Leveraging Hardware Isolation for Process Level Access Control & Authentication

Syed Kamran Haider† Hamza Omar† Ilia Lebedev‡ Srinivas Devadas‡ Marten van Dijk†
† University of Connecticut ‡ Massachusetts Institute of Technology
[syed.haider,hamza.omar,marten.van_dijk]@uconn.edu
[ilebedev,devadas]@mit.edu

ABSTRACT

Critical resource sharing among multiple entities in a processing system is inevitable, which in turn calls for the presence of appropriate authentication and access control mechanisms. Generally speaking, these mechanisms are implemented via trusted software "policy checkers" that enforce certain high level application-specific "rules" to enforce a policy. Whether implemented as operating system modules or embedded inside the application ad hoc, these policy checkers expose additional attack surface in addition to the application logic. In order to protect application software from an adversary, modern secure processing platforms, such as Intel's Software Guard Extensions (SGX), employ principled *hardware isolation* to offer secure software containers or *enclaves* to execute trusted sensitive code with some integrity and privacy guarantees against a privileged software adversary. We extend this model further and propose using these hardware isolation mechanisms to shield the authentication and access control logic essential to policy checker software. While relying on the fundamental features of modern secure processors, our framework introduces productive software design guidelines which enable a guarded environment to execute sensitive policy checking code – hence enforcing application control flow integrity – and afford flexibility to the application designer to construct appropriate high-level policies to customize policy checker software.

CCS CONCEPTS

•**Security and privacy** → *Systems security; Software and application security;*

KEYWORDS

Hardware Isolation; Secure Processors; Program Authentication

ACM Reference format:
Syed Kamran Haider† Hamza Omar† Ilia Lebedev‡ Srinivas Devadas‡ Marten van Dijk†. 2017. Leveraging Hardware Isolation for Process Level Access Control & Authentication. In *Proceedings of SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA, ,* 9 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078882

SACMAT'17, June 21–23, 2017, Indianapolis, IN, USA
 ACM ISBN 978-1-4503-4702-0/17/06…$15.00.
DOI: http://dx.doi.org/10.1145/3078861.3078882

1 INTRODUCTION

Technology plays a major role in shaping various different aspects of our lives in the modern digital world. We, as users, interact with hundreds of digital devices in our day-to-day lives such as mobile devices, electronic households, medical equipments, automobiles, media players and many more. One way or the other, these devices access certain resources that are shared among various entities, be it other users or internal management modules of the system. For instance, in a cloud setting, users share physical storage and computational resources to store their private data and perform arbitrary computations on this data. *Authentication* and *access control* mechanisms are vital components of such systems that verify a user's/task's identity and regulate its requests to access certain system resources respectively.

Authentication and access control are typically quite interrelated. In general, a user is granted or denied permission to access a certain resource by first authenticating its identity and then looking up an *access permissions list* corresponding to this particular user. Common authentication mechanisms include password based and public key based schemes [26]. Password based systems require an initial (secure) setup phase before the authentication phase, whereas public key based systems use a trusted certification authority. Other advanced mechanisms incorporate multiple authentication factors, e.g., by exploiting physiological biometrics (iris, fingerprints, hand, retinal), behavioral characteristics (voice, typing pattern) [11], and human psychology to understand and predict human behavior [21].

In modern applications, typically the authentication and access control mechanisms are implemented via trusted software "policy checker" modules that enforce certain high level application-specific "rules" to enforce a policy. Preventing any malicious manipulations to an application's control flow is crucial for the overall program integrity, as well as for any software based authentication/access control policies to be effective. Control flow manipulations in the untrusted (i.e., potentially buggy) parts of the application code are possible by exploiting any *memory safety violations*, e.g., through a buffer overflow vulnerability etc. Whereas parts of the application trusted to be bug-free, including the policy checker software, are not vulnerable to such attacks. Nevertheless, substantial amount of research has been done on the subject of detecting and protecting against software-level memory safety violations at instruction-by-instruction level fine granularity as well as checkpointing based coarse granularity. Among the various proposals are software only approaches [1, 3, 18–20] and partially or fully hardware-assisted approaches [5, 8, 24].

In spite of the policy checker software being trusted, a fundamental problem that still persists is that this software is vulnerable

to *privileged software attacks*. For instance, a compromised operating system or hypervisor (that runs at a higher privilege level than the application – hence called a privileged software) having full control over the system resources can not only learn the application's sensitive information [25], but also tamper with its data and/or manipulate its control flow to bypass the policy checking software. State-of-the-art *secure processors*, such as Intel SGX [17] and Sanctum [6], offer hardware assisted secure containers or *enclaves* to run the sensitive application code (e.g., policy checker) in a protected environment. Although this paradigm protects the application against a bunch of privileged software attacks, yet it does not enforce the "correct" execution ordering of enclaves from the application's perspective. In other words, the application's control flow can still be manipulated at enclaves level granularity, resulting in crucial policy violations. For example, a compromised OS could completely bypass the policy checker enclave and allow the adversary an unauthorized access to the system resources.

In this paper, we introduce an application software design framework based on modern secure processors which, in a nutshell, uses an enclaved policy checker engine to manage the capabilities of the rest of the system. Given that the policy engine is a persistent trusted component, it offers a powerful security property: *trustworthy ordering* of enclaves' executions in the presence of compromised privileged software. In our approach, the application designer wraps the policy checker code into a separate *policy enclave* along with creating several other enclaves for sensitive parts of the application. The policy enclave, depending upon the application, administers corresponding rules for authentication and access control, as well as verifies the enclaves' execution flow at each step. While offering traditional isolation guarantees, the proposed approach ensures that the policy enclave is invoked upon all "sensitive" transitions of the program; particularly before each entry to a regular enclave. In addition to preventing the privileged software from circumventing the policy checks, our framework offers the application designer a global picture of the application execution state (through the policy enclave) which in turn provides the designer a much richer set of information to design policy checker software.

2 BACKGROUND

The attacks that can be performed on a computer system can broadly be classified into *physical attacks* and *software/remote attacks*. To launch physical attacks, the adversary having the physical access to the computer system might tamper with the system or exploit its physical implementation to perform an authorized operation or learning secret information. In contrast, remote adversaries can only launch software attacks which might involve executing potentially malicious software at the victim system and/or accessing the victim's confidential data. Generally, a remote adversary is more common than a physical adversary since usually the user either owns the system (e.g. a personal computer) or the computing infrastructure (e.g. a cloud server) is managed by trusted parties. With this model in mind, we only focus on the remote adversaries in this paper. Adversaries having physical access to the computer system are out of scope of this work.

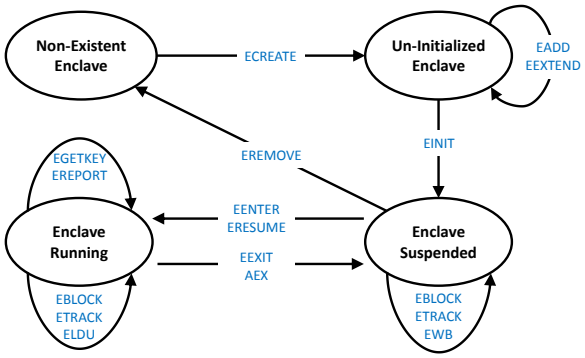

Figure 1: Life cycle of an enclave in Intel SGX.

2.1 Trusted Hardware

In the presence of physical adversaries, the privacy of user's sensitive data becomes a serious concern. To address this challenge, various *trusted-hardware* based secure processor architectures have been proposed in the literature [10, 13–17, 22, 23]. A trusted-hardware platform receives user's encrypted data, which is decrypted and computed upon inside the trusted boundary, and finally the encrypted results of the computation are sent to the user. The trusted-hardware is assumed to be *tamper-resistant*, i.e. an adversary cannot probe the processor chip to learn any information.

In addition to protecting against certain physical adversaries described above, secure processors also introduced the notion of *secure containers* based on two important properties, *Hardware Isolation* and *Attestation*. Secure containers also called *enclaves* can be used to defend against certain software adversaries. For example, the XOM architecture [13–15] uses isolated containers to execute sensitive code on the user's data without trusting the OS. Aegis [22, 23] makes use of a *trusted security kernel* to isolate each container from other software running on the computer system by configuring the page tables used for address translation. The security kernel is a subset of a typical OS kernel, and handles virtual memory management, processes, and hardware exceptions.

This leads us to give the reader a quick background of two state-of-the-art secure processor architectures: an industrial flagship architecture Intel SGX and an academic architecture Sanctum with a similar API. We will be using these systems as the underlying hardware for our proposed framework that fits both these architectures equally well.

2.2 Intel SGX

Intel's Software Guard Extentions (SGX) [17] follows Bastion's [4] approach of having the untrusted OS manage the page tables; and also adapts the ideas from Aegis and XOM to multi-core processors (having a shared, coherent last-level cache) by introducing the concept of *enclaves*. An enclave is a protected environment that contains the code and data of a security-sensitive computation. Each enclave's environment is isolated from the untrusted software outside the enclave, as well as from other enclaves. SGX maintains *isolation* by setting aside a memory region, called the *Processor*

Reserved Memory (PRM). The CPU protects the PRM from all non-enclave memory accesses, including kernel, hypervisor and system management mode accesses, and DMA accesses from peripherals. PRM holds an *Enclave Page Cache* (EPC) which contains information regarding enclave's code and data. An enclave is allowed to only access its own information stored in the EPC.

Intel SGX also provides the attestation capabilities for authentication purposes. Attestation allows an enclave to attest its execution environment to other enclaves on the same system platform *(Local Attestation)*, and also to the entities outside the platform *(Remote Attestation)*. Remote attestation is a method by which an SGX enabled processor authenticates its hardware and software configuration to a remote host. It is achieved with the help of a *Quoting Enclave*, a specially designed enclave that uses an attestation key to sign the execution "report" and allows the remote party to verify the signature using a public key.

Figure 1 shows how the enclaves in the SGX architecture are created, initialized, and loaded. It also explains how, with the use of specified instructions, an application designer can context switch between enclaves and/or non-enclave code. The following instructions (embedded as micro-code in the ISA), represented in Figure 1, explain the major transitions during a SGX enclave's life cycle.

- *ECREATE:* Creates a unique instance of an enclave.
- *EADD:* Adds pages and thread control structures into the enclave.
- *EEXTEND:* Generates a cryptographic hash of the content of the enclave.
- *EINIT:* Initializes the enclave and marks it ready to be used.
- *EENTER, ERESUME:* Enters the enclave or resumes after context switching/exiting.
- *EEXIT, AEX:* Exits the enclave synchronously, or asynchronously.
- *EREMOVE:* Refers to a tear-down of an enclave as it deallocates the page from the EPC permanently.
- *EREPORT:* Application enclave calls this instruction to generate a report structure for a desired target enclave for local attestation.
- *EGETKEY:* Used by the target enclave to retrieve the report key generated by the application enclave.

As complementary mechanisms, SGX also implements memory integrity verification and encryption of the memory.

2.3 Sanctum

Sanctum [6] follows SGX's API and offers the same promise of strong provable isolation of software running concurrently and sharing resources. It is built on top of an open source implementation of Rocket RISC-V core. Sanctum is a hardware software co-design that introduces a small *trusted software security monitor* that effectively extends the ISA as in SGX. Furthermore, Sanctum introduces nominal hardware extensions:

- ***DMA Tranfer Filtering:*** A protection circuitry to prevent peripherals accessing enclave's memory regions.
- ***Cache Address Shifter:*** An additional hardware to circularly shift the Physical Page Number to the right by a certain amount of bits to map collections of cache sets to contiguous DRAM regions.
- ***Page Walker Input:*** A small circuit to select the appropriate page table base and forward either *EPTBR* (enclave page table base register) or *PTBR* (page table base register).

As discussed in section 2.2, SGX hardware ensures security via its high privileged microcode which extends to the Instruction Set Architecture (ISA). Updates to the microcode in SGX can only be made through Intel. In contrast, Sanctum uses an *open-sourced security monitor* implementation which is provided at the *machine level* (highest privilege level) in RISC-V. For updating the security monitor, Sanctum employs a *bootstrapping* mechanism which allows a user to write and update its own security monitor. All layers of Sanctum's trusted computing base (TCB) are open-sourced at www.github.com/pwnall/sanctum.

Unlike SGX, each Sanctum enclave is responsible for its own memory management with respect to page swapping between reserved enclave memory and the untrusted DRAM. The *security monitor* provides API calls to the OS and enclaves for DRAM region allocation and enclave management, and also guards sensitive registers, such as the page table base register. The life cycle of a Sanctum enclave is very similar to that of its SGX counterpart, as shown in Figure 1. Sanctum's software attestation process relies on *mailboxes*, a simplified version of SGX's crypto based local attestation mechanism (*EGETKEY* and *EREPORT*) that uses key-derivation and MAC algorithms. Enclave mailboxes are stored in metadata regions and isolated from other mailboxes, which cannot be accessed by any software/OS other than the *security monitor*. In Sanctum, an enclave (say A) can invoke a secure inter-enclave messaging service to send an *accept message* monitor call to specify the mailbox that will receive the message and the identity of the enclave (say B) that is expected to send the message. The sending enclave (B) performs a *send message* call that specifies the identity of the receiving enclave and a mailbox within that enclave. The security monitor delivers messages to mailboxes that expect them. It notifies the enclave (A) that it has received a message, which issues a *read message* call to the security monitor that moves the message from the mailbox into enclave's memory.

Unlike SGX, Sanctum protects against an important class of additional software attacks that infer private information through a program's cache access patterns. It employs the concept of cache partitioning based on page coloring. Sanctum shows that *isolation* of concurrent software modules (which relies on caching DRAM regions in distinct sets) provides strong security guarantees against a subtle software threat model at the cost of small performance overhead.

2.4 Promises vs. Limitations

As we discussed so far, modern secure processors offer strong isolated environments to securely execute sensitive code in the presence of strong software adversaries. Nonetheless, it is important to highlight *what secure processors do not promise!* In particular, SGX/Sanctum only guarantee that each *individual* enclave is securely and correctly executed, and this can be verified through the hash digest or so called *report* of the enclave. However, in an application with multiple enclaves having interdependencies, it

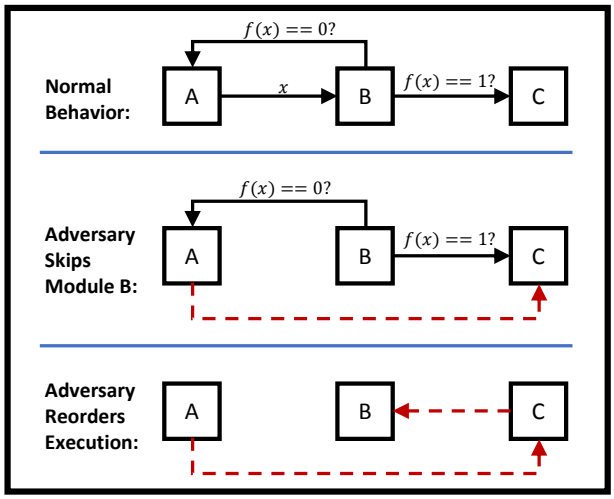

Figure 2: Possible control flow manipulations done by an adversary.

is the programmer's responsibility to not only verify (via enclave reports) that each enclave has executed the expected code on the expected/correct inputs, but also that the global order in which the enclaves are executed is consistent with the application design. The latter condition is equally important for the overall integrity and security of the application.

For instance, Figure 2 shows a toy example of a program with three enclaves A, B and C. Under the *normal behavior* of the program, enclave A provides some input credentials x to B which computes an authentication function f on x. If the authentication is successful, i.e. $f(x) = 1$, only then enclave C is allowed to execute (and then access some critical system resource). In case of authentication failure, i.e. $f(x) = 0$, the control is returned back to enclave A without executing C. Now, a strong adversary, who can alter the global ordering of enclaves' execution, can do the following: it can skip the authentication step performed by enclave B and directly execute C gaining an unauthorized access to the corresponding resource, or it can first execute C and then perform the (useless) authentication step as shown in Figure 2. Hence it is crucial to maintain the global execution ordering of the application modules. At least for now, it is out of scope for secure processors to enforce such invariants by themselves. Although these processors provide enclaves the capability of attesting themselves to other enclaves (local attestation) if requested by them, it is the programmer's job to actually ask for the attestation while switching between the modules/enclaves to ensure ordering consistency.

3 THREAT MODEL

We assume an insidious software adversary capable of subverting privileged system software (operating system or hypervisor), reconfiguring software-programmable devices (such as a DMA controller), and actively seeking to subvert a sensitive application executing on a modern "secure processor" system. The adversary is expected to attempt to mount a *privilege escalation* attack (in the context of sensitive software employing trusted and enclaved authentication/access control mechanism) in order to achieve behavior that violates the application's policy.

Denial of service, however, is not in the scope of this work, as a privileged adversary controls resource allocation, and may easily deny resources to a victim application. Also not modeled is an adversary seeking to infer private information from direct or indirect side effects of the execution of the sensitive software.

The adversary is assumed to have full control of the OS/Hypervisor, and is able to arbitrarily orchestrate system resources, computation, and launch or destroy processes including enclaves at will. The secure processor is assumed to correctly implement the enclave primitive, consisting of a process with private memory with integrity and privacy guarantees (an adversary is unable to modify the text or data of an enclave, and is unable to tamper with the execution of an enclave other than to deny it service, destroy it, or falsify untrusted inputs).

In order to compromise the sensitive application, the attacker may directly manipulate the application's control flow (e.g., by exploiting a vulnerability in the application code), or via a privileged attack whereby malicious system software violates its own process abstraction, undermining the integrity of the program and/or its data.

The application is assumed to be designed in accordance with the principles outlined in this manuscript: a modular system whereby each module is responsible for some well-defined task, e.g., collecting clients' requests, policy checking, database access etc. Following the standard model of secure processors, we consider that the system is initially in a safe state, following a trusted bootstrapping process. It is assumed that the application code is provided by a trusted software vendor, and the various enclaved code is authenticated via the enclaves' measurement (signature) *SIGSTRUCT* through the vendor's verification key *VK*. The system is assumed to correctly implement enclave measurement, whereby the measurement of a sealed enclave is descriptive of its state. In the case of SGX, incorrect or extraneous pages loaded by an adversary via *EADD* alter the enclave's measurement created by *EEXTEND* and are detectable during attestation.

In order to minimize the trusted computing base, the application designer must not enclave the entire sensitive application, and instead employ several collaborative enclaves, as described in Section 4. The reason being security concerns – as large, buggy trusted code is not trustworthy – and performance considerations since enclave memory is limited and fragmented on systems such as SGX. In a system with multiple communicating enclaves the adversary is assumed capable of tampering with the order of enclave execution, mount replay attacks, and tamper with enclave communications (by dropping, reordering, or replaying messages, as enclaves are authenticated), as described in Section 2.4 (cf. Figure 2).

This work fully trusts enclave software modules and assumes that the modules are correctly implemented. This work does not guard against software vulnerabilities (buffer overflows and other examples where an application violates its own security) in critical enclaved modules.

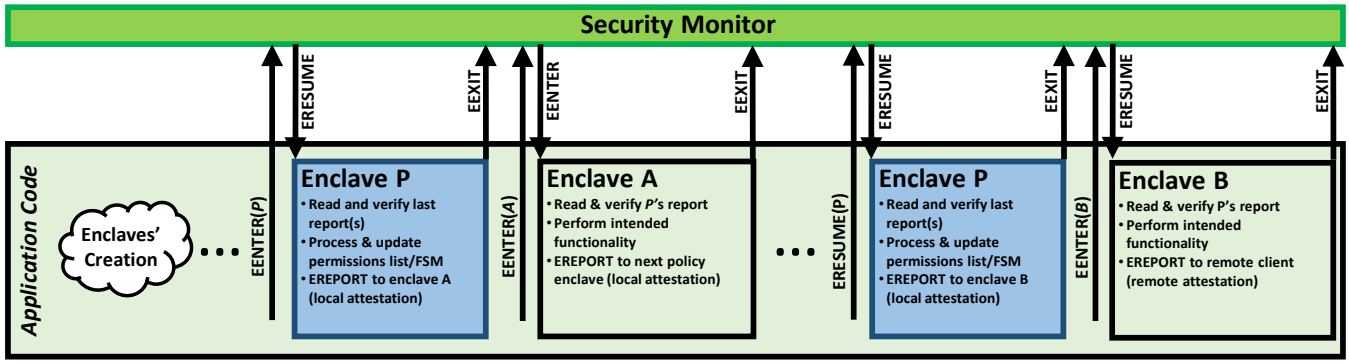

Figure 3: Application execution flow under the proposed framework on Sanctum architecture.

4 PROPOSED FRAMEWORK

Now that we have provided sufficient background regarding the functioning of secure processors and discussed our threat model, we move forward and present our proposed methodology through which these secure processors can be leveraged to offer concrete guarantees for authentication and access control schemes.

4.1 The Big Picture

4.1.1 Types of Application Enclaves. As mentioned earlier, the application designer is expected to design the application code in a modular fashion in order to perform different subtasks of the application. Since the application is designed to run on a secure processor, it is vital that all the *sensitive* modules of the application – such as authentication, access control, database accessing – are wrapped inside secure enclaves (also termed as "regular" enclaves in this paper). In our model, one specific responsibility of the programmer is to implement a centralized policy checker module for not only all the authentication and access control jobs, but also to enforce/verify the global execution ordering of various modules. This module must be wrapped inside a secure enclave, called *policy enclave*.

4.1.2 An Alternative Approach: Distributed FSM. In order to guarantee that the identity of a user/task requesting a certain resource is properly authenticated and proper access control checks are performed by the policy checker module, the policy enclave must be invoked accordingly. Moreover, since a compromised OS can control the order in which modules/enclaves are executed, it can completely bypass the policy enclave or execute it in an incorrect order to avoid the access control checks and hence perform an unauthorized access (cf. Figure 2). Since the secure processors in their current form do not enforce "correct" execution orderings of enclaves, this problem might not be detected through individual enclaves' reports. To counter this problem, the programmer may implement a "distributed" finite state machine (FSM) such that each subsequent enclave is provided with the report of the previous previous enclave in order to ensure the correct execution and enclave ordering. However, distributing the FSM implementation in this manner introduces much higher complexity for the programmer, and also makes it harder to reason about the security properties offered by such a scheme.

4.1.3 Proposed Approach: Centralized FSM. In order to address this issue and provide the programmer the flexibility of having a centralized ordering/policy checker FSM, we propose to design the application such that it always invokes the policy enclave before entering any regular enclave. The policy enclave is created at the start of the application, and being part of the application, it is created and initialized in the same fashion as the regular enclaves. A high level interaction between the policy enclave and the regular enclaves of the application is shown in Figure 3. Each time a regular enclave A is entered, the application calls the policy enclave P which is provided with a report of the last enclave executed. The policy enclave performs vital access control checks based on the available permission list, as well as verifies the provided report. A successful verification also ensures a correct enclave execution order (cf. Section 4.2 for details). Finally, P creates its own report and sends it to enclave A, which verifies it in a similar manner upon entry. Upon exit, enclave A sends its latest report to policy enclave for future use. This sequence continues until the application terminates with the execution of the final regular enclave B which attests itself to the remote client via remote attestation.

4.2 Implementation Details

Algorithm 1 and Algorithm 2 show detailed pseudo codes of the policy enclave and the regular enclave(s) under our framework respectively. Using these algorithms, any generic FSM can be mapped to our framework to keep track of the application's execution flow.

For an application with N unique regular enclaves, the policy enclave initializes two null initialized arrays Counters and Nonce each of size N through the SetupPolicyEnclave procedure which also resets its internal FSM. The application, after the initial setup, starts with calling its first regular enclave. After which, the RunPolicyEnclave is always called before running a regular enclave with an identifier $NextID$ and is provided with the report $Report$ of the previously executed regular enclave identified by id. $Report$ also contains a monotonically increasing counter c and a random nonce r (cf. Algorithm 2), the purpose of which will be explained in Section 4.3. RunPolicyEnclave calls VerifyReport that first verifies the consistency of c and r against the corresponding values stored in arrays Counters and Nonce for id, followed by updating the stored values upon successful verification. It then performs

Algorithm 1 Pseudo Code for Policy Enclave.

```
1:  procedure SetupPolicyEnclave(N)
2:      Counters := {0}^N                          ▷ Null initialized array.
3:      Nonce := {⊥}^N                             ▷ Empty nonce array.
4:      FSM-Reset( )
5:  end procedure

1:  procedure RunPolicyEnclave(Report, NextID)
2:      id ← Report                                ▷ Extract sender ID.
3:      if VerifyReport(Report)== false then
4:          Go to line 15.
5:      end if
6:      FSM-UpdateState(id)
7:      S ← FSM-GetNextStates()
8:      if (NextID ∈ S) then
9:          Nonce[NextID] ← RandomNonce
10:         r := Nonce[NextID]; c := Counters[NextID]
11:         R := EREPORT(·||r||c)
12:         SendReport(R, NextID)
13:         return
14:     end if
15:     Raise Exception.
16:     return
17: end procedure

1:  procedure VerifyReport(R)
2:      id ← R                                     ▷ Extract sender ID.
3:      r ← R                                      ▷ Extract nonce.
4:      c ← R                                      ▷ Extract counter.
5:      if (Nonce[id] ≠ r ∨ Counters[id] ≠ c) then
6:          return false
7:      else
8:          Nonce[id] ← RandomNonce
9:          Counters[id] + +
10:     end if
11:     Verify R via EGETKEY
12:     if Verification Unsuccessful then
13:         return false
14:     else
15:         return true
16:     end if
17: end procedure
```

Algorithm 2 Pseudo Code for Regular Enclaves.

```
1:  procedure SetupRegularEnclave( )
2:      Counter = 0                                ▷ Null initialized counter.
3:  end procedure

1:  procedure RunRegularEnclave(Report)
2:      id ← Report                                ▷ Extract sender ID.
3:      r ← Report                                 ▷ Extract nonce.
4:      c ← Report                                 ▷ Extract counter.
5:      if (Counter ≠ c ∨ id ≠ PolicyID) then
6:          Go to line 17
7:      end if
8:      Verify Report via EGETKEY
9:      if Verification Unsuccessful then
10:         Go to line 17
11:     end if
12:     Execute Application Functionality
13:     Counter + +
14:     R := EREPORT(·||r||c)
15:     SendReport(R, PolicyID)
16:     return
17:     Raise Exception.
18:     return
19: end procedure
```

the standard full report verification according to the secure processor's provided mechanisms. The policy enclave's FSM is then updated and the set of next valid state transitions S is identified. If the requested transition (enclave $NextID$) is allowed, then a new report is created with updated nonce and counter values and sent to $NextID$ enclave. Any failed verification during this process would raise an exception.

Each regular enclave maintains its execution count in a variable *Counter* initialized upon enclave creation by SetupRegularEnclave. RunRegularEnclave wrapper is called whenever the application wants to enter this enclave. It is also provided with a report *Report* of the policy enclave that was just executed, and this report is verified in a similar manner. Upon successful verification and after executing the intended application functionality of the enclave, a new report is created using the last nonce and counter values and sent to the policy enclave for future use. *Counter* is incremented to keep track of the execution count of the enclave.

4.3 Security of the Proposed Framework

*4.3.1 Asynchronous Exits (*AEX*).* So far we talked about enclaves voluntarily exiting, i.e., via the *EEXIT* instruction. However, the OS can swap out an enclave asynchronously whenever needed. Notice that the entry point to renter this enclave will only be available through the *ERESUME* instruction. If a compromised OS tries to execute some other enclave between *AEX* and *ERESUME*, it will fail the report verification step in RunRegularEnclave procedure since the policy enclave will not create a report for this enclave until the previously suspended enclave finishes its execution. This means that the system cannot proceed until the OS allows the suspended enclave (and the suspended enclave only) to run to completion, hence maintaining the sequential ordering of the program. The only other possibility is a denial of service attack which is out of scope of this work (cf. Section 3).

4.3.2 Preventing Replay Attacks. Since an adversary may try to replay/re-execute a regular enclave more than once with and/or without executing the policy enclave in between (i.e., to bypass the policy checker), we introduce a random nonce r in the report creation process which binds two back to back executions of a policy and regular enclave.

When a policy enclave grants permission to run a regular enclave $NextID$, it embeds a random nonce in its report sent to enclave $NextID$ and stores it a local array. When the $NextID$ enclave enters, it first expects a report from the policy enclave to be available, and

only executes if that is the case. When it finishes execution, it sends back a report to the policy enclave with the same random nonce embedded into it. Since the policy enclave expects the same random nonce, the verification succeeds.

The adversary can try the following cases:

(1) If it tries to illegitimately run a regular enclave more than once without calling the policy enclave in between, then upon the second run of the regular enclave, the report from the policy enclave will not be available since it was not run again. Same goes for multiple consecutive runs of the policy enclave.
(2) If it tries to run the combination of back to back executions of the policy and a regular enclave causing an execution flow that violates the normal behavior of the application, this would result in attempting an invalid transition in the FSM of the policy enclave. Consequently, an exception will be raised.
(3) If it tries to replay an instance of a regular enclave from an older run together with an instance of policy enclave from a newer run, with both the instances having same counter values, then the random nonce comes into the picture and causes the verification to fail.

4.3.3 Preventing Tear down of Enclaves. The compromised OS may tear down or destroy any enclave and recreate it to run from the start. In order to prevent this scenario, the policy enclave maintains a monotonically increasing counter for each regular enclave which is embedded in each report sent to the corresponding enclave. The regular enclave also maintains its execution count in an independent counter. Unless one of the two enclaves, namely the policy enclave and the regular enclave, is destroyed/reset, their *independent* counters remain consistent. As soon as one of these enclaves is destroyed/reset, the counters become inconsistent and this violation is detected. In the worst case, if the adversary resets *all* the enclaves at once, including the policy enclave, then this simply means running the whole application from start in a normal fashion.

Hence, by utilizing the isolation and attestation properties of modern secure processors, our framework allows an application programmer to design a simplified policy checker to not only perform standard authentication and access control tasks but also to verify the correct program control flow at a coarse module-level granularity. A centralized policy checker to enforce correct control flow is only made possible by the vital role played by the proposed wrappers and the design methodology for the secure enclaves.

5 APPLICATIONS AND DISCUSSION

In this section, we show and discuss a couple of applications of our proposed framework which, when combined with programmer's intelligence, can prove to be a strong authentication and access control framework. We discuss the vulnerabilities in a transaction processing system (TPS) and how such cases can be dealt with using the proposed framework. Furthermore, we extend our discussion from a TPS to a distributed system setting where multiple compute nodes communicate with each other and convey their execution states via the proposed scheme.

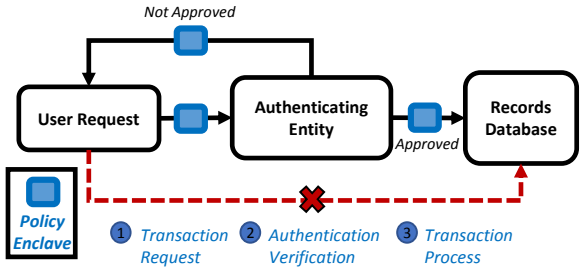

Figure 4: A Simple Transaction Processing System.

5.1 Transaction Processing System

Access control systems perform identity verification/authentication, access approval, and accountability of entities via login credentials such as passwords, personal identification numbers (PINs), biometric scans, and physical or electronic keys. In the past, authentication was almost synonymous with password systems, but today's authentication system must do more. One potential example that requires strong authentication and access control protocol can be a transaction process which should be authenticated, atomic and consistent.

Figure 4 shows a toy example of such a transaction processing system. Under normal conditions, the user inputs the request alongside its credentials to an *authenticating entity* which authenticates the user. Upon successful authentication, the user is granted permission to proceed with the request and execute a database query. The *database management system (DBMS)* processes the query and returns the corresponding result to the user.

Notice that the application flow in the transaction process is being managed by the underlying operating system. If the OS gets compromised, there can be multiple events where it can rattle the system (as shown in Figure 2) by either re-ordering the execution, or bypassing multiple steps to disrupt the process.

Our framework deals with such problematic scenarios as follows. When the user generates a request, the policy enclave verifies the *report* generated by the *user enclave*. Upon verification, the request is forwarded to the *authenticating enclave* to authenticate the user credentials. If the authentication does not succeed, the user is not granted permission to continue, and the request is terminated after the policy enclave updates its states for future requests. However, if the user authentication is successful then before accessing the database, first the policy enclave is invoked again which verifies that the last enclave executed was actually the authentication enclave. If the OS grants the user an unauthorized access to execute a database query without proper authentication (i.e., without authenticating its access permissions) then the above mentioned verification will fail and an exception will be raised. As the policy enclave enforces the correct execution order, there exists no such scenario where a compromised OS can re-order the enclave execution sequence, neither can it skip the execution of some enclaves. Hence, maintaining proper authentication and access control guarantees.

5.2 Smart Grid

Smart grid (SG) has emerged as the next generation of power grid due to its reliability, flexibility, and efficiency. In SG, the users are

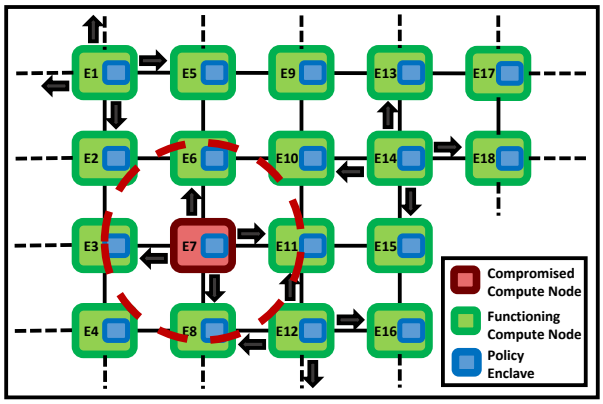

Figure 5: A distributed system with multiple compute nodes, each containing a policy checker. Logging mechanism of one node notifies the neighboring nodes about its status.

no longer passive players. Instead, they can undertake active roles to effectively minimize energy consumption by communicating back and forth with the provider. Smart grids follow the policy of *distributed systems* where work is divided among multiple compute nodes (devices). These devices communicate with each other to provide the final response to end users. Numerous machines including sensing devices, smart meters, and control systems are expected to be in between the provider and end users to facilitate this two-way communication system in SG.

In Section 5.1 we discussed how the proposed framework protects the transaction processing system via a stringent authentication and access control policy. Moving this discussion forward, consider a distributed system as shown in Figure 5. In such a system, the compute nodes communicate and interact with each other in order to achieve a common goal. If one of the nodes gets compromised, it might access some confidential information of other nodes in the system, and/or inject faulty messages into the network, consequently disrupting the whole process. In case of a smart grid, this unauthorized entity can send fabricated messages [12] to the neighborhood devices in the grid. If the messages cannot be filtered and processed, the control center might ultimately be misled and make incorrect decisions such as incorrect load balancing. Such scenarios can prove to be dreadful.

Consider that the policy enclave designed by the application developer has a "logging" functionality as well to report its system status to the neighboring nodes. Even with such a policy enclave, without enforcing the enclave execution ordering, the compromised OS might bypass the policy enclave preventing it from reporting its (compromised) status via the logging mechanism to the neighboring nodes.

On the other hand, under our framework with the correct execution ordering enforced, if the OS tries to inject new packets into the network, e.g., by executing a *network enclave*, this would require the policy enclave to be executed first (to maintain our framework's invariant). Therefore, at this point, either the the policy enclave must be executed first – which would result in reporting the compromised status log to the neighbors – or no "faulty" packets can be injected into the network. Communicating the log to the neighbors would allow them to suspend/ignore their future communications with the compromised node until that node is recovered.

5.3 MapReduce

When processing distributed data sets of extreme size, MapReduce [7], as exemplified by *Apache Hadoop*, is a commonly used framework, as it offers a productive abstraction for large-scale multiprocessor data processing by hiding the complexity associated with fault tolerance, data movement, and work orchestration in a large computing environment.

The MapReduce framework is organized into three distinct operations: Map (a programmer-supplied program that emits key-value pairs), a Shuffle (whereby the mapped pairs are grouped by key), and Reduce (a programmer-supplied script that processes the values grouped by key). A canonical "big data" example of MapReduce is a word count application: Map processes chunks of text input and emits (*word*, 1) for each word in the input, and after Shuffle groups these pairs by key, Reduce counts the number of values associated with each key to obtain a word count.

Consider a software adversary that seeks to subvert the execution. While an enclaved system cannot defend against a privileged denial of service, an attacker may seek to bias the computation by suppressing or falsifying keys during a shuffle operation, or subverting the map or reduce nodes. The MapReduce controller (scheduler) is an unequivocally trusted component: the controller parcels out work to various nodes within a computing cluster, and orchestrates the communication streams (shuffle) between Map and Reduce nodes. Enclaving the controller is not alone sufficient to guard the MapReduce framework against a software adversary, however, as both the messages sent by the controller, and the mapper and reducer nodes must be authenticated and tied to trusted enclaved code. Even with individual nodes enclaved and therefore resistant to an adversary's attempts to tamper with the computation, MapReduce may be vulnerable to a capable adversary suppressing some key-value pairs during a shuffle operation, or performing replay attacks.

Consider the policy enclave implementing the functions of a MapReduce controller, parceling out shares of input data to (enclaved) mapper nodes, which 1). authenticate the controller's messages, and 2) produce a certificate attesting to the integrity of the result of the Map or Reduce operation. A software adversary is unable to defeat the integrity of the MapReduce controller (the policy enclave), and therefore cannot falsify the input partitioning or shuffle. Should an adversary tamper with Map or Reduce enclaves, they would either not run or not produce a certificate that authenticates their output, either being a failure, which the MapReduce framework is designed to tolerate.

A prudent system designer can extend this sketch to implement an augmented MapReduce, where the the mapper and reducer nodes enforce differential privacy [9], or aggregate secret datasets on behalf of multiple mutually distrusting entities. The system composed of enclaves orchestrated and authenticated by the policy enclave acting as a MapReduce controller offers a trusted, authenticated party that thwarts capable software adversaries and enforces high-level policies.

5.4 Discussion

Our framework opens up several other possibilities for richer application design. We discuss two of the possible improvements that can be made to the applications under this model.

Since the dynamic execution flow is guaranteed to be consistent with the application designer's intended control flow, one can extend the policy enclave to collect "behavioral" information about the regular enclaves running in the system. With some standard machine learning algorithms, the policy checker can be enhanced to incorporate multiple behavioral authentication factors resulting in dynamically adjusting to the program's behavior at run time. This can also be used to develop intelligent anomaly detection mechanisms. For instance, if a user/enclave starts reading unusually high amount of records from a database, this could potentially mean that this user is trying to replicate or clone the database and should be prevented from doing so.

The policy enclave can also be enriched by incorporating a logging mechanism for a distributed system setting. With our model, for example, the *stealthy logging* mechanism from [2] can potentially be simplified since the adversary (e.g., the compromised OS) cannot directly access the reserved memory of the policy enclave where the system's log is stored.

6 CONCLUSION

We have proposed a software application design methodology that leverages the software isolation container (enclave) primitive offered by modern secure processors to enforce correct flow of execution within a sensitive application. With no required changes to modern hardware, our methodology allows the application designer to securely perform authentication and access control. We provide a comprehensive discussion of security of the proposed scheme in the presence of a privileged software adversary. In addition to thwarting an adversary attempting to circumvent application policy checks, our framework offers the application designer a global view of application state via the policy checker enclave, which in turn enables high-level policies to be described and tailored for a given application.

ACKNOWLEDGMENTS

The work is partially supported by NSF grants CNS-1413920 and CNS-1413996 for MACS: A Modular Approach to Cloud Security.

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*.

[2] Kevin D Bowers, Catherine Hart, Ari Juels, and Nikos Triandopoulos. 2014. Pillarbox: Combating next-generation malware with fast forward-secure logging. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 46–67.

[3] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*.

[4] David Champagne and Ruby B Lee. 2010. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 1–12.

[5] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. *ACM SIGARCH Computer Architecture News* 36, 3 (2008).

[6] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[8] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 103–114.

[9] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*. 1–12. https://doi.org/10.1007/11787006_1

[10] Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis)*. 3–8.

[11] Hugo Gamboa and Ana Fred. 2004. A behavioral biometric system based on human-computer interaction. In *Defense and Security*. International Society for Optics and Photonics, 381–392.

[12] H. Li, R. Lu, L. Zhou, B. Yang, and X. Shen. 2014. An Efficient Merkle-Tree-Based Authentication Scheme for Smart Grid. *IEEE Systems Journal* 8, 2 (June 2014), 655–663. https://doi.org/10.1109/JSYST.2013.2271537

[13] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. 2003. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[14] D. Lie, C. Thekkath, and M. Horowitz. 2003. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. 178–192.

[15] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. 168–177.

[16] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 311–324.

[17] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution.. In *HASP@ ISCA*. 10.

[18] Santosh Nagarakatte, Milo Martin, and Stephan A Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th International Symposium on Computer Architecture*.

[19] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[20] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 3 (2005), 477–526.

[21] Elaine Shi, Yuan Niu, Markus Jakobsson, and Richard Chow. 2010. Implicit authentication through learning user behavior. In *International Conference on Information Security*. Springer, 99–113.

[22] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of the* 17^{th} *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*. ACM, New-York. http://csg.csail.mit.edu/pubs/memos/Memo-474/Memo-474.pdf(revisedone)

[23] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. 2005. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *Proceedings of the* 32^{nd} *ISCA'05*. ACM, New-York. http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf

[24] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 457–468.

[25] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P'15*.

[26] Larry Zhu, Sam Hartman, and Karthik Jaganathan. 2005. The Kerberos Version 5 Generic Security Service Application Program Interface (GSS-API) Mechanism: Version 2. (2005).