fig. 4: Workflow of flexible code generation

The code generator we built is mainly based on LLVM (Fig. 4), and the assembly-level instrumentation is the core module. To address the challenge of limited computing resources described in Section III-B, this code generator tool is designed and implemented comprehensively, to make the policy verifier small and exquisite. More specifically, we implemented modules for checking memory writing instructions, RSP modification, indirect branches and for building shadow stack. And we reformed an instrumentation module to generate side-channel-resilient annotations. To support flexible control of code to comply with different security policies, we implanted a set of switches into our code generator. These switches work on the IR level and their on/off states can be passed down to the target code level for further control, depending on the policies to be enforced. This *separating mechanism and policy* design makes that we can not only demonstrate the security policies for several real-world scenarios, modules of the annotation generation for customized functionalities can also be integrated into the code generator.

Here is an example (Figure 5). The main function of the module for checking explicit memory write instructions (P1) is to insert annotations before them. Suppose there is such a memory write instruction in the target program, '`mov reg, [reg+imm]`', the structured annotation first sets the upper and lower bounds as two temporary Imms (0x3fffffffffff and 0x4fffffffffff), and then compares the address of the

```
pushq   %rbx     ;save execution status
pushq   %rax
leaq    [reg+imm], %rax ;load the operand
movq    $0x3FFFFFFFFFFFFFFF, %rbx   ;set bounds
cmpq    %rbx, %rax
ja      exit_label
movq    $0x4FFFFFFFFFFFFFFF, %rbx   ;set bounds
cmpq    %rbx, %rax
jb      exit_label
popq    %rax
popq    %rbx
movq    reg, [reg+imm]
```

Fig. 5: Store instruction instrumentation

destination operand with the bounds. The real upper/lower bounds of the memory write instruction are specified by the loader later. If our instrumentation finds the memory write instruction trying to write data to illegal space, it will cause the program to exit at runtime.

B. Building Bootstrap Enclave

Following the design in Section IV-D, we implemented a *Dynamic Loading after RA mechanism* for the bootstrap enclave. The enclave is initiated based upon a configuration file (a.k.a. the manifest file), which specifies the system calls the enclave is allowed to make in compliance with security policies, the protection enforced through instrumented OCall stubs. During the whole service, the data owner can only see the attestation messages related to the bootstrap's enclave quote, but learn nothing about service providers code.

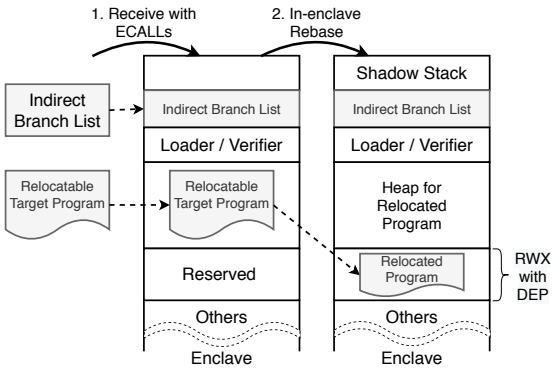

Fig. 6: Detailed workflow of the dynamic loader

Remote attestation. Once the bootstrap enclave is initiated, it needs to be attested. We leverage the RA-TLS routine [44] and adjust it to our implementation. The RA procedures are invoked inside the bootstrap enclave after secret provision between parties. After obtaining a quote of the bootstrap enclave, the remote data owner submit the quote to IAS and obtain an attestation report.

Dynamic loader. When the RA is finished, the trust between data owner and the bootstrap enclave is established. Then the user can locally/remotely call Ecall (`ecall_receive_binary`) to load the service binary instrumented with security annotations and the indirect branch list without knowing the code. User data is loaded from untrusted memory into the trusted enclave memory when the user remotely calls Ecall (`ecall_receive_userdata`), to copy the data to the section reserved for it.

Then, the dynamic loader in the bootstrap enclave loads and relocates the generated code. The indirect branch list, which is comprised of symbol names that will be checked in indirect branch instrumentations, will be resolved at the very beginning. Our implementation utilizes 4M memory (by default) space for storing indirect branch targets, and for operating a shadow stack. We further reserve more memory space for hosting the uploaded binary and data, which can be configured in the manifest file. After relocation, the detailed memory layout and some key steps are shown in Figure 6.

Policy verifier. The policy-compliance verifier, is composed with three components - a clipped disassembler, a verifier, and an immediate operand rewritter.

• *Clipped disassembler.* We enforce each policy at assembly level. Thus, we incorporate a lightweight disassembler inside the enclave. To implement it, we remove unused components of this existing wide-used framework, and use Recursive Descent Disassembly to traverse the code. Also, we use the *diet* mode, making the engine size at least 40% smaller [45].

• *Policy verifier.* The verifier and the following rewriter do the work just right after the target binary is disassembled, according the structured guard formats provided by our code generator. The verifier uses a simple scanning algorithm to ensure the policies applied in assembly language instrumentation. Specifically, the verifier scans the whole assembly recursively along with the disassembler. It follows the clipped disassembler to scan instrumentations before/after certain instructions are in place, and checks if there is any branch target pointing between instructions in those instrumentations.

• *Imm rewriter.* One last but not least step before executing the target binary code is to resolve and replace the Imm operands in instrumentations, including the base of the shadow stack, and the addresses of indirect branch targets (i.e. legal jump addresses). For example, the genuine base address of shadow stack is the start address `__ss_start` of the memory space reserved by the bootstrap enclave for the shadow stack. And the ranges are determined using functions of Intel SGX SDK during dynamic loading (Section IV-D). Table I shows what the specific values should be before and after rewriting, respectively. The first column of table I shows the target we need to rewrite while loading. For instance, the upper bound address of data section would be decided during loading, but it would be 0x3fffffffffffffff (shown in the 2nd. column) during the proof generation and will be modified to the real upper data bound address. The third column shows the variable name used in our prototype.

VI. Analysis and Evaluation

A. Security Analysis

TCB analysis. The hardware TCB of DEFLECTION includes the TEE-enabled platform, i.e. the SGX hardware. The software TCB includes the components shown in Table II. The loader we implemented consists of less than 600 lines of code

TABLE I: Fields to be rewritten

Target imm description	**From** (hex)	**To**
Upper bound of .data	3fffffffffffffff	`upper_data_bound`
Lower bound of .data	4fffffffffffffff	`lower_data_bound`
Upper bound of stack	5fffffffffffffff	`upper_stack_bound`
Lower bound of stack	6fffffffffffffff	`lower_stack_bound`
Upper bound of .code	7fffffffffffffff	`lower_code_bound`
Lower bound of .code	8fffffffffffffff	`lower_code_bound`
# of indirect branch targets	1fffffff	`branch_target_idx`
Addr. of branch target list	1ffffffffffffff	`__branch_target`
Addr. of the shadow stack	2ffffffffffffff	`__ss_start`

(LoCs) and the verifier includes less than 700 LoCs, also integrating the SGX SDK and part of Capstone libraries. The shields of SCONE and Graphene-SGX increase the code size further. Nevertheless, the binary sizes of shielded applications increase to 5 times compared to ours. Currently, Occlum has not integrated the SFI feature in its latest version [46], thus we can only know the lower bound of its TCB size. Altogether, our software TCB contains a self-contained enclave binary (1.9 MB) with a shim libc (2.6 MB). By comparison, most solutions are at least an order of magnitude larger as compared to DEFLECTION.

TABLE II: TCB comparison with other solutions

Shielding runtimes	**Core components**	kLoCs	**Size(MB)**
Ryoan	Eglibc NaCl sandbox Naclports	892 216 460	> 19
SCONE	OS Shield and shim libc	187	> 16
Graphene-SGX	Glibc LibPAL Graphene LibOS	1200 22 34	58.5
Occlum	Occlum shim libc Verifier Occlum LibOS and PAL	93 N/A 24.5	> 8.6
DEFLECTION	Interface and libc Loader/Verifier RA/Encryption	65 1.3 0.2	3.5

Policy analysis. Here we show how the policies on the untrusted code, once enforced, prevent information leaks from the enclaves. In addition to side channels, there are two possible ways for a data operation to go across the enclave boundaries: bridge functions [14] and memory write.

• *Bridge functions*. With the enforcement of P0, the loaded code can only invoke our OCall stubs, which prevents the leak of plaintext data through encryption and controls the amount of information that can be sent out.

• *Memory write operations*. All memory writes, both direct memory store and indirect register spill, are detected and blocked. Additionally, software DEP is deployed so the code cannot change itself. Also the control-flow integrity (CFI) policy, P5, prevents the attacker from bypassing the checker with carefully constructed gadgets by limiting the control flow to only legitimate target addresses.

As such, possible ways of information leak to the outside of the enclave are controlled. As proved by previous works [21], [47] the above-mentioned policies (P1-P5) guarantee the property of confidentiality. Furthermore the policy (P5) of *protecting return addresses and indirect control flow transfer, together with preventing writes to outside* has been proved to be adequate to construct the confinement [47], [48]. So, enforcement of the whole set of policies from P0 to P5 is sound and complete in preventing explicit information leaks. In the meantime, our current design is limited in side-channel protection. We can mitigate the threats of page-fault based attacks and exploits on L1/L2 cache once Hyper-threading is turned off or HyperRace [42] is incorporated (P6). However, defeating the attacks without triggering interrupts, such as inference through LLC is left for future research.

With such protection, still our design cannot eliminate all covert channels, which is known to be hard. However, it is important to note that other SGX runtimes, including SCONE, Graphene-SGX, Occlum, provide no such protection at all. An exception is Ryoan, which pads its enclave output to the same size, as we do. However, it does not handle the leak from the hardware-based channels.

B. Performance Evaluation

Here we discuss performance overhead of different level protections DEFLECTION can provide. These settings include just explicit memory write check (P1), both explicit memory write check and implicit stack write check (P1+P2), all memory write and indirect branch check (P1-P5), and together with side channel mitigation (P1-P6).

Testbed setup. In our research, we evaluated the performance of our prototype and tested its code generation and code execution. All experiments were conducted on Ubuntu 18.04 (Linux kernel version 4.4) with SGX SDK 2.5 installed on Intel Xeon CPU E3-1280 with 64GB memory. Also we utilized GCC 5.4 to build the bootstrap enclave and the SGX application, and the parameters '-fPIC', '-fno-asynchronous-unwind-tables', '-fno-addrsig', and '-mstackrealign' to generate x86 binaries.

TABLE III: Performance overhead on nBench

Program Name	**P1**	**P1+P2**	**P1-P5**	**P1-P6**
NUMERIC SORT	+5.18%	+6.05%	+6.79%	+12.0%
STRING SORT	+8.05%	+10.2%	+12.4%	+18.4%
BITFIELD	+6.11%	+11.3%	+15.5%	+17.9%
FP EMULATION	+0.20%	+0.27%	+0.33%	+5.36%
FOURIER	+2.48%	+2.72%	+2.89%	+7.45%
ASSIGNMENT	+6.73%	+15.6%	+25.0%	+39.8%
IDEA	+2.34%	+2.66%	+3.13%	+12.1%
HUFFMAN	+15.5%	+16.6%	+18.1%	+21.3%
NEURAL NET	+13.8%	+19.4%	+20.2%	+23.1%
LU DECOMPOSITION	+4.30%	+7.03%	+9.67%	+22.6%

Performance on nBench. We instrumented all applications in the SGX-nBench [49], and ran each testcase of the nBench suites under a few settings, each for 10 times. Table III shows the average execution time under different settings. Without side channel mitigation (P1-P5), our prototype introduces an 0.3% to 25% overhead (on FP-emulation). Apparently, the store instruction instrumentation alone (P1) does not cause a large performance overhead, with largest being 6.7%. Also, when P1 and P2 are applied together, the overhead just becomes slightly higher than P1 is enforced alone. Besides, almost all benchmarks in nBench perform well under the CFI

check P5 (less than 3%) except for the benchmarks Bitfield (about 4%) and the Assignment (about 10% due to its frequent memory access pattern).

Performance on real-world applications. We further evaluated our prototype on various real-world applications, including personal health data analysis, personal financial data analysis, and Web servers. We implemented those macro-benchmarks and measured the differences between their baseline performance (without instrumentation) in enclave and the performance of our prototype.

• *Sensitive genome data analysis*. We implemented the NeedlemanWunsch algorithm [50] that aligns two human genomic sequences in the FASTA format [51] taken from the 1000 Genomes project [52]. The algorithm uses dynamic programming to compute recursively a two dimensional matrix of similarity scores between subsequences; as a result, it takes N^2 memory space where N is the length of the two input sequences. We measured the sequence alignment program execution time under the aforementioned settings. Figure 7 shows the performance of the sequence alignment algorithm with different input lengths (x-axis). The overall overhead (including all kinds of instrumentations) is no more than 20% (with the P1 alone no more than 10%), when input size is small (less than 200 Bytes). When input size is greater than 500 Bytes, the overhead of P1+P2 is about 19.7% while P1-P5 spends 22.2% more time than the baseline.

For sequence generation, Figure 8 shows the performance when the output size (x-axis) varies from 1K to 500K nucleotides. Enforcing P1 alone results in 5.1% and 6.9% overheads when 1K and 100K are set as the output lengths. When the output size is 200K, our prototype yields less than 20% overhead. Even when the side channel mitigation is applied, the overhead becomes just 25%. With the increase of processing data size, the overhead of the system also escalates; however, the overall performance remains acceptable.

• *Personal credit score analysis*. In our study, we implemented a BP neural network-based credit scoring algorithm [53] that calculates user's credit scores. The model was trained on 10000 records and then used to make prediction (i.e., output a confidence probability) on different test cases. As shown in Figure 9, on 1000 and 10000 records, enforcement of P1-P5 would yields around 15% overhead. While processing more than 50000 records, the overhead of the full check does not exceed 20%. The overhead of P1-P6 does not exceed 10% when processing 100K records.

• *HTTPS server*. We built an HTTPS server in enclave using the mbed TLS library [54]. A client executes a stress test tool - Siege [55] - on another host in an isolated LAN. Siege was configured to send continuous HTTPS requests (with no delay between two consecutive ones) to the web server for 10 minutes. We measured its performance in the presence of different concurrent connections to understand how our instrumented HTTPS server implementation would perform.

Figure 10 shows the response times and throughput when all policies are applied to the HTTPS server benchmark. When the concurrent connections are less than 75, the instrumented HTTPS server has similar performance of the in-enclave https server without instrumentation. When the concurrency increases to 100, the performance goes down to some extent. While after the concurrency increases to 150, the response time of instrumented server goes up significantly. On average, enforcing P1-P6 results in 14.1% overhead in the response time. As for throughput, when the number of the concurrent connections is between 75 and 200, the overhead is less than 10%. These experiments on realistic workloads show that all policies, including side-channel mitigation, can be enforced at only reasonable cost.

Performance comparison on HTTPS server. Here we compare the performance overheads induced by existing shielding runtimes with our solution. Since Occlum has not integrated the SFI feature in its latest version [46] and Graphene-SGX does not support our security policies, we cannot get their performance details to compare against ours when policy-enforcing instrumentations are added. In our study, we ran an HTTPS server within those runtimes. As expected, their performance is affected by the workload, sizes of files requested from the server. As shown in Figure 11, unprotected Graphene-SGX has the best transfer rate with relatively small files. However, with the size growing, DEFLECTION outperforms both runtimes (77% of running the server on the native Linux), even when our approach implements security policies (P0-P5) while these runtimes do not.

VII. DISCUSSION

Supporting other side/covert channel defenses. The framework of our system is highly flexible, which means assembling new policies into current design can be very straightforward. In Section IV-C, we talked about policy enforcement approaches for side channel resilience. It demonstrated that our framework can take various side channel mitigation approaches to generate code carried with proof. Besides AEX based mitigations which we learnt from Hyperrace [42], others [43], [56]–[61] can also be transformed and incorporated into the design, specifically for mitigating cache timing, memory bus timing [62], and other timing channels. ORAM [63], [64] can also be integrated to DEFLECTION as a policy, to relieve memory access based side- or covert- channel leakage to some extend. Additionally, policies such as *on-demand aligning/blurring processing time* can be added for preventing processing-time based covert channels [65]. Even though new attacks have been kept being proposed and there is perhaps no definitive and practical solutions to all side/covert channel attacks, we believe eventually some efforts can be integrated in our work, even using SGXv2 [66].

Supporting multi-threading. SGX supports multi-threaded execution. To concurrently service many clients, policies such as isolating each thread's private memory and setting read-only permissions on cross-thread shared memory can be enforced. Multi-threading could introduce serious bugs [67]. The proof enforcement of CFI may suffer from a time of check to time of

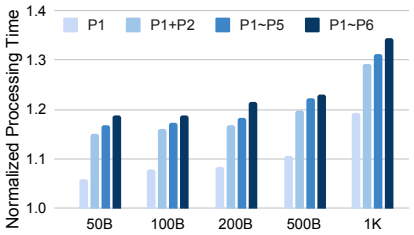

Fig. 7: Sequence alignment

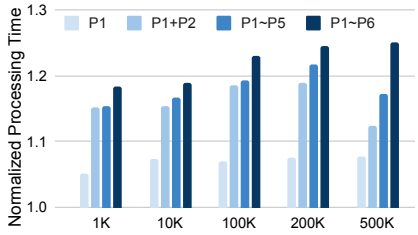

Fig. 8: Sequence generation

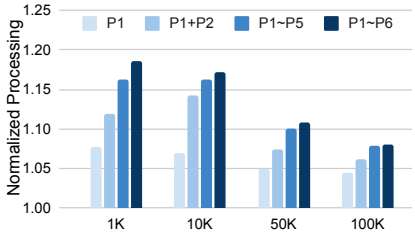

Fig. 9: Credit scoring

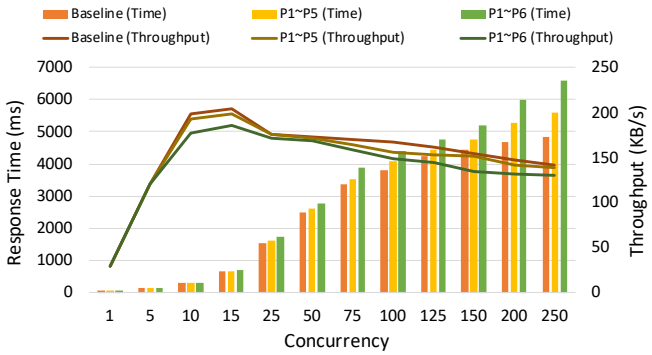

Fig. 10: Performance on HTTPS server

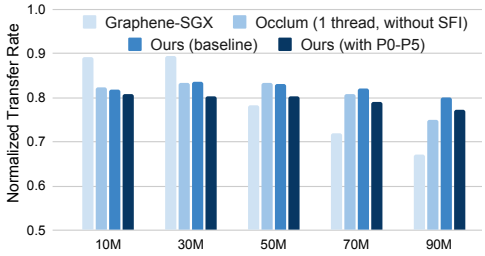

Fig. 11: Performance comparison

use (TOCTOU) problem [68]. To cope with that, we can make all CFI metadata to be kept in the register or hardware [69] instead of in memory, and guarantee that the instrumented proof could not be modified by any threads [70].

VIII. Related Work

Secure computing using SGX. Many existing works propose using SGX to secure cloud computing systems, e.g., VC3 [48], TVM [71], by using sand-boxing, containers [72], and others [73], [74]. In-enclave JVM interpreter is also a good choice [75]. These systems protect the enclave on untrusted platform, as a result, they either do not protect the code privacy or they consider a one-party scenario, i.e., the code and data needed for the computation are from the same participant.

Data confinement with SFI. Most related to our work are data confinement technologies, which confines untrusted code with confidentiality and integrity guarantees. Ryoan [10] and its follow-up work [76] provide an distributed sand-box by porting NaCl to the enclave environment, confining untrusted data-processing modules to prevent leakage of the users input data. However the overhead of Ryoan turns out huge (e.g., 100% on genes data) and was evaluated on an software emulator for supporting SGXv2 instructions. XFI [77] is the most representative unconventional PCC work based on SFI, which places a verifier at OS level, instead of a TEE. Our compiler-based generator is more efficient in providing forward-edge CFI and our runtime enforcement is simpler than inline reference monitor or dynamic binary translation used by traditional SFI [78], [79]. Occlum [8] is a design of SGX-based library OS, enforcing in-enclave task isolation with MPX-based multi-domain SFI scheme. Side channels pose serious threats [29], [41], [80], [81], while existing work assumes side channels as an orthogonal research topic [8], [21], [82]. Our framework is designed with side channels in mind and it can flexibly support integration of instrumentation based side channel defenses [42], [58], [83]–[85].

Code privacy. Code secrecy is an easy to be ignored but very important issue [86]. TEEshift [13], DynSGX [87] and SGXElide [88] both make possible that developers execute their code privately in public cloud environments, enabling developers to better manage the scarce memory resources. However, they only care about the developer's privacy but ignore the confidentiality of data belonging to users.

Confidentiality verification of enclave programs. With formal verification tools, Moat [21] and its follow-up works [47] verify if an enclave program has the risk of data leakage. The major focus of them is to verify the confidentiality of an SGX application outside the enclave formally and independently. Although it is possible that the verification could be performed within a "bootstrap enclave", the TCB would include the IR level language (BoogiePL) interpreter [89] and a theorem prover [34]. Moreover, neither of them can discharge the large overhead introduced by instruction modeling and assertion proving when large-scale real-world programs are verified.

IX. Conclusion

In this paper we proposed Deflection, which allows the user to verify the code provided by untrusted parties without undermining their privacy and integrity. Meanwhile, we instantiated the design of a code generator and a code consumer (the bootstrap enclave) - a lightweight PCC-type framework. Due to the differences between normal binary and SGX binary, we retrofit the framework to be fitted into SGX. In return, we reduce the framework's TCB as small as possible. Our work does not use formal certificate to validate the loaded private binary, but leverage data/control flow analysis to fulfill the goal of verifying if a binary has such data leakage, allowing our solution to scale to real-world software.

REFERENCES

[1] 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." Hasp, isca, vol. 10, no. 1, 2013.

[2] M. Russinovich, "Introducing Azure confidential computing," Seattle, WA: Microsoft, 2017.

[3] "Google. Asylo," 2019. [Online]. Available: https://asylo.dev/

[4] "Confidential Computing Consortium," 2019. [Online]. Available: https://confidentialcomputing.io

[5] Z. Zhang, X. Ding, G. Tsudik, J. Cui, and Z. Li, "Presence attestation: The missing link in dynamic trust bootstrapping," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 89–102.

[6] S. Chakrabarti, M. Hoekstra, D. Kuvaiskii, and M. Vij, "Scaling intel® software guard extensions applications with intel® sgx card," in Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, 2019, pp. 1–9.

[7] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "Sgx-lkl: Securing the host os interface for trusted execution," arXiv preprint arXiv:1908.11143, 2019.

[8] 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 Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 955–970.

[9] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell et al., "{SCONE}: Secure linux containers with intel {SGX}," in 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), 2016, pp. 689–703.

[10] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," ACM Transactions on Computer Systems (TOCS), vol. 35, no. 4, p. 13, 2018.

[11] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards memory safe enclave programming with rust-sgx," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 2333–2350.

[12] 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 Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, 2019, pp. 114–121.

[13] T. Lazard, J. Götzfried, T. Müller, G. Santinelli, and V. Lefebvre, "Teeshift: protecting code confidentiality by selectively shifting functions into tees," in Proceedings of the 3rd Workshop on System Software for Trusted Execution, 2018, pp. 14–19.

[14] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 1741–1758.

[15] G. C. Necula, "Proof-carrying code," in Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1997, pp. 106–119.

[16] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," in Informatics. Springer, 2001, pp. 86–101.

[17] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline, "A certifying compiler for java," in ACM SIGPLAN Notices, vol. 35, no. 5. ACM, 2000, pp. 95–107.

[18] X. Leroy, "Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant," in ACM SIGPLAN Notices, vol. 41, no. 1. ACM, 2006, pp. 42–54.

[19] H. Pirzadeh, D. Dubé, and A. Hamou-Lhadj, "An extended proof-carrying code framework for security enforcement," in Transactions on computational science XI. Springer, 2010, pp. 249–269.

[20] G. C. Necula and S. P. Rahul, "Oracle-based checking of untrusted software," in ACM SIGPLAN Notices, vol. 36, no. 3. ACM, 2001, pp. 142–154.

[21] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 1169–1184.

[22] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system f to typed assembly language," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 21, no. 3, pp. 527–568, 1999.

[23] "Capstone - The Ultimate Disassembler," http://www.capstone-engine.org/.

[24] "Cat-sgx," https://github.com/SecInTheShell/cat-sgx.

[25] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 523–539.

[26] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against intel {SGX}," in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 1213–1227.

[27] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with intel sgx," in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2019, pp. 177–196.

[28] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2017, pp. 3–24.

[29] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing," in 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 557–574.

[30] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 955–972.

[31] 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. IEEE, 2015, pp. 640–656.

[32] P. V. Homeier and D. F. Martin, "A mechanically verified verification condition generator," The Computer Journal, vol. 38, no. 2, pp. 131–141, 1995.

[33] L. C. Paulson, "Isabelle: The next 700 theorem provers," arXiv preprint cs/9301106, 2000.

[34] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[35] Y. Bertot and P. Castéran, Interactive theorem proving and program development: CoqArt: the calculus of inductive constructions. Springer Science & Business Media, 2013.

[36] A. W. Appel, N. Michael, A. Stump, and R. Virga, "A Trustworthy Proof Checker," Journal of Automated Reasoning, vol. 31, no. 3-4, pp. 231–260, 2003.

[37] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in International Conference on Computer Aided Verification. Springer, 2011, pp. 463–469.

[38] A. W. Appel, "Foundational proof-carrying code," in Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. IEEE, 2001, pp. 247–256.

[39] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008, 2008, pp. 315–328.

[40] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To detect stack buffer overflow with polymorphic canaries," in 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2018, pp. 243–254.

[41] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 2421–2434.

[42] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018, pp. 178–194.

[43] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 217–233.

[44] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij, "Integrating remote attestation with transport layer security," arXiv preprint arXiv:1801.05863, 2018.

[45] N. A. Quynh, "Capstone: Next-gen disassembly framework," Black Hat USA, 2014.

[46] "Occlum," https://github.com/occlum/occlum.

[47] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A Design and Verification Methodology for Secure Isolated Regions," in ACM SIGPLAN Notices, vol. 51, no. 6. ACM, 2016, pp. 665–681.

[48] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 38–54.

[49] "Sgx nbench," https://github.com/utds3lab/sgx-nbench.

[50] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," Journal of molecular biology, vol. 48, no. 3, pp. 443–453, 1970.

[51] "Fasta format," https://en.wikipedia.org/wiki/FASTA_format.

[52] "1000 genomes project," https://en.wikipedia.org/wiki/1000_Genomes_Project.

[53] H. L. Jensen, "Using neural networks for credit scoring," Managerial finance, vol. 18, no. 6, pp. 15–26, 1992.

[54] "mbed tls," https://tls.mbed.org/.

[55] "Siege," https://www.joedog.org/siege-home/.

[56] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," ACM Transactions on Information and System Security (TISSEC), vol. 18, no. 1, pp. 1–32, 2015.

[57] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in 25th {USENIX} Security Symposium ({USENIX} Security 16), 2016, pp. 53–70.

[58] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in NDSS, 2017.

[59] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2018, pp. 15–26.

[60] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in 28th {USENIX} Security Symposium ({USENIX} Security 19), 2019, pp. 657–674.

[61] M. Orenbach, Y. Michalevsky, C. Fetzer, and M. Silberstein, "Cosmix: a compiler-based system for secure memory instrumentation and execution in enclaves," in 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19), 2019, pp. 555–570.

[62] F. Liu, H. Wu, and R. B. Lee, "Can randomized mapping secure instruction caches from side-channel attacks?" in Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, 2015, pp. 1–8.

[63] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx." IACR Cryptol. ePrint Arch., vol. 2017, p. 549, 2017.

[64] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on intel sgx." in NDSS, 2019.

[65] W. Liu, D. Gao, and M. K. Reiter, "On-demand time blurring to support side-channel defense," in European Symposium on Research in Computer Security. Springer, 2017, pp. 210–228.

[66] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: closing controlled channels with self-paging enclaves," in Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–16.

[67] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in European Symposium on Research in Computer Security. Springer, 2016, pp. 440–457.

[68] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "{CONFIRM}: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in 28th {USENIX} Security Symposium ({USENIX} Security 19), 2019, pp. 1805–1821.

[69] C. DeLozier, K. Lakshminarayanan, G. Pokam, and J. Devietti, "Hurdle: Securing jump instructions against code reuse attacks," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 653–666.

[70] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019, pp. 985–999.

[71] N. Hynes, R. Cheng, and D. Song, "Efficient deep learning on multi-source private data," arXiv preprint arXiv:1807.06689, 2018.

[72] D. Tian, J. I. Choi, G. Hernandez, P. Traynor, and K. R. Butler, "A practical intel sgx setting for linux containers in the cloud," in Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, 2019, pp. 255–266.

[73] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves." in NDSS, 2017.

[74] K. Shanker, A. Joseph, and V. Ganapathy, "An evaluation of methods to port legacy code to sgx enclaves," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1077–1088.

[75] J. Jiang, X. Chen, T. Li, C. Wang, T. Shen, S. Zhao, H. Cui, C.-L. Wang, and F. Zhang, "Uranus: Simple, efficient sgx programming and its applications," in Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, 2020, pp. 826–840.

[76] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," arXiv preprint arXiv:1803.05961, 2018.

[77] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006, pp. 75–88.

[78] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 558–569.

[79] G. Tan et al., Principles and implementation techniques of software-based fault isolation. Now Publishers, 2017.

[80] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution," in 27th {USENIX} Security Symposium ({USENIX} Security 18), 2018, pp. 991–1008.

[81] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2019, pp. 142–157.

[82] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A Formal Foundation for Secure Remote Execution of Enclaves," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017, pp. 2435–2450.

[83] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, 2016, pp. 317–328.

[84] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting {SGX} enclaves from practical side-channel attacks," in 2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18), 2018, pp. 227–240.

[85] R. Sinha, S. Rajamani, and S. A. Seshia, "A compiler and verifier for page access oblivious computation," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 649–660.

[86] K. A. Küçük, D. Grawrock, and A. Martin, "Managing confidentiality leaks through private algorithms on software guard extensions (sgx) enclaves," EURASIP Journal on Information Security, vol. 2019, no. 1, p. 14, 2019.

[87] R. Silva, P. Barbosa, and A. Brito, "DynSGX: A Privacy Preserving Toolset for Dinamically Loading Functions into Intel (R) SGX Enclaves," in 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2017, pp. 314–321.

[88] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "SGXElide: enabling enclave code secrecy via self-modification," in Proceedings of the 2018 International Symposium on Code Generation and Optimization. ACM, 2018, pp. 75–86.

[89] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in International Symposium on Formal Methods for Components and Objects. Springer, 2005, pp. 364–387.