Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

HHS Public Access

Author manuscript

Proc (Int Conf Dependable Syst Netw). Author manuscript; available in PMC 2022 August 01.

Published in final edited form as:

Proc (Int Conf Dependable Syst Netw). 2021 June ; 2021: 413–425. doi:10.1109/dsn48987.2021.00052.

Practical and Efficient in-Enclave Verification of Privacy Compliance

Weijie Liu[*], **Wenhao Wang**[†,§], **Hongbo Chen**[*], **XiaoFeng Wang**[*,§], **Yaosong Lu**[†], **Kai Chen**[†], **Xinyu Wang**[‡], **Qintao Shen**[†], **Yi Chen**[¶], **Haixu Tang**[*]

[*]Indiana University Bloomington

[†]Institute of Information Engineering, CAS

[‡]Shanghai Jiao Tong University

[¶]The Chinese University of Hong Kong

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. 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.

Index Terms—

Intel SGX; Confidential Computing; Proof-Carrying Code; Enclave Shielding Runtime

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

[§]Corresponding authors: wangwenhao@iie.ac.cn, xw7@iu.edu.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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, a pharmaceutical company can run its proprietary algorithm inside an enclave hosting patient medical records, without exposing the algorithm but can still ensure the compliance of data use with the hospital's privacy policy. Another example can be a privacy-preserving credit evaluation service, in which a customer's transactions are only exposed to an enclave running the credit evaluation code in compliance with a set of public privacy-protection rules (such as GDPR). We consider confidential computing as a service (*CCaaS*) as a privacy extension of today's online data processing services like machine-learning as a service [2]. CCaaS is hosted by the party that operates its own target binary on the data provided by remote users. 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. Although the shielding runtimes such as Library OSes [6], [7], SCONE container [8], Ryoan sandbox [9] and the interpreters/compilers built for SGX

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

[10], [11] 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 [12], which introduces performance overhead. More importantly, the confinement mechanisms (sometimes including a whole interpreter) significantly increase the TCB, leading to a challenge in ensuring its security [13].

A promising direction we envision that could lead to a practical solution is *proof-carrying code* (*PCC*) [14], [15], a technique that enables a *verification condition generator* (*VCGen*) [16]–[18] 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 [18], [19]) 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 [20].

Our solution.

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 Software-based Fault Isolation (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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

further work, other proofs could be extended given a formal model of the x64 instruction set (e.g., as in [21]). 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 [22] 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 shielding runtimes. 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 [23].

II. Background

Intel SGX.

Intel SGX [1] is a user-space TEE, which is characterized by flexible process-level isolation. 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 virtual memory support is available, it incurs significant overheads in paging [8].

Another problem caused by SGX's design is a large attack surface. The application can invoke a pre-defined function inside the enclave, passing input parameters and pointers to shared memory within the application. Those invocations from the application to the enclave are called ECall. When an enclave executes, it can perform an OCall to a predefined function in the application. Contrary to an ECall, an OCall cannot share enclave memory with the application, so it must copy the parameters into the application memory before the OCall. When an enclave program contains memory vulnerabilities, attacks can happen to compromise enclave's privacy protection. Prior research demonstrates that a Return-oriented programming (ROP) attack can succeed in injecting malicious code inside an enclave, which can be launched by the OS, Hypervisor, or BIOS [24]–[26]. Another security risk is side-channel leak [27]–[29], 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., [30]).

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. Traditional PCC schemes tend to utilize formal verification for proof generation and validation. Techniques for this purpose includes verification condition generation [16], [31], theorem proving [32]–[34], and proof checking [35], 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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

automatically transform a binary to IL for in-enclave verification. BAP [36] 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 [19]. Particularly, today's VCGens are built on either an interpreter/compiler even a virtual machine [17], and therefore will lead to a huge TCB. Prior attempts [37] 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 to be certified [14]. Although techniques are there to reduce the proof size [18], [37], they are too complicated to scale to real-world applications [35].

III. Deflection

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. 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, 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 proprietary programs to protect its intellectual property. Here, DEFLECTION can enforce the data privacy on behalf of the data-processing services. Besides, the framework of our system is highly flexible, which means assembling new policies into current design can be very straightforward. Different on-demand policies can be appended/withdrawn to serve various goals. For example, DEFLECTION can make the quick patch possible on software level, like the way people coping with 1-day vulnerabilities - emergency quick fix.

A. The Delegation Model

Attestation service.—Attestation Service (AS) 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. It further 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.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Data owner.—The data owner uploads sensitive data to use in-enclave services and intends to keep her data secret during the computation.

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.

Key agreement procedure.—Both parties, a service provider (code provider) and a remote user (data owner), inspect and agree on the implementation details of the bootstrap enclave. The data owner and the code provider can both attest that the bootstrap loader is correctly running on SGX platforms using remote/local attestation. In particular, data owner and code provider both generate a measurement of the bootstrap enclave, which acts as the trust anchor of their agreement and is required for verification during SGX remote/local attestation.

After the two attestations are done and shared session keys are negotiated by Diffie–Hellman key exchange, all messages can be transferred through the two trusted channels. Since the data owner already knows the measurement/hash of the service code. The bootstrap enclave first extracts and verifies the measurement/hash of the service code, and then sends the measurement/hash to the data owner. After the data owner is sure about the authenticity of the service code, she can begin to feed data into the service.

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 a limited Enclave Page Cache (EPC). To maintain reasonable performance, we expect the software stack of the DEFLECTION model to control its resource use.

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 side-loading 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 redirection [25], which therefore need proper protection.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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

- 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 known vulnerabilities during the 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.

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.

A. Architecture Overview

The code generator and the binary/proof it produces are all considered untrusted. The code consumer in the TCB is with two components: a dynamic-loader operating a rewriter for relocating 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 immediates being rewriten (step 5). Finally, after the bootstrap transfers the execution to the target program, the service begins and policies are checked at runtime.

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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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).

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-of-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 [38].
- *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 pages with `RWX` properties. DEP protection should, therefore, 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.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Examples include the controlled side channel attack [30] that relies on triggering page faults, and the attacks on L1/L2 caches [39], 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 [40]. Such protection can be integrated into DEFLECTION to mitigate 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 consumer (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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

`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, so runtime code modification cannot be stopped solely by page-level protection.

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 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. All the legitimate labels are also 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.—We incorporated Hyperrace [40] to enforce P6. When an exception or interrupt takes 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 [40], [41]. Specifically, the code generator enforces 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. The instrumented code inspects the marker every q instructions within a basic block, which guarantees that the consecutive AEX(s) triggered will be detected and counted at least once. If an AEX is detected, a co-location test via data race probability will be performed to check co-location of the two threads. Through counting the number of consecutive AEXes, the protected target binary can be aborted if the counted number of AEXs exceeds a preset threshold. The threshold, as a tradeoff of performance and security, can be set by profiling the enclave program in benign environments under reasonable workload. Meanwhile, we parameterized the threshold to control the possibility of an attack is co-located.

We empirically evaluated the accuracy of the co-location tests. As the primary goal of the co-location test is to raise alarms when the two threads are not co-located, we define a false positive α as a false alarm (i.e., the co-location test fails) when the two threads are indeed scheduled on the same physical core. We run the same co-location test code on four different processors (i.e., i7-6700, E3-1280 v5, i7-7700HQ, and i5-6200U). Accuracy values are estimated by conducting 25,600,000 unit tests and results are on the same order of magnitude. We believe it is reasonable to select a desired α value to approximate false positives in practice. More details can be found at our previous work [40].

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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

(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.

We use entropy control to mitigate covert-channels. Since the data owner is the recipient of an enclave's output, all a malicious enclave program can do is to signal to the untrusted OS the content of the data through covert channels, e.g., through system call interfaces. To address this type of covert channel leak, we control the enclave program's input and output behaviors. Specially, the wrapper for `send` encrypts the message to be delivered and pads it to a fixed length. 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.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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 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.

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 [22] as the disassembler.

A. Multi-level Instrumentation

The code generator we built is mainly based on LLVM (Fig. 4), and the assembly-level instrumentation is the core module. More specifically, we implemented modules for checking memory writing instructions, RSP modification, indirect branches and for building shadow stack. We also reformed an instrumentation module to generate side-channel-

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

resilient annotations. To support flexible control of 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. On the in-enclave verifier side, we also use this *separating mechanism and policy* design, allowing for smooth integration of a loading-time pass that supports a new mitigation scheme. More specifically, we provide high-level APIs that allows the developers to implement their instrumentation and validation passes and plug them into the loader [23].

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 (0×3fffffffffff and 0×4fffffffffff), and then compares the address of the 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 provider's code.

Remote attestation.—Once the bootstrap enclave is initiated, it needs to be attested. We leverage the RA-TLS routine [42] and adjust it to our implementation. The conception of "Role" (code owner or data owner) is incorporated in RA-TLS, to make sure the bootstrap enclave can distinguish the two parties and communicate with them using different schemes. 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 submits the quote to IAS and obtains an attestation report.

Dynamic loader.—When the RA is finished, trust between the data owner and the bootstrap enclave is established. The user then 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. The memory size of our bootstrap enclave when initialing is about 96 MB by default, including 1 MB reserved for shadow stack, 1 MB for indirect branch targets, 64 MB for data, 28 MB for service

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

binary code, and less than 2 MB of the loader/verifier. After loading the service binary, the memory cost would be the size of the service binary plus the necessary libraries (e.g., libc, mbedtls, etc.).

Policy verifier.—The policy-compliance verifier, is composed of three components - a clipped disassembler, a verifier, and an immediate operand rewriter.

- *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. When dealing with conditional branching instructions, we add call/jump target instructions to a list of deferred code to be disassembled at later time using the recursive descent algorithm. As a control flow-based algorithm, it can provide very complete code coverage with minimal code. Also, we use the *diet* mode, making the engine size at least 40% smaller [43]. The clipped Capstone consists of 9.1 KLoC as the base of our verifier.
- *Policy verifier*. The verifier and the following rewriter do the work just right after the target binary is disassembled. The verifier uses a simple scanning algorithm to ensure that the policies are 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. The ranges are determined using functions of Intel SGX SDK during dynamic loading (Section IV-D).

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 I. Security and privacy are guaranteed by the lightweight in-enclave verifier (in TCB), even if the code generator has mis-compilation errors. The correctness of our verifier can be formally verified, using memory safety verification tools such as SMACK and model checking tools such as SPIN.

The loader we implemented consists of less than 600 lines of code (LoCs) and the verifier includes less than 700 LoCs, also integrating the SGX SDK and part of Capstone libraries. The binary sizes of shielded runtimes such as Graphene-SGX increase to 2.5 times or

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

more compared to ours. Currently, Occlum has not integrated the SFI feature in its latest version [44], 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.

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 [13] 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 work [20], [45] 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 [45], [46]. 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 [40] 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 either. 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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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.

Performance on nBench.—We instrumented all applications in the SGX-nBench [47], and ran each testcase of the nBench suites under a few settings, each for 10 times. Table II shows the average execution time under different settings. Without side channel mitigation (P1-P5), our prototype introduces an0.3% to 25% overhead (on FP-emulation). Apparently, the store instruction instrumentation alone (P1) does not cause a large performance overhead, with the largest being 6.7%. Also, when P1 and P2 are applied together, the overhead just becomes slightly higher than P1 is enforced alone. The performance overhead fluctuates from application to application since different instrumentations are applied. FP EMULATION has much less memory write operations than others. And it has rare indirect branches. Compared to it, ASSIGNMENT uses a lot of function pointers, which leads to a relatively heavy instrumentation overhead of enforcing P5. Besides, almost all benchmarks in nBench perform well under the CFI check P5 (less than 4%) except for 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. We evaluated multiple settings (input sizes) and reported the most representative results, in ascending order across at least one order of magnitude. The baseline results are measured on a pure loader, with no security/privacy policies enforced.

- *Sensitive genome data analysis*. We implemented the Needleman–Wunsch algorithm [48] that aligns two human genomic sequences in the FASTA format [49] taken from the 1000 Genomes project [50]. 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. Sometimes

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

the differences between P1+P2 and P1-P5 seem slight mainly because the instrumentations on indirect branches (P5) are few. Meanwhile, the instrumentation to enforce P1/P2 can be reused to enforce P3/P4 (via different boundaries). Thus, the performance overhead caused by P3/P4 is negligible (when P1/P2 are already enforced).

- *Personal credit score analysis*. In our study, we implemented a BP neural network-based credit scoring algorithm [51] that trains a model to calculate 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 [52]. The case of HTTPS server is to show that DEFLECTION is capable of handling multiple clients and it outperforms other solutions when the data size is increasing. A client executes a stress test tool - Siege [53] - 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 [44] 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.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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 [40], others [41], [54]–[59] can also be transformed and incorporated into the design, specifically for mitigating cache timing, memory bus timing [60], and other timing channels. ORAM [61], [62] 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 [63]. 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 [64].

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 [65]. The proof enforcement of CFI may suffer from a time of check to time of use (TOCTOU) problem [66]. To cope with that, we can make all CFI metadata to be kept in the register or hardware [67] instead of in memory, and guarantee that the instrumented proof could not be modified by any threads [68].

VIII. Related Work

Secure computing using SGX.

Many existing works propose using SGX to secure cloud computing systems, e.g., VC3 [46], TVM [69], by using sand-boxing, containers [70], and others [71], [72]. In-enclave JVM interpreter is also a good choice [73]. 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 [9] and its follow-up work [74] provide an distributed sand-box by porting NaCl to the enclave environment, confining untrusted data-processing modules to prevent leakage of the user's 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 [75] 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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

translation used by traditional SFI [76], [77]. The advantage of our design compared to other state-of-the-art shielding runtimes (e.g., Occlum) is three-fold. Firstly, DEFLECTION is more general. The memory access check of Occlum relies on hardware (Intel MPX) which is no longer supported, significantly hindering deployment. Secondly, DEFLECTION has a smaller size of TCB. Other than importing Zydis and PyVEX (in Python) to be the disassembler and verifier, we shrank and modified Capstone (in C) to implement our smaller disassembler and verifier. Thirdly, DEFLECTION can mitigate some side/covert channel leaks while others provide no such protection.

Code privacy.

Code secrecy is an easy to be ignored but very important issue [78]. TEEshift [12], DynSGX [79] and SGXElide [80] 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 [20] and its follow-up works [45] 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 [81] and a theorem prover [33]. 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. 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.

Acknowledgment

We would like to express our sincere thanks to our shepherd Rüdiger Kapitza and the anonymous reviewers for their valuable feedback to help us improve the paper. This research is supported in part by NIH R01HG010798 and NSF CNS-1838083. Wenhao Wang was partially supported by National Natural Science Foundation of China (Grant No.61802397).

References

[1]. McKeen F, Alexandrovich I, Berenzon A, Rozas CV, Shafi H,Shanbhogue V, and Savagaonkar UR, "Innovative Instructions and Software Model for Isolated Execution." HASP, vol. 10, no. 1, 2013

[2]. Russinovich M, "Introducing Azure Confidential Computing," Seattle, WA: Microsoft, 2017.

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

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

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

[5]. Zhang Z, Ding X, Tsudik G, Cui J, and Li Z, “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]. Priebe C, Muthukumaran D, Lind J, Zhu H, Cui S, Sartakov VA, and Pietzuch P, “SGX-LKL: Securing the Host OS Interface for Trusted Execution,” arXiv preprint arXiv:1908.11143, 2019.

[7]. Shen Y, Tian H, Chen Y, Chen K, Wang R, Xu Y, Xia Y, and Yan S, “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.

[8]. Arnautov S, Trach B, Gregor F, Knauth T, Martin A, Priebe C,Lind J, Muthukumaran D, O’Keeffe D, Stillwell ML, Goltzsche D,Eyers D, Kapitza R, Pietzuch P, and Fetzer C, “SCONE: Secure Linux Containers with Intel SGX,” in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, pp. 689–703.

[9]. Hunt T, Zhu Z, Xu Y, Peter S, and Witchel E, “Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data,” ACM Transactions on Computer Systems (TOCS), vol. 35, no. 4, p. 13, 2018.

[10]. Wang H, Wang P, Ding Y, Sun M, Jing Y, Duan R, Li L, Zhang Y,Wei T, and Lin Z, “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.

[11]. Wang H, Bauman E, Karande V, Lin Z, Cheng Y, and Zhang Y, “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.

[12]. Lazard T, Götzfried J, Müller T, Santinelli G, and Lefebvre V, “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.

[13]. Van Bulck J, Oswald D, Marin E, Aldoseri A, Garcia FD, and Piessens F, “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.

[14]. Necula GC, “Proof-Carrying Code,” in Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages. ACM, 1997, pp. 106–119.

[15]. Schneider FB, Morrisett G, and Harper R, “A Language-based Approach to Security,” in Informatics. Springer, 2001, pp. 86–101.

[16]. Colby C, Lee P, Necula GC, Blau F, Plesko M, and Cline K, “A Certifying Compiler for Java,” in ACM SIGPLAN Notices, vol. 35, no. 5. ACM, 2000, pp. 95–107.

[17]. Leroy X, “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.

[18]. Pirzadeh H, Dubé D, and Hamou-Lhadj A, “An Extended Proof-Carrying Code Framework for Security Enforcement,” in Transactions on computational science XI. Springer, 2010, pp. 249–269.

[19]. Necula GC and Rahul SP, “Oracle-based Checking of Untrusted Software,” in ACM SIGPLAN Notices, vol. 36, no. 3. ACM, 2001, pp. 142–154.

[20]. Sinha R, Rajamani S, Seshia S, and Vaswani K, “Moat: Verifying Confidentiality of Enclave Programs,” in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 1169–1184.

[21]. Morrisett G, Walker D, Crary K, and Glew N, “From System F to Typed Assembly Language,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 21, no. 3, pp. 527–568, 1999.

[22]. “Capstone - The Ultimate Disassembler,” http://www.capstone-engine.org/.

[23]. “DEFLECTION,” https://github.com/StanPlatinum/Deflection.

[24]. Lee J, Jang J, Jang Y, Kwak N, Choi Y, Choi C, Kim T, Peinado M, and Kang BB, “Hacking in Darkness: Return-Oriented Programming against Secure Enclaves,” in 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 523–539.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

[25]. Biondo A, Conti M, Davi L, Frassetto T, and Sadeghi A-R, “The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel SGX,” in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 1213–1227.

[26]. Schwarz M, Weiser S, and Gruss D, “Practical Enclave Malware with Intel SGX,” in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2019, pp. 177–196.

[27]. Schwarz M, Weiser S, Gruss D, Maurice C, and Mangard S, “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.

[28]. Lee S, Shih M-W, Gera P, Kim T, Kim H, and Peinado M, “Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing,” in 26th USENIX Security Symposium (USENIX Security17), 2017, pp. 557–574.

[29]. Gras B, Razavi K, Bos H, and Giuffrida C, “Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks,” in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 955–972.

[30]. Xu Y, Cui W, and Peinado M, “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems,” in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 640–656.

[31]. Homeier PV and Martin DF, “A Mechanically Verified Verification Condition Generator,” The Computer Journal, vol. 38, no. 2, pp. 131–141, 1995.

[32]. Paulson LC, “Isabelle: The Next 700 Theorem Provers,” arXiv preprint cs/9301106, 2000.

[33]. De Moura L and Bjørner N, “Z3: An Efficient SMT Solver,” in International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008, pp. 337–340.

[34]. Bertot Y and Castéran P, Interactive Theorem Proving and Program Development: Coq’Art: The Calculus of Inductive Constructions. Springer Science & Business Media, 2013.

[35]. Appel AW, Michael N, Stump A, and Virga R, “A Trustworthy Proof Checker,” Journal of Automated Reasoning, vol. 31, no. 3–4, pp. 231–260, 2003.

[36]. Brumley D, Jager I, Avgerinos T, and Schwartz EJ, “BAP: A Binary Analysis Platform,” in International Conference on Computer Aided Verification. Springer, 2011, pp. 463–469.

[37]. Appel AW, “Foundational Proof-carrying Code,” in Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. IEEE, 2001, pp. 247–256.

[38]. Wang Z, Ding X, Pang C, Guo J, Zhu J, and Mao B, “To Detect Stack Buffer Overflow with Polymorphic Canaries,” in 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2018, pp. 243–254.

[39]. Wang W, Chen G, Pan X, Zhang Y, Wang X, Bindschaedler V,Tang H, and Gunter CA, “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.

[40]. Chen G, Wang W, Chen T, Chen S, Zhang Y, Wang X, Lai T-H, and Lin D, “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.

[41]. Gruss D, Lettner J, Schuster F, Ohrimenko O, Haller I, and Costa M, “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory,” in 26th USENIX Security Symposium (USENIX Security 17), 2017, pp. 217–233.

[42]. Knauth T, Steiner M, Chakrabarti S, Lei L, Xing C, and Vij M, “Integrating Remote Attestation with Transport Layer Security,” arXiv preprint arXiv:1801.05863, 2018.

[43]. Quynh NA, “Capstone: Next-Gen Disassembly Framework,” Black Hat USA, 2014.

[44]. “Occlum,” https://github.com/occlum/occlum.

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

[46]. Schuster F, Costa M, Fournet C, Gkantsidis C, Peinado M, Mainar-Ruiz G, and Russinovich M, “VC3: Trustworthy Data Analytics in The Cloud Using SGX,” in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 38–54.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

[47]. “SGX nBench,” https://github.com/utds3lab/sgx-nbench.

[48]. Needleman SB and Wunsch CD, “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. [PubMed: 5420325]

[49]. “Fasta format,” https://en.wikipedia.org/wiki/FASTA_format.

[50]. “1000 Genomes Project,” https://en.wikipedia.org/wiki/1000_Genomes_Project.

[51]. Jensen HL, “Using Neural Networks for Credit Scoring,” Managerial finance, vol. 18, no. 6, pp. 15–26, 1992.

[52]. “mbedTLS,” https://tls.mbed.org/.

[53]. “Siege,” https://www.joedog.org/siege-home/.

[54]. Doychev G, Köpf B, Mauborgne L, and Reineke J, “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.

[55]. Almeida JB, Barbosa M, Barthe G, Dupressoir F, and Emmi M, “Verifying Constant-Time Implementations,” in 25th USENIX Security Symposium (USENIX Security 16), 2016, pp. 53–70.

[56]. Shih M-W, Lee S, Kim T, and Peinado M, “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs,” in NDSS, 2017.

[57]. Wu M, Guo S, Schaumont P, and Wang C, “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.

[58]. Wang S, Bao Y, Liu X, Wang P, Zhang D, and Wu D, “Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation,” in 28th USENIX Security Symposium (USENIX Security19), 2019, pp. 657–674.

[59]. Orenbach M, Michalevsky Y, Fetzer C, and Silberstein M, “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.

[60]. Liu F, Wu H, and Lee RB, “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.

[61]. Sasy S, Gorbunov S, and Fletcher CW, “ZeroTrace: Oblivious Memory Primitives from Intel SGX,” IACR Cryptol. ePrint Arch, vol. 2017, p. 549, 2017.

[62]. Ahmad A, Joe B, Xiao Y, Zhang Y, Shin I, and Lee B, “OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX,” in NDSS, 2019.

[63]. Liu W, Gao D, and Reiter MK, “On-Demand Time Blurring to Support Side-Channel Defense,” in European Symposium on Research in Computer Security. Springer, 2017, pp. 210–228.

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

[65]. Weichbrodt N, Kurmus A, Pietzuch P, and Kapitza R, “AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves,” in European Symposium on Research in Computer Security. Springer, 2016, pp. 440–457.

[66]. Xu X, Ghaffarinia M, Wang W, Hamlen KW, and Lin Z, “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.

[67]. DeLozier C, Lakshminarayanan K, Pokam G, and Devietti J, “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.

[68]. Burow N, Zhang X, and Payer M, “SoK: Shining Light on Shadow Stacks,” in 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019, pp. 985–999.

[69]. Hynes N, Cheng R, and Song D, “Efficient Deep Learning on Multi-Source Private Data,” arXiv preprint arXiv:1807.06689, 2018.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

[70]. Tian D, Choi JI, Hernandez G, Traynor P, and Butler KR, “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.

[71]. Shinde S, Le Tien D, Tople S, and Saxena P, “Panoply: Low-TCB Linux Applications With SGX Enclaves.” in NDSS, 2017.

[72]. Shanker K, Joseph A, and Ganapathy V, “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.

[73]. Jiang J, Chen X, Li T, Wang C, Shen T, Zhao S, Cui H, Wang C-L, and Zhang F, “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.

[74]. Hunt T, Song C, Shokri R, Shmatikov V, and Witchel E, “Chiron: Privacy-preserving Machine Learning as a Service,” arXiv preprint arXiv:1803.05961, 2018.

[75]. Erlingsson Ú, Abadi M, Vrable M, Budiu M, and Necula GC, “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.

[76]. Zhou Y, Wang X, Chen Y, and Wang Z, “ARMlock: Hardware-based Fault Isolation for ARM,” in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 558–569.

[77]. Tan G, Principles and Implementation Techniques of Software-Based Fault Isolation. Now Publishers, 2017.

[78]. Küçük KA, Grawrock D, and Martin A, “Managing confidentiality leaks through private algorithms on Software Guard eXtensions (SGX) enclaves,” EURASIP Journal on Information Security, vol. 2019, no. 1, p. 14, 2019.

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

[80]. Bauman E, Wang H, Zhang M, and Lin Z, “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.

[81]. Barnett M, Chang B-YE, DeLine R, Jacobs B, and Leino KRM, “Boogie: A Modular Reusable Verifier for Object-Oriented Programs,” in International Symposium on Formal Methods for Components and Objects. Springer, 2005, pp. 364–387.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

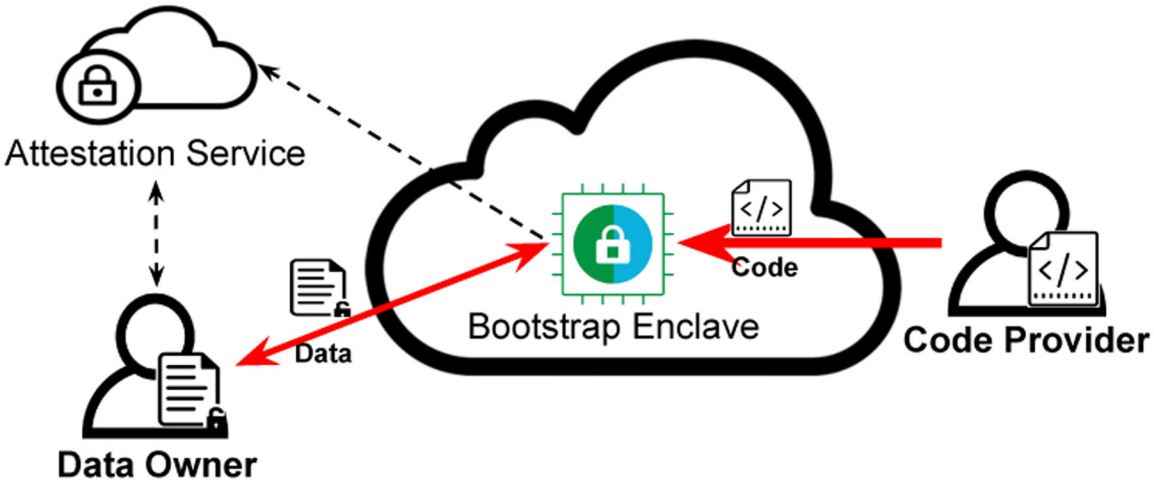

Fig. 1:
The DEFLECTION model

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

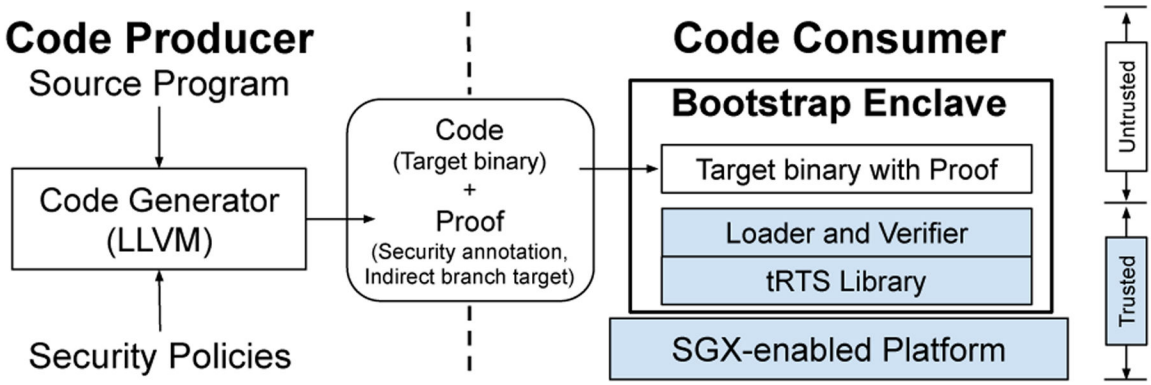

Fig. 2:
System overview

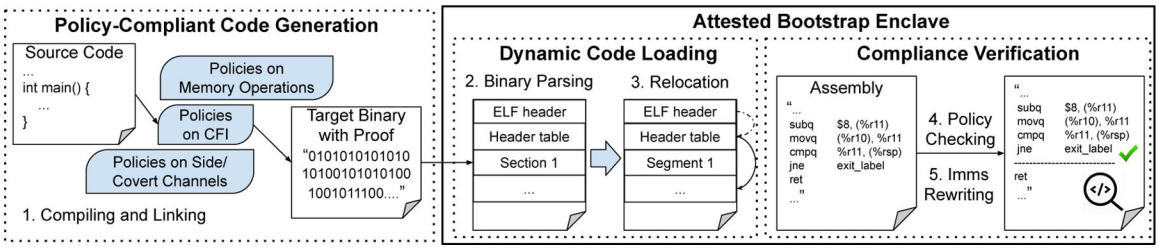

Fig. 3:
Detailed framework and workflow

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

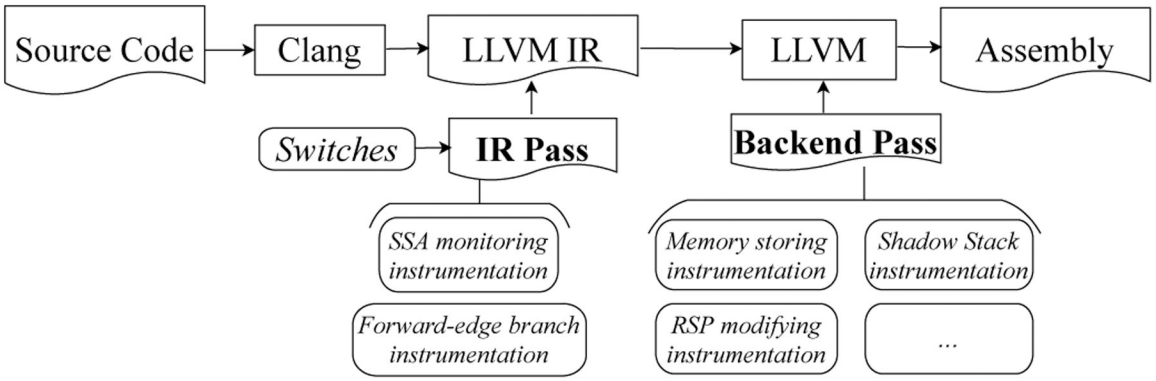

Fig. 4:
Workflow of flexible code generation

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

```
1  pushq   %rbx      ;save execution status
2  pushq   %rax
3  leaq    [reg+imm], %rax ;load the operand
4  movq    $0x3FFFFFFFFFFFFFFF, %rbx  ;set bounds
5  cmpq    %rbx, %rax
6  ja      exit_label
7  movq    $0x4FFFFFFFFFFFFFFF, %rbx  ;set bounds
8  cmpq    %rbx, %rax
9  jb      exit_label
10 popq    %rax
11 popq    %rbx
12 movq    reg, [reg+imm]
```

Fig. 5:
Store instruction instrumentation

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

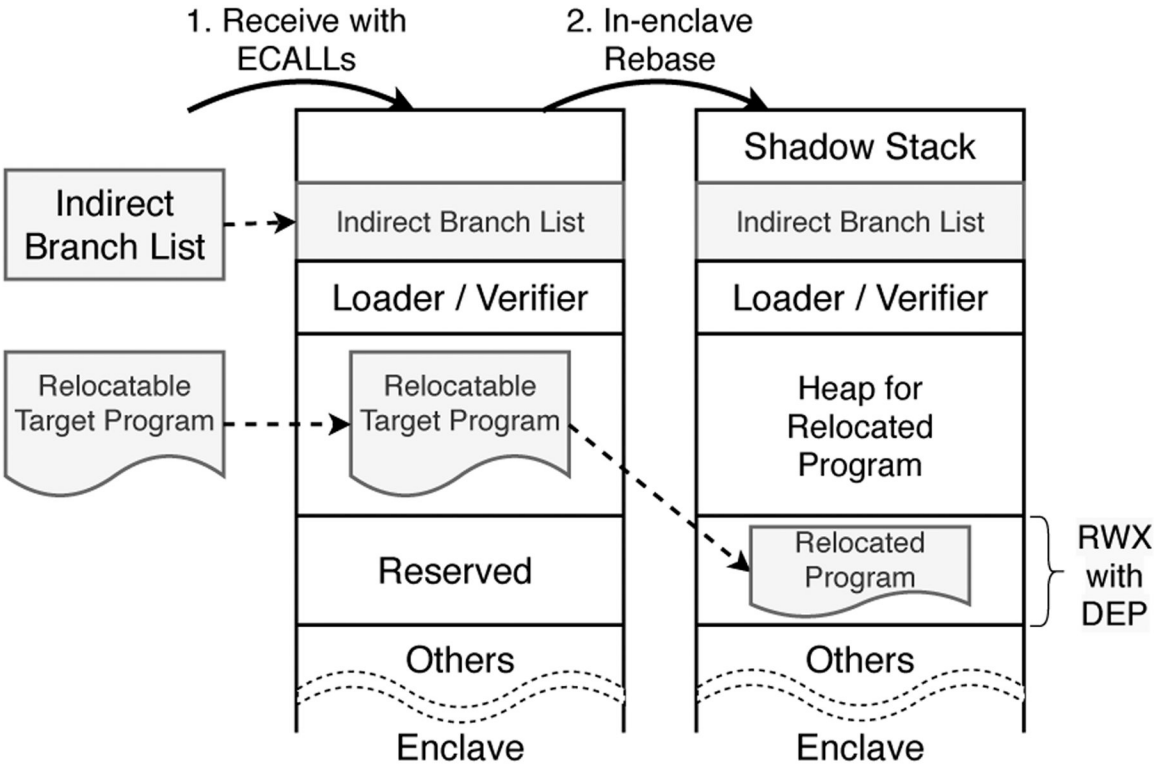

Fig. 6:
Detailed workflow of the dynamic loader

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

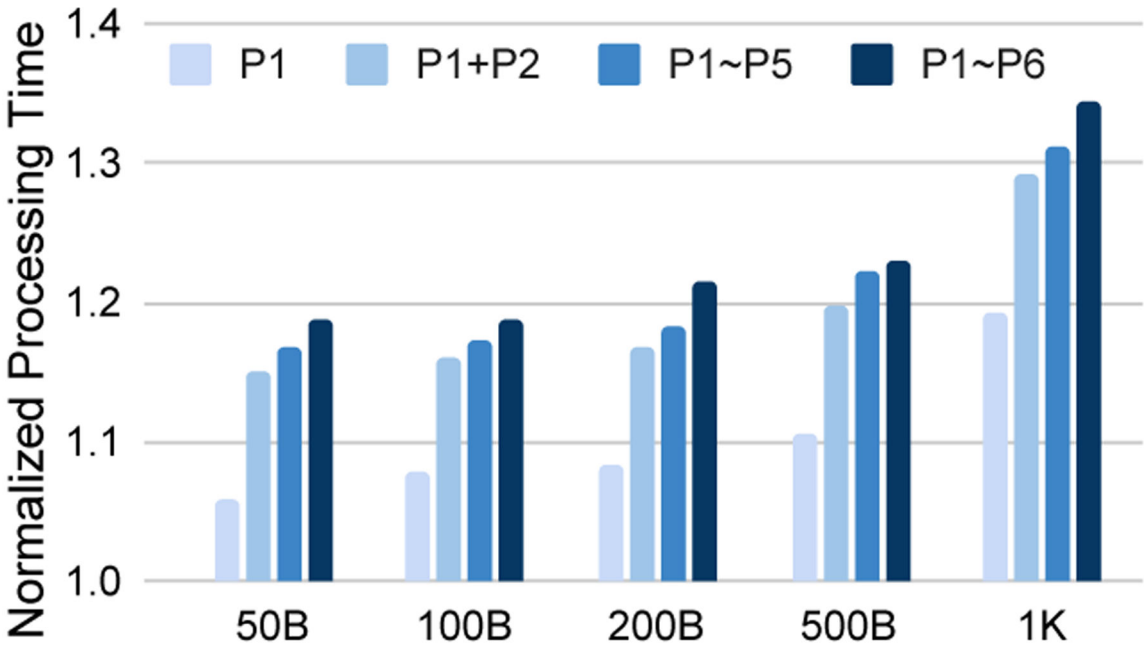

Fig. 7:
Sequence alignment

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

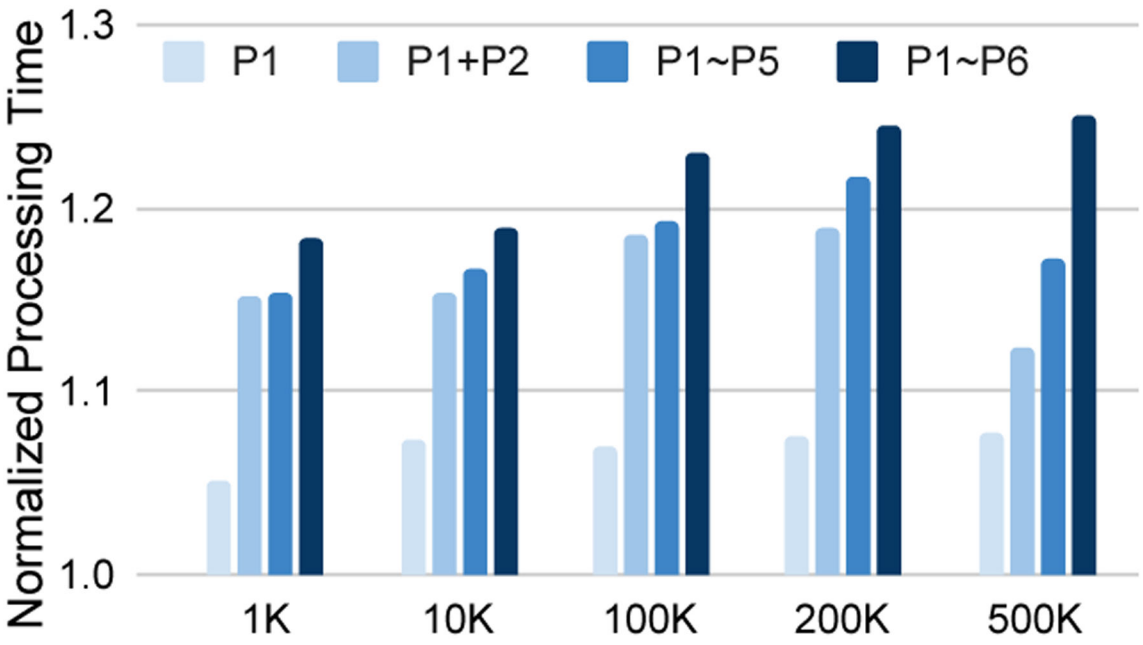

Fig. 8:
Sequence generation

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

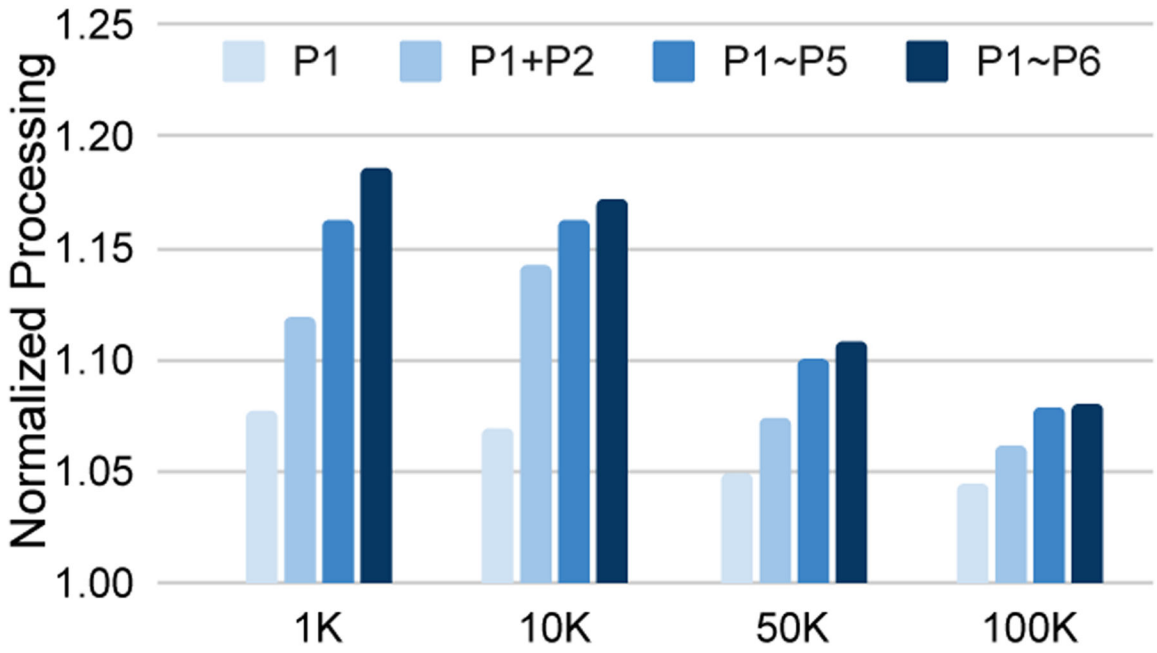

Fig. 9:
Credit scoring

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

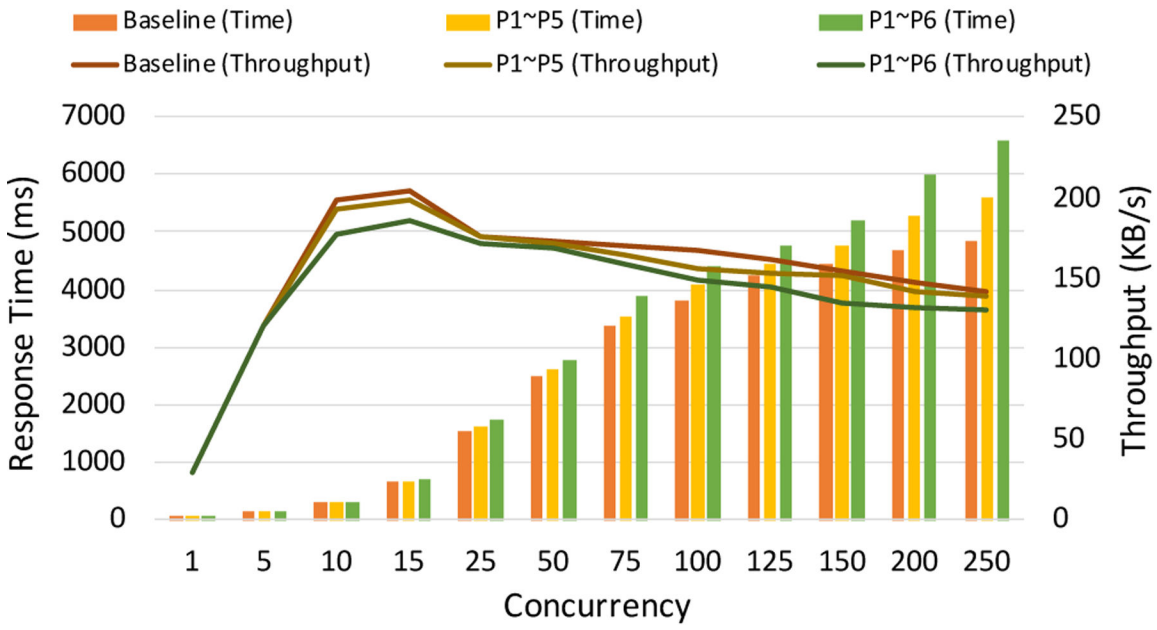

Fig. 10:
Performance on HTTPS server

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

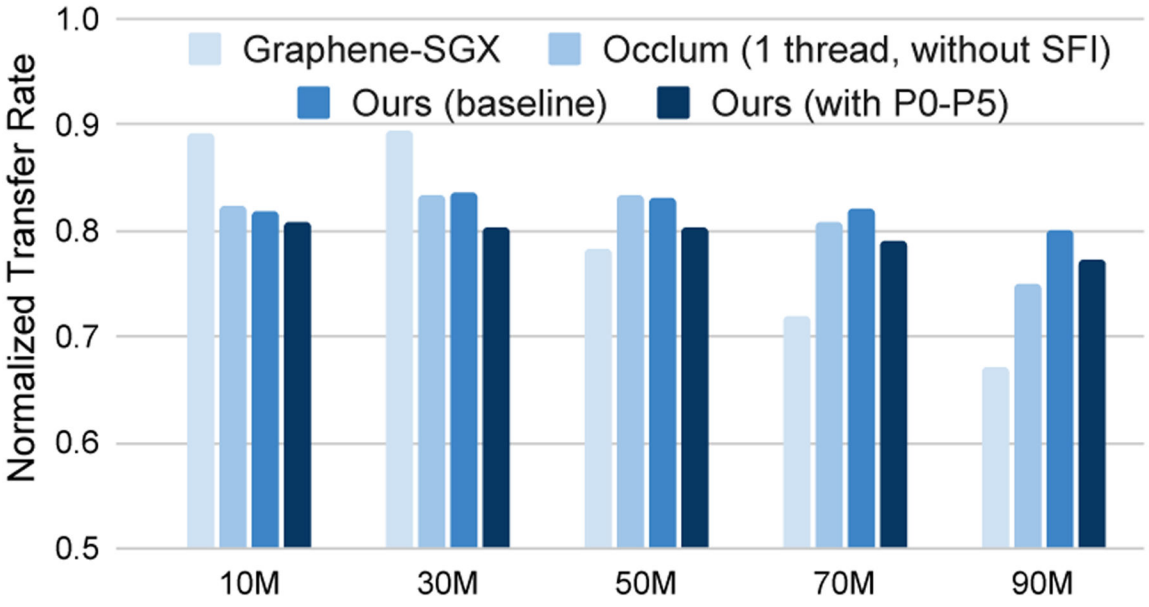

Fig. 11:
Performance comparison

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE I:

TCB comparison with other solutions

Shielding runtimes	Core components	kLoCs	Size(MB)
Ryoan	Eglibc	892	> 19
	NaCl sandbox	216	
	Naclports	460	
SCONE	OS Shield and shim libc	187	> 16
Graphene-SGX	Glibc	1200	> 58.5
	LibPAL	22	
	Graphene LibOS	34	
Occlum	Occlum shim libc	93	> 8.6
	Verifier	N/A	
	Occlum LibOS and PAL	24.5	
DEFLECTION	Loader/Verifier	1.3	3.5
	RA/Encryption	0.2	
	Shim libc	33	
	Capstone base	9.1	
	Other dependencies	23	

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

TABLE II:

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%