Verification of Confidentiality Properties of Enclave Programs

Rohit Sinha
Sriram Rajamani
Sanjit A. Seshia
Kapil Vaswani

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-162
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-162.html

June 1, 2015
Verification of Confidentiality Properties of Enclave Programs

Rohit Sinha  
University of California, Berkeley  /rsinha@berkeley.edu

Sanjit A. Seshia  
University of California, Berkeley  /sseshia@berkeley.edu

Sriram Rajamani  
Microsoft Research, India  
sriram@microsoft.com

Kapil Vaswani  
Microsoft Research, India  
kapilv@microsoft.com

ABSTRACT

Security-critical applications running in the cloud constantly face threats from exploits in lower computing layers such as the operating system, virtual machine monitors, or even attacks from malicious datacenter administrators. To help protect application secrets from such attacks, there is increasing interest in hardware implementations of primitives for trusted computing, such as Intel’s Software Guard Extensions (SGX). These primitives enable hardware protection of memory regions containing code and data, root of trust for measurement, remote attestation, and cryptographic sealing. However, vulnerabilities in the application itself (e.g. incorrect use of SGX instructions, memory safety errors) can be exploited to divulge secrets. We introduce Moat, a tool which formally verifies confidentiality properties of applications running on SGX. We create formal models of relevant aspects of SGX, develop several adversary models, and present a verification methodology for proving that an application running on SGX does not contain a vulnerability that causes it to reveal secrets to the adversary. We evaluate Moat on several applications, including a one time password scheme, off-the-record messaging, notary service, and secure query processing.

1. INTRODUCTION

Building applications that do not leak secrets (i.e., satisfying confidentiality) with both client devices and cloud services as components is non-trivial. There are at least three kinds of attacks a developer must guard against. The first kind of attack, which we call protocol attack, is due to vulnerabilities in the cryptographic protocol used to establish trust between various distributed components. Examples of protocol attacks include man-in-the-middle or replay attacks. The second kind of attack, which we call application attack, is due to errors or vulnerabilities in the application code itself which can be exploited to leak confidential information from the application (e.g. Heartbleed bug [14]). The third kind of attack, which we call infrastructure attack, is due to exploits in the software stack (e.g. operating system (OS), hypervisor) that the application relies upon, where the privileged malware controls the CPU, memory, I/O devices, etc. Infrastructure attacks can result in an attacker gaining control of the application’s memory and reading secrets at will.

Several mitigation strategies have been proposed for each of these kinds of attacks. In order to guard against protocol attacks, we can use protocol verifiers (e.g., ProVerif [8], CryptoVerif [9]) to check for protocol errors. In order to guard against application attacks, the application can be developed in a memory-safe language with information flow control, such as Jif [22]. Infrastructure attacks are the hardest to protect against, since the attack can happen even if the application is error free (i.e., without application vulnerabilities or protocol vulnerabilities). While some efforts are under way to build a fully verified software stack ground-up (e.g. [16, 19]), this approach is unlikely to scale to real-world OS and system software. An alternative approach to guarding against infrastructure attack is by hardware features that enable a user-level application to be protected from privileged malware. For instance, Intel SGX [17, 18] is an extension to the x86 instruction set architecture, which provides any application the ability to create protected execution contexts called enclaves containing code and data. SGX features include 1) hardware-assisted isolation from privileged malware for enclave code and data, 2) measurement and attestation primitives for detecting attacks on the enclave during creation, and 3) sealing primitives for storing secrets onto untrusted persistent storage. If these primitives are used correctly, an application can reduce its trusted computing base to only the enclave code and SGX processor, thereby defending against infrastructure attacks.

Although SGX has the potential to protect against infrastructure attacks, the developer must still take care to use SGX primitives correctly, use safe cryptographic protocols, avoid traditional bugs due to memory safety violations, etc. For instance, the enclave may suffer from exploits like Heartbleed exploits by using vulnerable SSL implementations. Similarly, by excluding some component of enclave state from the measurement, the enclave may suffer from masquerading attacks. With a compromised OS acting as the adversary, the adversary can modify non-enclave memory, modify page tables, generate interrupts, modify network messages, etc. Developers find it non-trivial to write secure enclaves because they must account for all potential enclave behaviors in the presence of such adversarial actions. Currently, there is no formal adversary model or technique for reasoning about enclave behaviors in the presence of an active adversary. This paper takes a first step to solving this problem. We explore the contract between the SGX hardware and the enclave developer, formalize various adversary models, and implement a verification methodology that helps developers find vulnerabilities in enclave code that can be exploited via protocol, application, and infrastructure attacks. This research makes the following specific contributions:

- We develop the first formal model of the SGX platform and its new instruction set, working from publicly-available
documentation [18].

- We formalize several adversary models ranging from passive to active privileged adversaries. We present a sound method of composing the adversary and the enclave program, where the composition encodes all potential runtime behaviors of the enclave in the presence of an active adversary. To enable efficient verification, we show how a very general active adversary can be reduced to a much simpler one without loss of soundness.

- We develop Moat, a system for statically verifying security properties of an enclave program in the face of application and infrastructure attacks. More precisely, we formalize confidentiality (ignoring side channel leaks) for the instruction-level behavior of enclave programs. Moat employs a flow- and path-sensitive type checking algorithm (based on satisfiability modulo theories solving [5]) for automatically verifying whether an enclave program (composed with the adversary) provides confidentiality guarantees.

Though we study these issues in the context of Intel SGX, similar issues arise in other architectures based on trusted hardware such as ARM TrustZone [2] and Sancus [23], and our approach is potentially applicable to them as well. The theory we develop with regard to attacker models and our verifier is mostly independent of the specifics of SGX, and our use of the term “enclave” is also intended in the more general sense.

2. BACKGROUND

We demonstrate the use of SGX by an example of a one-time password (OTP) service, although the exposition extends naturally to any secret provisioning protocol. OTP is typically used in two factor authentication as an additional step to traditional knowledge based authentication via username and passphrase. A user demonstrates ownership of a pre-shared secret by providing a fresh, one-time password that is derived deterministically from that secret. For instance, RSA SecurID® is a hardware-based OTP solution, where possession of a tamper-resistant hardware token is required during login. In this scheme, a pre-shared secret is established between the OTP service and the hardware token. From then on, they compute a fresh one-time password as a function of the pre-shared secret and time duration since the secret was provisioned to the token. The user must provide the one-time password on the token during authentication, in addition to her username and passphrase. This OTP scheme is both expensive and inconvenient because it requires distributing tamper-resistant hardware tokens physically to the users. Although pure software implementations have been attempted, they are often prone to infrastructure attacks from malware, making them untrustworthy. Thus, it is natural to see if we can build OTP using new hardware platforms such as SGX, which guard against infrastructure attacks by providing primitives for measurement, attestation, and memory protection. Hoeleka et al. [17] propose a OTP scheme on SGX, which we implement (see Figure 1) and verify using Moat. Consider the following protocol that a bank OTP server uses to provision the pre-shared secret to a client, which is running on a potentially infected machine with SGX hardware:

1. The server sends the client an attestation challenge nonce. Consequent messages in the protocol use the nonce to guarantee freshness.

2. The client and OTP server engage in an authenticated Diffie-Hellman key exchange in order to establish a symmetric session_key. The client uses ereport instruction to send a report containing a signature over the Diffie-Hellman public key dh_pubkey and the enclave’s measurement. The signature guarantees that the report was generated by an enclave on SGX, while the measurement guarantees that the reporting enclave is an untampered OTP enclave. After verifying the signatures, both the client and OTP server compute the symmetric session_key.

3. The OTP server sends the pre-shared OTP secret to the client by first encrypting it with the session_key, and then signing the encrypted content with the bank’s private TLS key. The client verifies the signature and decrypts the message to retrieve the pre-shared otp_secret.

4. For future use, the client requests for sealing_key (using getkey instruction), encrypts otp_secret using sealing_key, and writes the sealed_secret to disk.

The basic necessities for implementing this protocol securely are (1) ability to perform the cryptographic operations (or any trusted computation) without interference from the adversary, (2) protected memory for computing and storing secrets, and (3) root of trust for measurement and attestation. Intel SGX processors provide these primitives.

The Intel SGX instructions allow a user-level application to instantiate a protected execution context, called an enclave, containing both code and data. The enclave memory resides within the application’s virtual address space, but is protected from accesses by any privileged software — only the enclave code is allowed to access enclave memory. It is for this reason that we store secrets (session_key, sealing_key, and otp_secret) in enclave heap and implement cryptographic operations in enclave code (Figure 1). It is important to clarify that SGX does not protect the enclave program from inadvertently leaking secrets, thus necessitating a verifier for enclave programs. For instance, the enclave in Figure 1 contains a vulnerability that we describe in § 3.1.

The untrusted host application creates an enclave using a combination of SGX instructions: ecreate, eadd, eextend, and einit. The host application invokes ecreate to reserve protected memory for enclave use. To populate the enclave with code and data, the host application uses a sequence of eadd and eextend instructions. eadd loads code and data pages from non-enclave memory to enclave’s reserved memory. eextend extends the current enclave measurement with the measurement of the newly added page. Finally, einit terminates the initialization phase, which prevents any further modification to the enclave state from non-enclave code.

The reader may observe that code and data must be distributed in cleartext, opening it up to eavesdropping and tampering by adversaries. For instance, an adversary may modify a OTP enclave’s binary (on user’s machine) so that it leaks a user’s login credentials. SGX provides measurement primitive eextend and attestation primitive ereport to defend against this class of attacks. The OTP server uses remote attestation to prove that the user is running the same enclave as the one distributed by the bank. The enclave participates in attestation by invoking ereport, which generates a hardware-signed report of the enclave’s measurement. The enclave can also use ereport to bind data to its measurement, thereby adding authenticity to that data (e.g. authenticated Diffie-Hellman exchange in Figure 1). After verifying the attestation report,
the OTP server may choose to provision the otp_secret to the enclave. For future access, the enclave may use egetkey to attain a hardware-generated sealing key, and store the sealed secret to untrusted storage.

A typical application (such as our OTP client) uses enclave code to implement cryptographic operations or any other trusted computation. However, enclave code is not allowed to invoke any privileged instructions or even system calls, forcing the enclave to rely on non-enclave code to issue system calls, perform I/O, etc. For instance, to send the Diffie-Hellman public key to the server, enclave (1) invokes ereport with enclave_state.dh_pubkey, (2) copies the report to non-enclave memory app_heap, (3) invokes eexit to transfer control to the app, and (4) waits for app to invoke the socket system calls to send the report to the bank server. For this reason, app and enclave perform a sequence of enter and eexit to transfer control back and forth. The host application transfers control to the enclave by invoking enter (at some point after einit), which transfers control to the enclave’s entry point. The enclave executes atomically until one of the following events occur: (1) enclave code invokes eexit to transfer control to the host application, (2) enclave code incurs a fault or exception (e.g., page fault, divide by 0 exception, etc.), and (3) the CPU receives a hardware interrupt and transfers control to a privileged interrupt handler. In the case of faults, exceptions, and interrupts, the CPU saves state (registers, etc.) in State Save Area (SSA) pages, which is a region in enclave memory dedicated to saving CPU state during such events. Although this design makes an enclave vulnerable to denial of service attacks, we show that an enclave can still guarantee safety properties such as data confidentiality.

Confidentiality requires protecting secrets, which requires understanding of the contract between the enclave developer and the SGX hardware. First, the enclave developer must follow the enclave creation guidelines (see §3.2) so that the hardware protects the enclave from an attacker that has gained privileged access to the system. Even then, the enclave developers need to ensure that their code does not leak secrets via application attacks and protocol attacks. For instance, they should encrypt secrets before writing them to non-enclave memory. They should account for modifying non-enclave memory at any time, which could result in time-of-check-to-time-of-use attacks. Writing secure enclaves is non-trivial and is a topic that we explore formally in this paper. To our knowledge, there does not exist any formal method for reasoning about attacks or vulnerabilities in enclaves, or for formally proving security properties such as confidentiality.

3. OVERVIEW OF MOAT

We are interested in building secure distributed applications, which have components running in trusted and untrusted environments, where all communication channels are untrusted. For the application to be secure, we need (1) secure cryptographic protocols between the components (to protect from protocol attack), and (2) secure implementation in each component to protect from application attack and infrastructure attack. Moat focuses on application and infrastructure attacks. The adversary model used in Moat allows privileged malware to arbitrarily update non-enclave memory, generate interrupts, modify page tables, or launch other enclaves. Our goal is to prove that, even in the presence of such an adversary, the enclave does not leak its secrets to the adversary.

3.1 Protecting from Application Code Attack

We give an overview of our verifier, Moat, for proving confidentiality of a single enclave’s implementation (detailed exposition in §4 through §7). Moat accepts an enclave program in x86 Assembly, containing SGX instructions ereport, egetkey, and eexit. Moat is also given a set of annotations, called Secrets, indicating which component of state holds secret values. In the OTP example, the Secrets are otp_secret, session_key, and sealing_key. Moat proves that a privileged software adversary running on the same machine does not observe a value that depends on Secrets, regardless of any operations performed by that adversary. We demonstrate Moat on a snippet of OTP enclave code containing lines 22-26 from Figure 1, which are first compiled to x86+SGX Assembly (Figure 2). Here, the enclave invokes egetkey to retrieve a 128-
bit sealing key, which is stored in the byte array `sealed_secret`.

Next, the enclave encrypts `otp_secret` (using AES-GCM-128 encryption library function called `encrypt`) to compute the `sealed_secret`. Finally, the enclave copies `sealed_secret` to untrusted memory `app_heap` (to be written to disk). Observe that the size argument to `memcpy` (line 26 in Figure 1) is a variable `size_field` which resides in non-enclave memory. This makes the enclave vulnerable to application attacks because the adversary can cause the enclave to leak secrets from the stack.

```
movl $0x701000,(%esp)
mov %eax,0x4(%esp)
lea -0x6e0(%ebp),%eax
movl $0x8080AC,0x8(%esp)
egetkey
```

The size of `sealed_secret` of `otp_secret` now that writing contents, which includes the `mem` argument to `memcpy` (line 26 of Figure 3), manifests as a `size_field` which resides in non-enclave memory. This makes the enclave vulnerable to application attacks because the adversary can cause the enclave to leak secrets from the stack.

```
movl $0x701000,(%esp)
mov %eax,0x4(%esp)
lea -0x6e0(%ebp),%eax
movl $0x8080AC,0x8(%esp)
egetkey
```

Figure 2: OTP enclave snippet (left) and `penc` (right)

To reason about enclave code and find such vulnerabilities, Moat first extracts a model in an intermediate verification language, as shown in Figure 2. We refer to the model as `penc`. `penc` models x86 (e.g. `load`, `store`) and SGX (e.g. `egetkey`) instructions as uninterpreted functions constrained with axioms. The axioms (presented in § 4.2) are part of our machine model, and they encode the ISA-level semantics of each instruction. `penc` models x86 semantics precisely, including updates to CPU flags. For brevity, Figure 2 omits all updates to flags as they are irrelevant to this code snippet.

Since `penc` executes in the presence of an active adversary, we must model the effects of adversarial operations on `penc`’s execution. Section 5 defines an active adversary `H`, which can perform the operation “`havoc mem`” once between consecutive instructions along any execution of `penc`. Here, `mem` denotes memory reserved by the SGX processor for enclave use, and `mem` is all other memory; `havoc mem` updates each address in `mem` with a non-deterministically chosen value. We define `H` this way because a privileged software adversary can interrupt `penc` at any point, perform `havoc mem`, and then resume `penc`. We model the effect of `H` on `penc`’s behavior by instrumenting `penc` to obtain `penc -H` (see Figure 3).

The OTP enclave implementation is vulnerable. The size argument to `memcpy` (line 26 in Figure 1) is a field within a data structure in non-enclave memory. This vulnerability manifests as a `load` (line 8 of Figure 3), which reads a value from non-enclave memory and passes that value as the size argument to `memcpy`. To perform the exploit, `H` uses `havoc mem` (in line 8) to choose the number of bytes that `penc -H` writes to non-enclave memory, starting at the base address of `sealed_secret`. By setting this value to be greater than the size of `sealed_secret`, `H` causes `penc -H` to leak the stack contents, which includes the `sealing_key`. We can assume for now that writing `sealed_secret` to `app_heap` is safe because it is encrypted using a secret key. We now explain how `Moat` evaluates whether a particular enclave output is safe. As a pragmatic choice, `Moat` does not reason about cryptographic operations for there is significant body of research on cryptographic protocol verification. For instance, if encryption uses a key established by Diffie-Hellman, `Moat` needs to reason about the authentication

Figure 3: `penc -H` constructed using OTP `penc`

pair of traces of `penc -H` that differ in the values of `Secrets`, if `H`’s operations along the two traces are equivalent, then `H`’s observations along the two traces must also be equivalent.

Our type system checks confidentiality by instrumenting `penc -H` with ghost variables that track the flow of `Secrets` within registers and memory, akin to taint tracking but performed using static analysis. Figure 4 demonstrates how `Moat` type-checks `penc -H`. For each state variable `x`, the type system instruments a ghost variable `C_x`. `C_x` is updated on each assignment that updates `x`, and is assigned to `false` only if `x`’s value is independent of `Secrets` (details in § 7). For instance, `C_mem[esp]` in line 13 is assigned to `false` if `C_mem[esp]` is true, because ghost value or secret address should cause the stored value to be a secret. Furthermore, for each secret in `Secrets`, we set the corresponding locations in `C_mem` to true. For instance, lines 1-3 assign true to those 16 bytes in `C_mem` where `egetkey` places the secret `sealing_key`. Information leaks can only happen via `store` to `mem-enc`, where `mem-enc` is a subset of `mem` that is reserved for use by `penc`, and `mem-enc` is either non-enclave memory or memory used by other enclaves. `enc(i)` is true if `i` is an address in `mem-enc`. For each `store` instruction, the type system instruments an `assert` checking that a secret value is not written to `mem-enc`. For a program to be well-typed, all assertions in the instrumented `penc -H` must be valid along any feasible execution. `Moat` feeds the instrumented program (Figure 4) to a static program verifier [3], which explores all feasible executions (i.e. all reachable states) of the enclave and checks that the assertions are valid along such executions. In Figure 4, the assertion in line 29 is invalid because `C_mem` is true for memory locations that hold the `sealing_key`. Our type system rejects this enclave program. Although memory safety vulnerabilities can be found using simpler approaches, `Moat` can identify many classes of vulnerabilities using these typing assertions. A fix to the OTP implementation is to replace `size_field` with the correct size, which is 64 bytes.

Declassification.

In the previous section, we make the claim that writing `sealed_secret` to `app_heap` is safe because it is encrypted using a secret key. We now explain how `Moat` evaluates whether a particular enclave output is safe. As a pragmatic choice, `Moat` does not reason about cryptographic operations for there is significant body of research on cryptographic protocol verification. For instance, if encryption uses a key established by Diffie-Hellman, `Moat` needs to reason about the authentication
and attestation scheme used in that Diffie-Hellman exchange in order to derive that the key can be safely used for encryption. When Moat encounters a cryptographic library call, it abstracts it as an uninterpreted function with the conservative axiom that secret inputs produce secret output. For instance in Figure 4, aes_gcm_encrypt on line 16 is an uninterpreted function, and C_aes_gcm_encrypt on line 15 marks the ciphertext as secret if any byte of the plaintext or encryption key is secret. Clearly, such conservative axiomatization is unnecessary because a secret encrypted with a key unknown to the adversary can be safely output. To reduce this imprecision in Moat, we introduce declassification to our type system. A declassified output is a safe, intentional information leak of the program, which may be manually annotated or proven safe using other means. In our experiments, we safely eliminate declassified outputs from information leakage checking if the protocol verifier has already proven them to be safe outputs. The choice of protocol verifier is orthogonal to our work.

To collect the Declassified annotations, we manually model the cryptographic protocol to verify using an off-the-shelf protocol verifier (e.g. ProVerif [8], CryptoVerif [9]). A protocol verifier accepts as input an abstract model of the protocol (in a formalism such as pi calculus), and proves properties such as confidentiality. We briefly describe how we use Moat in tandem with a protocol verifier. If Moat establishes that a particular value generated by p_enc is secret, this can be added to the set of secrecy assumptions made in the protocol verifier. Similarly, if the protocol verifier establishes confidentiality even while assuming that a p_enc’s output is observable by the adversary, then we can declassify that output while verifying p_enc with Moat. This assume-guarantee reasoning is sound because the adversary model used by Moat can simulate a network adversary — a network adversary reorders, inserts, and deletes messages, and the observable effect of these operations can be simulated by a havoc mem一直都s.

We demonstrate this assume-guarantee reasoning on lines 22-26 of the OTP enclave in Figure 1, where line 26 no longer has the memory safety vulnerability i.e. it uses the constant 64 instead of size_field. Despite the fix, Moat is unable to prove that memcpy in line 26 is safe because its axiomatization of aes_gcm_encrypt prevents the derivation that the ciphertext is non-secret. We proceed by first proving in Moat that the sealing_key is not leaked to the adversary. Next, we annotate the ProVerif model with the assumption that sealing_key is secret, which allows ProVerif to prove that the outbound message (via memcpy) is safe. Based on this ProVerif proof, we annotate the sealed_secret as Declassified, hence telling Moat that the assert on line 29 of Figure 4 is valid.

Figure 4: $p_{enc}$ instrumented with typing assertions

and attestation scheme used in that Diffie-Hellman exchange in order to derive that the key can be safely used for encryption. When Moat encounters a cryptographic library call, it abstracts it as an uninterpreted function with the conservative axiom that secret inputs produce secret output. For instance in Figure 4, aes_gcm_encrypt on line 16 is an uninterpreted function, and C_aes_gcm_encrypt on line 15 marks the ciphertext as secret if any byte of the plaintext or encryption key is secret. Clearly, such conservative axiomatization is unnecessary because a secret encrypted with a key unknown to the adversary can be safely output. To reduce this imprecision in Moat, we introduce declassification to our type system. A declassified output is a safe, intentional information leak of the program, which may be manually annotated or proven safe using other means. In our experiments, we safely eliminate declassified outputs from information leakage checking if the protocol verifier has already proven them to be safe outputs. The choice of protocol verifier is orthogonal to our work.

To collect the Declassified annotations, we manually model the cryptographic protocol to verify using an off-the-shelf protocol verifier (e.g. ProVerif [8], CryptoVerif [9]). A protocol verifier accepts as input an abstract model of the protocol (in a formalism such as pi calculus), and proves properties such as confidentiality. We briefly describe how we use Moat in tandem with a protocol verifier. If Moat establishes that a particular value generated by p_enc is secret, this can be added to the set of secrecy assumptions made in the protocol verifier. Similarly, if the protocol verifier establishes confidentiality even while assuming that a p_enc’s output is observable by the adversary, then we can declassify that output while verifying p_enc with Moat. This assume-guarantee reasoning is sound because the adversary model used by Moat can simulate a network adversary — a network adversary reorders, inserts, and deletes messages, and the observable effect of these operations can be simulated by a havoc mem一直都s.

We demonstrate this assume-guarantee reasoning on lines 22-26 of the OTP enclave in Figure 1, where line 26 no longer has the memory safety vulnerability i.e. it uses the constant 64 instead of size_field. Despite the fix, Moat is unable to prove that memcpy in line 26 is safe because its axiomatization of aes_gcm_encrypt prevents the derivation that the ciphertext is non-secret. We proceed by first proving in Moat that the sealing_key is not leaked to the adversary. Next, we annotate the ProVerif model with the assumption that sealing_key is secret, which allows ProVerif to prove that the outbound message (via memcpy) is safe. Based on this ProVerif proof, we annotate the sealed_secret as Declassified, hence telling Moat that the assert on line 29 of Figure 4 is valid.

The remainder of this paper describes our verification approach for defending against application attacks, which is the focus of this paper. Moat takes a binary enclave program and proves confidentiality i.e. it does not leak secrets to a privileged adversary. In order to construct proofs about enclave behavior, we first model the enclave’s semantics in a formal language that is amenable to verification, and also model the effect of adversarial operations on enclave behavior. This section describes (1) formal modeling of enclave programs, (2) formal model of the x86+SGX instruction set, and (3) formal modeling of active and passive adversaries.

4. FORMAL MODEL OF THE ENCLAVE PROGRAM AND THE ADVERSARY

The remainder of this paper describes our verification approach for defending against application attacks, which is the focus of this paper. Moat takes a binary enclave program and proves confidentiality i.e. it does not leak secrets to a privileged adversary. In order to construct proofs about enclave behavior, we first model the enclave’s semantics in a formal language that is amenable to verification, and also model the effect of adversarial operations on enclave behavior. This section describes (1) formal modeling of enclave programs, (2) formal model of the x86+SGX instruction set, and (3) formal modeling of active and passive adversaries.
4.1 Syntax and Semantics of Enclave Programs

Our model of a x86+SGX machine consists of an unbounded number of Intel SGX CPUs operating with shared memory. Although SGX allows an enclave to have multiple CPU threads, we restrict our focus to single-threaded enclaves for simplicity, and model all other CPU threads as running privileged adversarial code. A CPU thread is a sequence of x86+SGX instructions. In order to reason about enclave execution, Moat models the semantics of all x86+SGX instructions executed by that enclave. This section describes Moat's translation of x86+SGX Assembly program to a formal model, called $p_{enc}$, as seen in Figure 2.

Moat first uses BAP [11] to lift x86 instructions into a simple microarchitectural instruction set: load from mem, store to mem, bitwise (e.g. xor) and arithmetic (e.g. add) operations on regs, conditional jumps cjmp, unconditional jumps jmp, and usermode SGX instructions (ereport, egotkey, and eexit). We choose BAP for its precise modeling of x86 instructions, usermode SGX instructions (ereport, egotkey, and eexit). In section 4.1, we lifted x86 to a microarchitectural instruction sequence, and modeled each microarchitectural instruction as an uninterpreted function (e.g. xor, load, ereport). In this section, we add axioms to these uninterpreted functions in order to model the effect of instructions on machine state.

A state $\sigma$ is a valuation of all Vars, which consists of mem, isa-visible CPU registers regs, etc., basically the state components that we choose to include in our x86+SGX machine model. We define the state $\sigma \in \Sigma$ of $p_{enc}$ at a given program location to be a valuation of all variables in Vars. The semantics of a BoogiePL statement $s \in Stmt$ is given by a relation $R(s) \subseteq \Sigma \times \Sigma$ over pairs of pre and post states, where $(\sigma, \sigma') \in R(s)$ if and only if there is an execution of $s$ starting at $\sigma$ and ending in $\sigma'$. We use standard axiomatic semantics for each Stmt in Figure 5 [4].

Enclaves have an entroypoint which is configured at compile time and enforced at runtime by a callgate-like mechanism. Therefore, Moat makes BAP perform a walk over the control flow graph, starting from the entroypoint, while translating x86+SGX instructions to microarchitectural instructions. Procedure calls are either inlined or abstracted away as uninterpreted functions. Specifically, trusted library calls (e.g. AES-GCM authenticated encryption) are abstracted as uninterpreted functions with standard axioms — the cryptographic library is in our trusted computing base. Furthermore, Moat soundly unrolls loops to a bounded depth by adding an assertion that any iteration beyond the unrolling depth is unreachable. Our $p_{enc}$ model is sound under the following assumptions: (1) control flow integrity, (2) code regions are not modified (enforced using page permissions), (3) the trusted cryptographic library calls return to the instruction following the call, and (4) the trusted cryptographic library implementation is memory safe.

By bounding the number of loop iterations and recursion depth, the resulting verification problem becomes decidable, and one that can be checked using a theorem prover. Several efficient techniques [4] transform this loop-free and call-free procedure containing assertions into a compact logical formula in the Satisfiability Modulo Theories (SMT) format by a process called verification-condition generation. This formula is valid if and only if $p_{enc}$ does not fail any assertion in any execution — validity checking is done by an automated theorem prover based on SMT solving [13]. In the case of assertion failures, the SMT solver also constructs a counter-example execution of $p_{enc}$ demonstrating the assertion failure. In § 7, we show how Moat uses assertions and verification-condition generation to prove confidentiality properties of $p_{enc}$.

$$x, X \in \text{Vars}$$

$$q \in \text{Relations}$$

$$f, g, h \in \text{Functions}$$

$$e \in \text{Expr} \ ::= \ x \mid X \mid [\text{e}] \mid f(\text{e}, \ldots, \text{e})$$

$$\phi \in \text{Formula} \ ::= \text{true} \mid \text{false} \mid \text{e} \equiv \text{e} \mid q(\text{e}, \ldots, \text{e}) \mid \phi \land \phi \mid \neg \phi$$

$$s \in \text{Stmt} \ ::= \text{skip} \mid \text{assert} \phi \mid \text{assume} \phi \mid X := e \mid X := [\text{e}] \mid \text{if (e) } \{s\} \text{ else } \{s\} \mid s; s$$

Figure 5: Syntax of programs.

4.2 Formal Model of x86 and SGX Instructions

While formal models of x86 instructions using BoogiePL has been done before (see for instance [27]), we are the first to model SGX instructions. In section 4.1, we lifted x86 to a microarchitectural instruction sequence, and modeled each microarchitectural instruction as an uninterpreted function (e.g. xor, load, ereport). In this section, we add axioms to these uninterpreted functions in order to model the effect of instructions on machine state.

A state $\sigma$ is a valuation of all Vars, which consists of mem, isa-visible CPU registers regs, and epcm. As their names suggest, physical memory mem is modeled as an unbounded array, with index type of 32 bits and element type of 8 bits. mem is partitioned by the platform into two disjoint regions: protected memory for enclave use (memapc), and untrusted memory (memapm). For any physical address $a$, epcm(a) is true if $a$ is an address in memapc. Furthermore, memenc is a subset of memapc that is reserved for use by $p_{enc}$. memenc is virtually addressed and it belongs to the host application’s virtual address space. For any virtual address $a$, enc(a) is true if $a$ resides in memenc. The epcm is a finite sized array of hardware-managed structures, where each structure stores security critical metadata about a page in memapc. epcm is a subset of epcm that stores metadata about each page in memenc — other epcm structures are either free or in use by other enclaves. regs is the set of ISA-visible CPU registers such as rax, rbp, etc.

Each microarchitectural instruction in $p_{enc}$ has side-effects on $\sigma$, which we model using axioms on the corresponding uninterpreted functions. In Figure 6, we present our model of a sample bitvector operation xor, sample memory instruction load, and sample SGX instruction eexit. We use the theorem prover’s built-in bitvector theories ($\oplus$ operator in line 1) for modeling microarchitectural instructions that perform bitvector operations. For load, we model both traditional checks (e.g. permission bits, valid page table mapping, etc.) and SGX-specific security checks. First, load reads the page table to translate the virtual address va to physical address pa (line 7) using a traditional page walk, which we model as an array lookup. Operations on arrays consist of reads $x := X[y]$ and writes $X[y] := x$, which are interpreted by the Theory of Arrays [5]. The boolean $ea$ denotes whether this access is made by enclave code to memenc. If $ea$ is true, then load asserts (line 14) that the following security checks succeed:
the translated physical address pa resides in mem_apc (line 9)
epcm contains a valid entry for address pa (lines 10 and 11)
eclave’s epcm entry and the CPU’s control register both agree that the enclave owns the page (line 12)
the page’s mapping in pagetable is same as when enclave was initialized (line 13)

If non-enclave code is accessing mem_apc, or if penc is accessing some other enclave’s memory (i.e. within mem_apc but outside mem_enc), then load returns a dummy value 0xff (line 16). We refer the reader to [20] for details on SGX memory access semantics. Figure 6 also contains a model of eexit, which causes the control flow to transfer to the host application. Models of other SGX instructions are available at [1].

1 function xor(x: bv32, y: bv32) { return x ⊕ y; }
2 function load(mem{bv32}bv, va: bv32) { return 0; }
3 | 4 var check : bool; //EPCM security checks succeed?
5 var pa : bv32; //translated physical address
6 var ea : bool; //enclave access to enclave memory?
7 pa := pagetable(va);
8 ea := CR_ENCLAVE_MODE && enc(va);
9 check := epcm(pa) &&
10 EPCM_VALID(epcm(pa)) &&
11 EPCM_PT(epcm(pa)) == PT_REG &&
12 EPCM_ENCLAVESECS(epcm(pa)) == CR_ACTIVE_SECS &&
13 EPCM_ENCLAVEADDRESS(epcm(pa)) == va; //read bit set and pagetable has valid mapping
14 assert (ea => check); //EPCM security checks
15 assert ...; //read bit set and pagetable has valid mapping
16 if (lea && epc(pa)) { return 0xff; } else { return mem[pa]; }
17 |
18 function eexit(mem{bv32}bv, ra: bv32) {
19 |
20 var mem’ := mem; var regs’ := regs;
21 reg’s(rip) := rbx; reg’s[CR_ENCLAVE_MODE] := false;
22 mem’[CR_TCS_PA] := 0x000;
23 return (mem’,regs’); |
24 |
Figure 6: Axioms for xor, load, and eexit instructions

4.3 Adversary Model

In this section, we formalize a passive and active adversary, which is general enough to model an adversarial host application and also privileged malware running in OS/VMM layer. penc’s execution is interleaved with the host application — host application transfers control to penc via eenter or eresume, and penc returns control back to the host application via eexit. Control may also transfer from penc to the OS (i.e. privileged malware) in the event of an interrupt, exception, or fault. For example, the adversary may generate interrupts or control the page tables so that any enclave memory access results in a page fault, which is handled by the OS/VMM. The adversary may also force a hardware interrupt at any time. Once control transfers to adversary, it may execute any number of arbitrary x86+SGX instructions before transferring control back to the enclave. Therefore, our model of an active adversary performs an unbounded number of following adversarial transitions between any consecutive microarchitectural instructions executed by penc:

1. Havoc all non-enclave memory (denoted by havoc mem_enc): While the CPU protects the epc region, a privileged software adversary can write to any memory location in mem_enc region. havoc mem_enc is encoded in BoogiePL as:

   assume Va. epc(a) → mem_enc[a] == mem[a];
   mem := mem_new;

where mem_new is an unconstrained symbolic value that is type-equivalent to mem. Observe that the adversary modifies an unbounded number of memory locations.

2. Havoc page tables: A privileged adversary can modify the page tables to any value. Since page tables reside in mem_apc, havoc mem_apc models havoc on page tables.

3. Havoc CPU registers (denoted by havoc regs). regs are modified only during adversary execution, and retrieve their original values once the enclave resumes. havoc regs is encoded in BoogiePL as:

   regs := regs_new;

where each register (e.g. eax) is set to an unconstrained symbolic value.

4. Generate interrupt (denoted by interrupt): The adversary can generate interrupts at any point, causing the CPU jump to the adversarial interrupt handler.

5. Invoke any SGX instruction with any operands (denoted by call sgx): The attacker may invoke ecreate, eadd, eextend, einit, eenter to launch any number of new enclaves with code and data of attacker’s choosing.

Any x86+SGX instruction that an active adversary invokes can be approximated by some finite-sized combination of the above 5 transitions. Our adversary model is sound because it allows the active adversary to invoke an unbounded number of these transitions. We define an active and passive adversary:

DEFINITION 1. General Active Adversary $\mathcal{G}$. Between any consecutive statements along an execution of penc, $\mathcal{G}$ may execute an unbounded number of transitions of type havoc mem_enc, havoc regs, interrupt, or call sgx, thereby modifying a component $\sigma_\mathcal{G}$ of state $\sigma$. Following each penc, microarchitectural instruction, $\mathcal{G}$ observes a projection $\sigma|_\text{obs}$ of machine state $\sigma$. Here, $\sigma|_\text{obs} \triangleq (\sigma._{\text{mem-enc}}, \sigma._{\text{regs}}, \sigma._{epcm-enc})$.

DEFINITION 2. Passive Adversary $\mathcal{P}$. The passive adversary $\mathcal{P}$ observes a projection $\sigma|_\text{obs}$ of machine state $\sigma$ after each microarchitectural instruction in penc. Here, $\sigma|_\text{obs} \triangleq (\sigma._{\text{mem-enc}}, \sigma._{\text{regs}}, \sigma._{epcm-enc})$ includes the non-enclave memory. $\mathcal{P}$ does not modify any state.

Enclave execution may result in exceptions (such as divide by 0 and page fault) or faults (such as general protection fault), in which case the exception codes are conveyed to the adversarial OS. We omit exception codes from $\sigma|_\text{obs}$ for both $\mathcal{P}$ and $\mathcal{G}$. This is not a major concern as we implement an analysis within Moat to prove the absence of exception-causing errors in penc such as divide by 0 errors. Although $\mathcal{G}$ can cause page fault exceptions, they only reveal memory access patterns (at the page granularity), which we consider to be a side-channel observation. Side-channels are out of our scope, hence we ignore page fault exceptions from $\sigma|_\text{obs}$.

5. COMPOSING ENCLAVE WITH THE ADVERSARY

Moat reasons about penc’s execution in the presence of an adversary ($P$ or $G$) by composing their state transition systems. An execution of penc is a sequence of statements $[l_1 : s_1, l_2 : s_2, \ldots, l_n : s_n]$, where each $s_i$ is a load, store, register
assignment $x := e$, conditional jmp, unconditional jmp, or
a usermode SGX instruction (ereport, egetkey, or eexit).
Since $p_{enc}$ is loop-free, each statement $s_i$ has a distinct la-
bel $l_i$ that corresponds to the program counter. We assume
that each microarchitectural instruction executes atomically,
although the atomicity assumption is architecture dependent.

**Composing enclave $p_{enc}$ with passive adversary $P$.**
In the presence of $P$, $p_{enc}$ undergoes a deterministic se-
quence of state transitions starting from initial state $\sigma_0$. $P$
cannot update Vars, therefore $P$ affects $p_{enc}$’s execution only
via the initial state $\sigma_0$. We denote this sequence of states as
trace $t = [\sigma_0, \sigma_1, \ldots, \sigma_n]$, where $(\sigma_i, \sigma_{i+1}) \in R(s_i)$ for each
$i \in 0, \ldots, n - 1$. We also write this as $(p_{enc}, \sigma_0) \downarrow t$.

**Composing enclave $p_{enc}$ with active adversary $G$.**
$G$ can affect $p_{enc}$ at any step of execution by executing an
unbounded number of adversarial transitions. Therefore, to
model $p_{enc}$’s behaviour in the presence of $G$, we consider
the following composition of $p_{enc}$ and $G$. For each $p_{enc}$ statement
$l : s$, we transform it to:

$$adv_1; \ldots ; adv_k; l : s$$

(2)

This instrumentation guarantees that between any consecu-
tive statements along an execution of $p_{enc}$, $G$ can execute
an unbounded sequence of adversarial transitions $adv_1; \ldots ; adv_k$
where each statement $adv_i$ is an adversarial transition of type
havoc mem-apis, havoc regs, interrupt, or call sgx. This com-
posed model, hereby called $p_{enc} \cdot G$, encodes all possible be-
aviours of $p_{enc}$ in the presence of $G$. An execution of $p_{enc} \cdot G$
is described by a sequence of states i.e. trace $t = [\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n, \sigma_n]$, where each $\sigma_i \in T$ denotes the state after
the last adversary transition $adv_k$ (right before execution res-
umes in the enclave). We coalesce the effect of all adversary
transitions into a single state $\alpha_i$ for cleaner notation. Follow-
ing $adv_k$, the composed model $p_{enc} \cdot G$ executes an enclave
statement $l : s$, taking the system from a state $\alpha_i$ to state $\sigma_i$.

Given a trace $t = [\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n, \sigma_n]$, we define

$$t_{\text{obs}} = [\sigma_0|\text{obs}, \sigma_1|\text{obs}, \ldots, \sigma_n|\text{obs}]$$

denoting the adversary-observable projection of trace $t$, ig-
noring the adversary controlled $\alpha$ states. Correspondingly, we
define

$$t_{\sigma} = [\alpha_0|\sigma, \alpha_1|\sigma, \ldots, \alpha_n|\sigma]$$
capturing the adversary’s effects within a trace $t$. We define the enclave projection of $\sigma$ to be

$$\sigma|\text{enc} = (\sigma_{\text{mem}, \sigma}, \sigma_{\text{regs}, \sigma}, \sigma_{\text{epc}, \sigma})$$

This is the component of machine state $\sigma$ that is accessible
only by $p_{enc}$. Correspondingly, we define

$$t_{\text{enc}} = [\sigma_0|\text{enc}, \sigma_1|\text{enc}, \ldots, \sigma_n|\text{enc}]$$

The transformation in (2) allows the adversary to perform
an unbounded number of operations $adv_1, \ldots, adv_k$, where $k$
is any natural number. Since we cannot verify unbounded
length programs using verification-condition generation, we
consider the following alternatives:

- Bound the number of operations (k) that the adversary is
  allowed to perform. Although this approach bounds the
  length of $p_{enc} \cdot G$, it unsoundly limits the $G$’s capabilities.
- Use alternative adversary models in lieu of $G$ with the hope of
  making the composed model both bounded and sound.

We explore the latter option in Moat. Our initial idea was
to try substituting $P$ for $G$. This would be the equivalent of
making $k$ equal 0, and thus $p_{enc} \cdot G$ bounded in length. How-
ever, for this to be sound, we must prove that $G$’s operations
can be removed without affecting $p_{enc}$’s execution, as required
by the following property.

$$\forall \sigma \in \Sigma. \forall i, j \in \Sigma^*. (p_{enc} \cdot G, \sigma) \downarrow t_i \land (p_{enc}, \sigma) \downarrow t_j \Rightarrow \forall i, t_i|\text{enc}[i] = t_j|\text{enc}[i]$$

(3)

If property (3) holds, then we can substitute $P$ for $G$ while
proving any safety (or k-safety) property of $p_{enc}$. While
attempting to prove this property in the Boogie verifier [3], we
quite unexpectedly discovered counter-examples that illustrate
the different ways in which $G$ affects $p_{enc}$’s execution:

1. Enclave instruction load(mem, a), where $a$ is an address
   in mem-apis. $G$ havoc mem-apis and $p_{enc}$ reads mem-apis for
   inputs, so this counter-example is not surprising.

2. load(mem, a), where $a$ is an address within SSA pages. $G$
can force an interrupt, causing the CPU to save enclave
   state in SSA pages. If the enclave resumes and reads from
   SSA pages, then the value read depends on the enclave
   state at the time of last interrupt.

If we prevent $p_{enc}$ from reading mem-apis or the SSA pages,
we successfully prove property (3). From hereon, we constrain
$p_{enc}$ to not read from SSA pages; we do not find this to be a
restriction in our case studies. However, the former constraint
(not reading mem-apis) is too restrictive in practice because
$p_{enc}$ must read mem-apis to receive inputs. Therefore, we must
explore alternative adversary models. Instead of replacing $G
with $P$, we attempt replacing $G$ with $H$ defined below.

**Definition 3. Havocing Active Adversary $H$.**
Between any consecutive statements along an execution of
$p_{enc}$, $H$ may execute a single havoc mem-apis operation, thereby
modifying a component $\sigma|H$ of machine state $\sigma$. Following
each $p_{enc}$ microarchitectural instruction, $H$ observes a projec-
tion $\sigma|\text{obs}$ of machine state $\sigma$. Here, $\sigma|\text{obs} = (\sigma_{\text{mem}, \text{apis}})$, and
$\sigma|H = (\sigma_{\text{mem}, \text{apis}})$.

**Composing enclave $p_{enc}$ with active adversary $H$.**
To construct $p_{enc} \cdot H$, we transform each $p_{enc}$ statement $l : s$
to:

$$\text{havoc mem-apis;} l : s$$

(4)

Figure 3 shows a sample transformation from $p_{enc}$ to $p_{enc} \cdot H$.
Similar to our prior attempt with $P$, we prove that it is sound
to replace $G$ with $H$ while reasoning about enclave execution.

**Theorem 1.** Given an enclave program $p_{enc}$, let $p_{enc} \cdot G$
be the composition of $p_{enc}$ and $G$ via the transformation in (2)
and $p_{enc} \cdot H$ be the composition of $p_{enc}$ and $H$ via the trans-
formation in (4). Then,

$$\forall \sigma \in \Sigma. \forall i \in \Sigma^*. (p_{enc} \cdot G, \sigma) \downarrow t_i \Rightarrow \exists t_k \in \Sigma^*. (p_{enc} \cdot H, \sigma) \downarrow t_k \land \forall i, t_k|\text{enc}[i] = t_k|\text{enc}[i]$$

Validity of this theorem implies that we can replace $G$
with $H$ while proving any safety property or k-safety hyperproperty
of enclave behaviour [12]. We prove theorem 1 with the use of
lemma 1 and lemma 2 stated below.
The transformation in (2) composed $p_{enc}$ with $G$ by instrumenting an unbounded number of adversary operations $adv_1; \ldots ; adv_k$ before each statement in $p_{enc}$. Now, let us further instrument $\text{havoc mem-asp}$ after each $adv_i \in \{adv_1; \ldots ; adv_k\}$ — this is sound because a $\text{havoc}$ on $\text{mem-asp}$ does not restrict the allowed values of $\text{mem-asp}$. The resulting instrumentation for each statement $l$: $s$ is:

$$adv_1; \text{havoc mem-asp}; \ldots ; adv_k; \text{havoc mem-asp}; l : s \quad (5)$$

Lemma 1 proves that the effect of $adv_i \in \{adv_1; \ldots ; adv_k\}$ on $p_{enc}$ can be simulated by a sequence of havocs to $\text{mem-asp}$. In order to define lemma 1, we introduce the following transformation on each statement $l$: $s$ of $p_{enc}$:

$$\text{havoc mem-asp}; \ldots ; \text{havoc mem-asp}; l : s \quad (6)$$

**Lemma 1.** Given an enclave program $p_{enc}$, let $p_{enc-H}$ be the composition of $p_{enc}$ and adversary via the transformation in (5) and $p_{enc-H^*}$ be the composition of $p_{enc}$ and adversary via the transformation in (6). Then,

$$\forall \sigma \in \Sigma. \forall t_1 \in \Sigma^*. (p_{enc-H^*}, \sigma) \Downarrow t_1 \Rightarrow \exists t_2 \in \Sigma^*. (p_{enc-H}, \sigma) \Downarrow t_2 \land \forall i. t_1|\text{enc}[i] = t_2|\text{enc}[i]$$

**Proof:** This intuition behind this proof is that the other adversarial transitions do not affect $p_{enc}$ in any way that is not simulated by $\text{havoc mem-asp}$. We prove this lemma by induction in the Boogie verifier [3]. This property is a predicate over a pair of traces, making it a 2-safety hyperproperty [12]. A counter-example to this property is a pair of traces $t_i, t_j$ where $\mathcal{G}$ has caused $t_i$ to diverge from $t_j$. We rewrite this as a 2-safety property and prove it via 1-step induction over the length of the trace, as follows. For any pair of states $(\sigma_i, \sigma_j)$ that is indistinguishable to the enclave, we prove that after one transition, the new pair of states $(\sigma_i', \sigma_j')$ is also indistinguishable. Here, $(\sigma_i, \sigma_i') \in \mathcal{R}(s_i)$ and $(\sigma_j, \sigma_j') \in \mathcal{R}(s_j)$, where $s_i$ is executed by $p_{enc-H^*}$ and $s_j$ is executed by $p_{enc-H^*}$. The state predicate $\text{Init}$ represents an enclave state after invoking $\text{einit}$ in the prescribed initialization sequence in (1).

$$\forall \sigma_i, \sigma_j. \text{Init}(\sigma_i) \land \text{Init}(\sigma_j) \Rightarrow \sigma_i|\text{enc} = \sigma_j|\text{enc} \quad (7)$$

$$\forall \sigma_i, \sigma_j, \sigma_i', \sigma_j'. \sigma_i|\text{enc} = \sigma_j|\text{enc} \land (\sigma_i, \sigma_i') \in \mathcal{R}(s_i) \land (\sigma_j, \sigma_j') \in \mathcal{R}(s_j) \land p(s_i, s_j) \Rightarrow \sigma_i'|\text{enc} = \sigma_j'|\text{enc} \quad (8)$$

where

$$p(s_i, s_j) = \begin{cases} s_i \in \{\text{getkey, ereport, eexit, load, store}\} \land s_j = s_i & \text{if } s_i = \text{havoc mem-asp} \land s_j = \text{havoc mem-asp} \\ s_i \text{ is an interrupt or call agx} & \text{if } s_i \in \{\text{havoc mem-asp, \ldots, interrupt, call agx}\} \end{cases}$$

**Lemma 2.** A sequential composition of unbounded number of $\text{havoc mem-asp}$ statements can be simulated by a single $\text{havoc mem-asp}$ statement.

Combining lemma 1 and lemma 2, we prove that the effect of $adv_1; \text{havoc mem-asp}; \ldots ; adv_k; \text{havoc mem-asp}$ (or $adv_1; adv_2; \ldots ; adv_k$) on enclave’s execution can be simulated by $\text{havoc mem-asp}$. By theorem 1, it is sound to prove any safety (or k-safety) property on $p_{enc-H}$ because $p_{enc-H}$ allows all traces allowed by $p_{enc-G}$. The benefits of composing with $H$ are (1) $p_{enc-H}$ is bounded in size, which allows using any off-the-shelf sequential program verifier to prove safety (or k-safety) properties of enclave executions, and (2) $H$ gives a convenient model of adversary’s effects on enclave execution. In this paper, we focus on proving confidentiality.

6. FORMALIZING CONFIDENTIALITY

Moat’s definition of confidentiality is inspired by standard non-interference definition [21], but adapted to the instruction-level modeling of the enclave programs. Confidentiality can be trivially achieved with the definition that $H$ cannot distinguish between $p_{enc}$ and an enclave that executes skip in each step. However, such definition prevents $p_{enc}$ from writing to $\text{mem-asp}$, which it must in order to return outputs or send messages to remote parties. To that end, we weaken this definition to allow for writes to $\text{mem-asp}$, but constraining the values to be independent of the secrets. An input to Moat is a policy that defines $\text{Secrets} = \{(l, v) \mid l \in L, v \in \mathcal{V}_{\text{ars}}\}$. Where a tuple $(l, v)$ denotes that variable $v$ holds a secret value at program location $l$. In practice, since secrets typically occupy several bytes in memory, $v$ is a range of addresses in the enclave heap. We define the following transformation from $p_{enc-H}$ to $p_{enc-H^*}$ for formalizing confidentiality. For each $(l, v) \in \text{Secrets}$, we transform the statement $l : s$ to:

$$l : s; \text{havoc } v; \quad (9)$$

$havoc$ $v$ assigns an unconstrained symbolic value to variable $v$. With this transformation, we define confidentiality as follows:

**Definition 4.** **Confidentiality** For any pair of traces of $p_{enc-H^*}$ that potentially differ in the values of the Secret variables, if $H$’s operations along the two traces are equivalent, then $H$’s observations along the two traces must also be equivalent.

$$\forall \sigma \in \Sigma. \forall t_1, t_2 \in \Sigma^*.(p_{enc-H^*}, \sigma) \Downarrow t_1 \land (p_{enc-H^*}, \sigma) \Downarrow t_2 \land \forall i. t_1|\text{enc}[i] = t_2|\text{enc}[i] \Rightarrow (\forall i. t_1|\text{obs}[i] = t_2|\text{obs}[i]) \quad (10)$$

The $\text{havoc}$ on $\text{Secrets}$ cause the secret variables to take potentially differing symbolic values in $t_1$ and $t_2$. However, property (10) requires $t_1|\text{obs}$ and $t_2|\text{obs}$ to be equivalent, which is achieved only if secrets do not leak to $H$-observable state.

While closer to the desired definition, it still prevents $p_{enc}$ from communicating declassified outputs that depend on secrets. For instance, recall that the OTP enclave outputs the encrypted secret to be stored to disk. In this case, since different values of secret produce different values of ciphertext, $p_{enc}$ violates property (10). The policy defines $\text{Declassified} = \{(l,v) \mid l \in L, v \in \mathcal{V}_{\text{ars}}\}$, where a tuple $(l, v)$ denotes that variable $v$ at location $l$ contains a declassified value. We can safely eliminate declassified outputs from information leakage checking as the protocol verifier has already proven them to be safe outputs. In practice, since outputs typically occupy several bytes in memory, $v$ is a range of addresses in the enclave heap. When declassification is necessary, we use the following property for checking confidentiality.

**Definition 5.** **Confidentiality with Declassification** For any pair of traces of $p_{enc-H^*}$ that potentially differ in the values of the Secret variables, if $H$’s operations along the two traces are equivalent, then $H$’s observations (ignoring Declassified outputs) along the two traces must also be equivalent.

$$\forall \sigma \in \Sigma. \forall t_1, t_2 \in \Sigma^*.(p_{enc-H^*}, \sigma) \Downarrow t_1 \land (p_{enc-H^*}, \sigma) \Downarrow t_2 \land \forall i. t_1|\text{enc}[i] = t_2|\text{enc}[i] \Rightarrow (\forall i. t_1|\text{obs}[i] = t_2|\text{obs}[i]) \land (\forall i. (i, \text{mem}[j]) \in \text{Declassified} \rightarrow t_1|\text{obs}[i]|\text{mem}[j] = t_2|\text{obs}[i]|\text{mem}[j]) \quad (11)$$

7. PROVING CONFIDENTIALITY
Our goal is to automatically check if $p_{\text{enc-H}}$ satisfies confidentiality (property 11), which would ensure safety against application attacks. Since confidentiality is a 2-safety hyper-property (property over pairs of traces), we cannot use black box program verification techniques, which are tailored towards safety properties. We cannot use dynamic techniques such as reference monitors for that reason. To that end, we create a security type system in which type safety implies that $p_{\text{enc-H}}$ satisfies confidentiality. We avoid a self-composition approach because of complications in encoding equivalence assumptions over adversary operations in the two traces of $p_{\text{enc-H-sec}}$ (property 11). As is standard in many type-based approaches [26, 22] for checking confidentiality, the typing rules prevent programs with explicit and implicit information leaks. Explicit leaks occur via assignments of secret values to $H$-observable state i.e. $\sigma|_{\text{obs}}$. For instance, the program \texttt{mem := store(mem,y,x)} is ill-typed if $x$'s value depends on a secret and $\text{enc}(y)$ is false i.e. it writes a secret to non-enclave memory. An implicit leak occurs when a conditional statement has a secret-dependent guard, but updates $H$-visible state in either branch. For instance, the program if ($x == 42$) \{ \text{mem := store(mem,y,i)} \} else \{ \text{skip} \} is ill-typed if $x$'s value depends on a secret and $\text{enc}(y)$ is false. In both examples above, $H$ learns the secret value $x$ by reading \texttt{mem} at address $y$. In addition to the \texttt{store} instruction, explicit and implicit leaks may also be caused by unsafe use of SGX instructions. For instance, \texttt{egetkey} returns a secret sealing key, which must not be leaked from the enclave. Similarly, \texttt{ereport} generates a signed report containing public values (e.g. measurement) and potentially secret values (enclave code may include 64 bytes of data, which may be secret). Our type system models these details of SGX accurately, and accepts $p_{\text{enc-H}}$ only if it has no implicit or explicit leaks.

A security type is either $\top$ (secret) or $\bot$ (public). At each program label, each memory location and CPU register has a security type based on the x86+SGX instructions executed until that label. The security types are needed at each label because variables (especially \texttt{regs}) may alternate between holding secret and public values. As explained later in this section, \texttt{Moat} needs the security types in order to decide whether a \texttt{store} instruction causes implicit or explicit leaks. $p_{\text{enc-H}}$ accompanies a policy containing \texttt{Secrets} = \{(l,v)\} and \texttt{Declassified} = \{(l,v)\}, where a tuple $\langle l,v \rangle$ denotes that variable $v$ at program location $l$ contains a secret and declassified value, respectively. However, there are no other type declarations; therefore, \texttt{Moat} implements a type inference algorithm based on computing refinement type constraints and checking their validity using a theorem prover. In contrast, type checking without inference would require the programmer to painstakingly provide security types for each memory location and CPU register.

\texttt{Moat}'s type inference algorithm computes logical constraints under which an expression or statement takes a security type. A typing judgment $\tau \vdash e : \top \Rightarrow \psi$ means that the expression $e$ has security type $\tau$ whenever the constraint $\psi$ is satisfied. An expression of the form $\text{op}(v_1, \ldots, v_n)$ (where $\text{op}$ is a relation or function) has type $\tau$ if all variables $\{v_1, \ldots, v_n\}$ have type $\tau$ or lower. For instance, an expression may have type $\bot$ iff its value is independent of \texttt{Secrets}.

For a statement $s$ to have type $\tau$, every assignment in $s$ must update a state variable whose security class is $\tau$ or higher. We write this typing judgment as $[\tau] \vdash s : \psi, \mathcal{F}$, where $\psi$ is a SMT formula and $\mathcal{F}$ is a set of SMT formulæ. Each satisfiable interpretation of $\psi$ corresponds to a feasible execution of $s$. $\mathcal{F}$ contains a SMT formula for each \texttt{store} instruction in $s$, such that the formula is valid iff the \texttt{store} does not leak secrets. We present our typing rules in Figure 7, which assume that $p_{\text{enc-H}}$ is first converted to static assignment form. $s$ has type $\tau$ if we derive $[\tau] \vdash s : \langle \psi, \mathcal{F} \rangle$ using the typing rules, and prove that all formulæ in $\mathcal{F}$ are valid. If $s$ has type $\top$, then $s$ does not update $H$-visible state, and thus cannot contain information leaks. Having type $\top$ also allows $s$ to execute in a context where a secret value is implicitly known through the guard of a conditional statement. On the other hand, type $\bot$ implies that $s$ either does not update $H$-observable state or the update is independent of \texttt{Secrets}.

By Theorem 1, $p_{\text{enc-H}}$ models all potential runtime behaviours of $p_{\text{enc}}$ in the presence of an active adversary ($\Psi$ or $H$). For that reason, \texttt{Moat} feeds $p_{\text{enc-H}}$ to the type checking algorithm. We now explain some of our typing rules from Figure 7. For each variable $v \in \texttt{Vars}$ within $p_{\text{enc-H}}$, our typing rules introduce a ghost variable $C_v$ that is true iff $v$ has security type $\top$. For a register variable $v$ (e.g. $C_{\text{reg}}$), $C_v$ is a boolean; for an array variable $v$, $C_v$ (e.g. $C_{\text{mem}}$) is an array and $C_v[i]$ denotes the security type for each index $i$. \texttt{exp1} rule allows inferring the type of any expression $e$ as $\top$. \texttt{exp2} rule allows inferring an expression type $e$ as $\bot$ if we derive $C_v$ to be false for all variables $v$ in the expression $e$. \texttt{store} rule marks the stored value as secret if either the input value or address is secret. \texttt{ereportL} rule classifies the memory locations updated by \texttt{ereport} according to the semantics of Intel SGX. \texttt{ereport} takes 64 bytes of data at address in $\texttt{rcx}$, and copies them to memory starting at $\texttt{rdx} + 320$, and the rest of the report consists of public data such as the MAC, measurement, etc. Hence, $C_{\text{mem}}$ retains the secrecy level for the 64 bytes of data, and assumes the secrecy level of $C_{\text{mem}}$ (rdx determines report’s location) for the public data. \texttt{egetkey} stores 16 bytes of the sealing key at address $\texttt{rcx}$, hence the \texttt{egetkey} rule marks those 16 bytes in $C_{\text{mem}}$ as secret. \texttt{eseit} jumps back to $H$ code without clearing any of the general purpose \texttt{regs}. Hence, the \texttt{eseit} rule asserts that those \texttt{regs} hold public values. We prove the following type soundness theorem in a companion report [1].

**Theorem 2.** For any $p_{\text{enc-H}}$ such that $[\tau] \vdash p_{\text{enc-H}} \Rightarrow (\psi, \mathcal{F})$ is derivable (where $\tau$ is either $\top$ or $\bot$) and all formulæ in $\mathcal{F}$ are valid, $p_{\text{enc-H}}$ satisfies property 11.

\texttt{Moat} implements this type system by replacing each statement $s$ in $p_{\text{enc-H}}$ by $I(s)$ using the instrumentation rules in Figure 8. Observe that we introduce $C_{\text{pc}}$ to track whether confidential information is implicitly known through the program counter. If a conditional statement’s guard depends on a secret value, then we set $C_{\text{pc}}$ to $true$ within the then and else branches. \texttt{Moat} invokes $I(p_{\text{enc-H}})$ and applies the instrumentation rules in Figure 8 recursively. Figure 4 demonstrates an example of instrumenting $p_{\text{enc-H}}$. \texttt{Moat} then feeds the instrumented program $I(p_{\text{enc-H}})$ to an off-the-shelf program verifier, which proves validity all assertions or finds a counterexample. Our implementation uses the Boogie [3] program verifier, which receives $I(p_{\text{enc-H}})$ and generates verification conditions (using Weakest Precondition calculus [4]) in the SMT format. Boogie uses the Z3 [13] theorem prover (SMT solver) to prove the verification conditions. An advantage of using SMT solving is that a typing error is explained using counter-example execution, demonstrating the information leak and exploit. We find this helpful during debugging.

In summary, \texttt{Moat}'s type system is inspired by the type-based approach for information flow checking by Volpano et
\[ e : τ \Rightarrow \text{true} \]
\[ e : \bot \Rightarrow \bigwedge_{\text{vars}(e)} \text{mem} \]
\[ \forall \left( \begin{array}{c}
\bigwedge_{\text{vars}(e)} \text{mem} \\
\neg \psi, A
\end{array} \right) \] (coercion)
\[ \left(\right. \begin{array}{c}
\text{mem} \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \text{true}, \{\emptyset\} \rangle \\
\end{array} \right) \] (skip)
\[ \left(\begin{array}{c}
\text{assert} \\
\phi
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \phi, \{\emptyset\} \rangle \\
\end{array} \right) \] (assert)
\[ \left(\begin{array}{c}
\langle \text{true}, \{\emptyset\} \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \phi, \{\emptyset\} \rangle \\
\end{array} \right) \] (assert)
\[ \left(\begin{array}{c}
\langle \phi, \{\emptyset\} \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \phi, \{\emptyset\} \rangle \\
\end{array} \right) \] (scalar)
\[ \left(\begin{array}{c}
\langle \text{load}(\text{mem}, e) \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \text{mem} \rangle \\
\end{array} \right) \] (load)
\[ \left(\begin{array}{c}
\langle \text{store}(\text{mem}, y) \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \text{store}(\text{mem}, y) \rangle \\
\end{array} \right) \] (store)
\[ \left(\begin{array}{c}
\langle \text{store}(\text{mem}, y) \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \text{store}(\text{mem}, y) \rangle \\
\end{array} \right) \] (storeH)
\[ \left(\begin{array}{c}
\langle \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx}, \text{rdx}) \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx}, \text{rdx}) \rangle \\
\end{array} \right) \] (reporL)
\[ \langle \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx}) \rangle \Rightarrow \left(\begin{array}{c}
\langle \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx}) \rangle \\
\end{array} \right) \] (egetkey)
\[ \langle \text{exit}(\text{mem}) \rangle \Rightarrow \left(\begin{array}{c}
\langle \text{exit}(\text{mem}) \rangle \\
\end{array} \right) \] (exit)
\[ \left(\begin{array}{c}
\langle \text{load}(\text{mem}, e) \rangle \\
\end{array} \right) \Rightarrow \left(\begin{array}{c}
\langle \text{load}(\text{mem}, e) \rangle \\
\end{array} \right) \] (ite)

Figure 7: Typing Rules for \( p_{\text{enc-\text{H}}} \)

Figure 8: Instrumentation rules for \( p_{\text{enc-\text{H}}} \)

Our type system includes rules for SGX instructions \( \text{ereport}, \text{egetkey}, \text{and exit} \). The rules precisely model the memory locations written by these instructions, and whether the produced data is public or confidential.

Our type system is also sensitive, path-sensitive, and avoids alias analysis because we instrument typing assertions within the program. A program is well-typed if the typing assertions are valid in all feasible executions. We ensure soundness by using a sound protocol verifier for checking all executions of the instrumented \( p_{\text{enc-\text{H}}} \).

We now describe some case studies which we verified using \( \text{Moat} \) and \( \text{ProVerif} \). We upload these case studies at [1], and summarize the results in Figure 9.

We use the following standard cryptographic notation and assumptions. \( m_1, \ldots, m_n \) denotes tagged concatenation of \( n \)
messages. We use a keyed-hash message authentication function $MAC_e(text)$ and hash function $H(text)$, both of which are assumed to be collision-resistant. For asymmetric cryptography, $K_{e}^{-1}$ and $K_e$ are principal $e$'s private and public signature keys, where we assume that $K_e$ is long-lived and distributed within certificates signed by a root of trust authority. Digital signature using a key $k$ is written as $\text{Sig}_k(text)$; we assume unforgeability under chosen message attacks. Intel provisions each SGX processor with a unique private key $K_{SGX}$ that is available to a special quoting enclave. In combination with this quoting enclave, an enclave can invoke ereport to produce quotes, which is essentially a signature (using the SGX private key) of the data produced by the enclave and its measurement. We write a quote produced on behalf of enclave $e$ as $\text{Quote}_e(text)$, which is equivalent to $\text{Sig}_{K_{SGX}}(H(text) \mid Me)$ — measurement of enclave $e$ is written as $Me$. $N$ is used to denote nonce. Finally, we write $\text{Enc}_e(text)$ for the encryption of $text$, for which we assume indistinguishability under chosen plaintext attack. We also use $\text{AEnc}_e(text)$ for authenticated encryption, for which we assume indistinguishability under chosen plaintext attack and integrity of ciphertext.

One-time Password Generator.

The abstract model of the OTP secret provisioning protocol (from §2), where client runs in a SGX enclave, bank is an uncompromised service, and disk is under adversary control:

$$\begin{align*}
\text{bank} \rightarrow \text{client} & : N \\
\text{client} \rightarrow \text{bank} & : N \mid g^a \mid \text{Quote}_{\text{client}}(N \mid g^a) \\
\text{bank} \rightarrow \text{client} & : N \mid g^b \mid \text{Sig}_{K_{\text{bank}}}(N \mid g^b) \mid \text{AEnc}_{H(g^b)}(\text{secret}) \\
\text{client} \rightarrow \text{disk} & : \text{AEnc}_{K_{\text{seal}}}(\text{secret})
\end{align*}$$

First, we use Moat to prove that $g^b$ and $K_{\text{seal}}$ are not leaked to $H$. Next, ProVerif uses secrecy assumption on $g^b$ and $K_{\text{seal}}$ to prove that secret is not leaked to the network (or disk) adversary. This proof allows Moat to declassify client's output to disk while proving property 11. Moat successfully proves that the client enclave satisfies confidentiality.

Query Processing over Encrypted Database.

In this case study, we evaluate Moat on a stand-alone application, removing the possibility of protocol attacks and therefore the need for any protocol verification. We build a database table containing two columns: name which is deterministically encrypted, and amount which is non-deterministically encrypted. Alice wishes to select all rows with name “Alice” and sum all the amounts. We partition this computation into two parts: unprivileged computation (which selects the rows) and enclave computation (which computes the sum).

Notary Service.

We implement a notary service introduced by [16] but adapted to run on SGX. The notary enclave assigns logical timestamps to documents, giving them a total ordering. The notary enclave responds to (1) a connect message for obtaining the attestation report, and (2) a notarize message for obtaining a signature over the document hash and the current counter.

$$\begin{align*}
\text{user} \rightarrow \text{notary} & : \text{connect} \mid N \\
\text{notary} \rightarrow \text{user} & : \text{Quote}_{\text{notary}}(N) \\
\text{user} \rightarrow \text{notary} & : \text{notarize} \mid H(text) \\
\text{notary} \rightarrow \text{user} & : \text{counter} \mid H(text) \mid \text{Sig}_{K_{\text{notary}}} (\text{counter} \mid H(text))
\end{align*}$$

The only secret here is the private signature key $K_{\text{notary}}^{-1}$. First, we use Moat to prove that $K_{\text{notary}}^{-1}$ is not leaked to $H$. This proof fails because the output of $\text{Sig}$ (in the response to notarize message) depends on the secret signature key — Moat is unaware of cryptographic properties of $\text{Sig}$. ProVerif proves that this message does not leak $K_{\text{notary}}^{-1}$ to a network adversary, which allows Moat to declassify this message and prove that the notary enclave satisfies the confidentiality property.

End-to-End Encrypted Instant Messaging.

We implement the off-the-record messaging protocol [10], which provides perfect forward secrecy and repudiability for messages exchanged between principals $A$ and $B$. We adapt this protocol to run on SGX, thus providing an additional guarantee that an infrastructure attack cannot compromise the Diffie-Hellman private keys. We only present a synchronous form of communication here for simplicity.

$$\begin{align*}
A \rightarrow B & : g^{a_1} \mid \text{Sig}_{K_{A}}^{-1}(g^{a_1}) \mid \text{Quote}_A(\text{Sig}_{K_{A}}^{-1}(g^{a_1})) \\
B \rightarrow A & : g^{b_1} \mid \text{Sig}_{K_{B}}^{-1}(g^{b_1}) \mid \text{Quote}_B(\text{Sig}_{K_{B}}^{-1}(g^{b_1})) \\
A \rightarrow B & : \text{Enc}_{H(g^{a_1}b_1)}(m_1) \mid \text{MAC}_{H(g^{a_1}b_1)}(g^{a_1} \mid \text{Enc}_{H(g^{a_1}b_1)}(m_1)) \\
B \rightarrow A & : \text{Enc}_{H(g^{b_2}a_2)}(m_2) \mid \text{MAC}_{H(g^{b_2}a_2)}(g^{b_2} \mid \text{Enc}_{H(g^{b_2}a_2)}(m_2)) \\
A \rightarrow B & : g^{a_3} \mid \text{Enc}_{H(g^{a_3}b_3)}(m_3) \mid \text{MAC}_{H(g^{a_3}b_3)}(g^{a_3} \mid \text{Enc}_{H(g^{a_3}b_3)}(m_3))
\end{align*}$$

The OTR protocol only needs a digital signature on the initial Diffie-Hellman exchange — future exchanges use MACs to authenticate a new key using an older, known-authentic key. For the same reason, we only append a SGX quote to the initial key exchange. First, we use Moat to prove that the Diffie-Hellman secrets computed by $\text{Quote}$ (i.e. $g^{a_1b_1}$, $g^{a_2b_2}$) are not leaked to $H$. Next, ProVerif uses this secrecy assumption to prove that messages $m_1$, $m_2$, and $m_3$ are not leaked to the network adversary. The ProVerif proofs allows Moat to declassify all messages following the initial key exchange, and successfully prove confidentiality.

![Figure 9: Summary of experimental results. Columns are (1) instructions analyzed by Moat not including crypto library, (2) size of $I(p_{nocc})$, (3) proof time, (4) number of secret and declassified annotations.](image-url)
protection and code from an active adversary. However, VC3's confidentiality guarantee is based on the assumption that enclave code does not leak secrets, and we can use Moat to verify this assumption.

Verifying Information Flow on Programs. Checking implementation code for safety is also a well studied problem. Type systems proposed by Sabelfeld et al. [24], Barthe et al. [6], and Volpano et al. [26] enable the programmer to annotate variables that hold secret values, and ensure that these values do not leak. However, these works assume that the infrastructure (OS/VMM, etc.) on which the code runs is safe, which is unrealistic due to malware and other attacks (e.g. Heartbleed [14]). Our approach builds upon this body of work, showing how it can be adapted to the setting where programs run on an adversarial OS/VMM, and instead rely on trusted SGX hardware for information-flow security.

Cryptographic Protocol Verification. There is a vast literature on cryptographic protocol verification (e.g. [8, 9]). Our work builds on top of cryptographic protocol verifiers showing how to use them to reason about protocol attacks and to generate annotations for more precise verification of enclave programs. In the future, it may also be possible to connect our work to the work on correct-by-construction generation of cryptographic protocol implementation [15].

10. CONCLUSION

This paper introduces a technique for verifying information flow properties of SGX enclave programs. Moat is a first step towards building an end-to-end verifier. Our current evaluation uses separate models for Moat and Proverif. In future work, we plan to design a single high-level language from which we can generate a penvc, a binary, and a protocol model.

11. REFERENCES

We show that in all cases, we derive \[ t_1 ≅ t_2 \].

In this case, our type system sets \( \sigma_1 \), \( \sigma_2 \) and \( \sigma_1', \sigma_2' \). By induction hypothesis, we know \( (\neg\sigma_1, C_{\text{mem}}) \land (\neg\sigma_2, C_{\text{mem}}) \Rightarrow (\sigma_1, C_{\text{mem}}) \). Propagating these equalities, we derive \( (\neg\sigma_1', C_{\text{mem}}) \land (\neg\sigma_2', C_{\text{mem}}) \). Regarding update to \( \text{reg} \), we have two cases:

1. \( t : e = \bot \) in both \( \sigma_1 \) and \( \sigma_2 \), i.e. \( \bigwedge_{e \in \text{Vars}(\sigma)} \neg\sigma_1, C_e \land \bigwedge_{e \in \text{Vars}(\sigma)} \neg\sigma_2, C_e \).
   
   In this case, property 12 dictates that \( e \) evaluates to the same value in states \( \sigma_1 \) and \( \sigma_2 \). Therefore, the updated register \( x' \) has the same value in both states \( \sigma_1' \) and \( \sigma_2' \). We derive \( \forall i, r \in \text{reg} \). \( (\neg t_1[i], C_i \land \neg t_2[i], C_i) \Rightarrow (t_1[i], r = t_2[i], r) \).

2. \( t : e \in \text{reg} \) and \( e \in \text{reg} \). \( (\neg t_1[i], C_i \land \neg t_2[i], C_i) \Rightarrow (t_1[i], r = t_2[i], r) \).

In both cases, we derive \( \forall i, r \in \text{reg} \). \( (\neg t_1[i], C_i \land \neg t_2[i], C_i) \Rightarrow (t_1[i], r = t_2[i], r) \). Conjuncting with our earlier derivation on \( \text{mem} \), we show that \( \text{scalar} \) preserves property 12.

Now we prove that \( \text{scalar} \) also preserves property 13. By induction hypothesis, we know \( \forall j, \neg\text{enc}(j) \Rightarrow (\neg\sigma_1, C_{\text{mem}}) \land (\neg\sigma_2, C_{\text{mem}}) \). Since scalar assignments do not update \( \text{mem} \), \( C_{\text{mem}} \) also retains its value. Therefore, we prove \( \forall j, \neg\text{enc}(j) \Rightarrow (\neg\sigma_1', C_{\text{mem}}) \land (\neg\sigma_2', C_{\text{mem}}) \). Property 13 is preserved by \( \text{scalar} \).

LOAD

A load performs one transition step, hence traces \( t_1 \) and \( t_2 \) contain two states each. \( t_1 = [\sigma_1, \sigma_1'] \), where \( \sigma_1, \sigma_1' \in \mathcal{R}(x' := e) \). Similarly, \( t_2 = [\sigma_2, \sigma_2'] \). Scalar assignments only update \( \text{reg} \), therefore \( \sigma_{1,\text{mem}} = \sigma_{1',\text{mem}} \) and \( \sigma_{2,\text{mem}} = \sigma_{2',\text{mem}} \). For the same reason, \( \sigma_{1,\text{mem}} = \sigma_{1',\text{mem}} \) and \( \sigma_{2,\text{mem}} = \sigma_{2',\text{mem}} \). By induction hypothesis, we know \( \forall j, (\neg\sigma_1, C_{\text{mem}}) \land (\neg\sigma_2, C_{\text{mem}}) \Rightarrow (\sigma_1, C_{\text{mem}}) \land (\sigma_2, C_{\text{mem}}) \). Propagating these equalities, we derive \( (\neg\sigma_1', C_{\text{mem}}) \land (\neg\sigma_2', C_{\text{mem}}) \). Regarding update to \( \text{reg} \), we have two cases:

1. \( t : e = \bot \) in both \( \sigma_1 \) and \( \sigma_2 \), i.e. \( \bigwedge_{e \in \text{reg} \land \text{load}} \neg\sigma_1, C_e \land \bigwedge_{e \in \text{reg} \land \text{load}} \neg\sigma_2, C_e \).
   
   In this case, property 12 dictates that \( e \) evaluates to the same value in states \( \sigma_1 \) and \( \sigma_2 \). We have two nested cases:
   
   a. \( C_{\text{mem}}[i] = \text{false} \) in both \( \sigma_1 \) and \( \sigma_2 \).
      
      Therefore, register \( x' \) is updated to the same value in both states \( \sigma_1 \) and \( \sigma_2 \). We derive \( \forall r \in \text{reg} \). \( (\neg\sigma_1', C_i \land \neg\sigma_2', C_i) \Rightarrow (\sigma_1, C_i) \). We appropriately set \( C_e \) to \( \sigma_1', r = \sigma_2', r \).
   
   b. \( C_{\text{mem}}[i] = \text{true} \) in either \( \sigma_1 \) or \( \sigma_2 \), or both.
      
      We appropriately set \( C_e \) to \( \sigma_1', r = \sigma_2', r \) trivially true.

2. \( t : e \in \text{reg} \) and \( e \in \text{reg} \). \( (\neg t_1[i], C_i \land \neg t_2[i], C_i) \Rightarrow (t_1[i], r = t_2[i], r) \).

In all cases, we derive \( \forall i, r \in \text{reg} \). \( (\neg t_1[i], C_i \land \neg t_2[i], C_i) \Rightarrow (t_1[i], r = t_2[i], r) \). Conjuncting with our earlier derivation on \( \text{mem} \), we show that \( \text{scalar} \) preserves property 12.
Now we prove that load also preserves property 13. By induction hypothesis, we know ∀y. \( \neg \text{enc}(j) \Rightarrow (\neg \sigma_1, \text{mem}[j] \land \neg \sigma_2, \text{mem}[j]) \). Since load does not update mem, \( \text{mem} \) also retains its value. Therefore, we prove ∀y. \( \neg \text{enc}(j) \Rightarrow (\neg \sigma'_1, \text{mem}[j] \land \neg \sigma'_2, \text{mem}[j]) \). Property 13 is preserved by load.

(storeH).

A map assignment performs one transition step, hence traces \( t_1 \) and \( t_2 \) contain two states each. \( t_1 = [\sigma_1, \sigma'_1] \), where \( (\sigma_1, \sigma'_1) \in \mathcal{R} (\text{mem}' \Rightarrow \text{store}(\text{mem}, y, e)) \). Similarly, \( t_2 = [\sigma_2, \sigma'_2] \), where \( (\sigma_2, \sigma'_2) \in \mathcal{R} (\text{mem}' \Rightarrow \text{store}(\text{mem}, y, e)) \). The storeH rule assigns \( \sigma'_1, \text{mem}[\text{var}], \text{mem}[\text{var}] \) to true. This allows us to derive ∀y. \( (\neg \sigma'_1, \text{mem}[j] \land \neg \sigma'_2, \text{mem}[j]) \Rightarrow (\sigma'_1, \text{mem}[j] = \sigma'_2, \text{mem}[j]) \). Furthermore, \( \text{mem}' \Rightarrow \text{store}(\text{mem}, y, e) \) doesn’t affect regs, allowing us to derive property 12.

Next, we prove that storeH also preserves property 13. Observe that storeH generates an assertion \( \text{enc}(y) \). Therefore, \( \text{mem} \) retains its value for any location \( y \) for which \( \neg \text{enc}(y) \) holds, allowing us to derive property 13. In other words, if this enc(y) is valid, then property 13 holds trivially in all executions.

(storeL).

A map assignment performs one transition step, hence traces \( t_1 \) and \( t_2 \) contain two states each. \( t_1 = [\sigma_1, \sigma'_1] \), where \( (\sigma_1, \sigma'_1) \in \mathcal{R} (\text{mem}' \Rightarrow \text{store}(\text{mem}, y, e)) \). Similarly, \( t_2 = [\sigma_2, \sigma'_2] \), where \( (\sigma_2, \sigma'_2) \in \mathcal{R} (\text{mem}' \Rightarrow \text{store}(\text{mem}, y, e)) \). Furthermore, note that storeL generates \( \neg \text{enc}(y) \Rightarrow (\psi_1 \land \psi_2) \).

First, we prove that storeL preserves property 12. Since registers are not updated, we derive ∀i, r \( \in \text{regs.} \) \( (\neg t_1[i], \text{C}_\text{r}, \land \neg t_2[i], \text{C}_\text{r}) \Rightarrow (t_1[i], r = t_2[i], r) \). By induction hypothesis, we know ∀y. \( (\neg \sigma_1, \text{mem}[j] \land \neg \sigma_2, \text{mem}[j]) \Rightarrow (\sigma_1, \text{mem}[j] = \sigma_2, \text{mem}[j]) \). We have three cases:

1. \( \text{enc}(y) \)
   - (a) \( t \models y : \bot \) in both \( \sigma_1 \) and \( \sigma_2 \), and \( t \models e : \bot \) in both \( \sigma_1 \) and \( \sigma_2 \)
   - i.e. \( \bigwedge_{v \in \text{Vars}(\text{enc}) \cup \text{Vars}(y)} \neg \sigma_1, \text{C}_\text{r} \land \bigwedge_{v \in \text{Vars}(\text{enc}) \cup \text{Vars}(y)} \sigma_2, \text{C}_\text{r} \).
   - By induction hypothesis, both e and y evaluate to the same value in both states. Therefore, we ∀y. \( (\neg \sigma'_1, \text{mem}[j] \land \neg \sigma'_2, \text{mem}[j]) \Rightarrow (\sigma'_1, \text{mem}[j] = \sigma'_2, \text{mem}[j]) \).

2. \( \neg \text{enc}(y) \)
   - The storeL rule generates an assertion \( \neg \text{enc}(y) \Rightarrow (\neg \text{mem}, \text{mem}[\text{var}]) \). Since \( \neg \text{enc}(y) \) holds in this case, this assertion checks that both the address y and value of e have type \( \bot \). By induction hypothesis, both y and e must evaluate to the same value in both states. Therefore, we ∀y. \( (\neg \sigma_1, \text{mem}[j] \land \neg \sigma_2, \text{mem}[j]) \Rightarrow (\sigma_1, \text{mem}[j] = \sigma_2, \text{mem}[j]) \).

Combining all the cases, we prove that storeL preserves property 12.

Next, we prove that storeL also preserves property 13. The storeL rule generates \( \neg \text{enc}(y) \Rightarrow (\neg \text{mem}, \text{mem}[\text{var}]) \). If this statement writes to \( \text{mem} \), then the assertion checks that both the address y and value of e have type \( \bot \). If this assertion is valid, then there does not exist any execution of \( \text{mem}' \Rightarrow \text{store}(\text{mem}, y, e) \) that violates ∀y. \( (\neg \sigma_1, \text{mem}[j] \land \neg \sigma_2, \text{mem}[j]) \Rightarrow (\sigma_1, \text{mem}[j] = \sigma_2, \text{mem}[j]) \).

(egetkey).

We first try proving property 12. Regarding the update to \( \text{mem} \), we have 2 cases:

1. \( \neg \sigma_1, \text{C}_\text{r}, \text{r} \) or \( \neg \sigma_2, \text{C}_\text{r}, \text{r} \):
   - Although rcx evaluates to the same value in \( \sigma_1 \) and \( \sigma_2 \) (from induction hypothesis), the egetkey rule marks 16 bytes starting at \( \sigma_1, \text{rcx} \) as secret. That is, it sets \( \sigma'_1, \text{mem}[\text{rcx}], \ldots, \sigma_1, \text{rcx} + 16 \) and \( \sigma'_2, \text{mem}[\sigma_2, \text{rcx}], \ldots, \sigma_2, \text{rcx} + 16 \) to true, where \( \sigma_1, \text{rcx} = \sigma_2, \text{rcx} \). Therefore, we derive ∀i, j. \( (\neg t_1[i], \text{mem}[j] \land \neg t_2[i], \text{mem}[j]) \Rightarrow (t_1[i], \text{mem}[j] = t_2[i], \text{mem}[j]) \).

2. \( \sigma_1, \text{C}_\text{r}, \text{r} \) or \( \sigma_2, \text{C}_\text{r}, \text{r} \):
   - Although rcx may evaluate to different values in \( \sigma_1 \) and \( \sigma_2 \), the egetkey rule marks 16 bytes starting at \( \sigma_1, \text{rcx} \) and 16 bytes starting at \( \sigma_2, \text{rcx} \) as secret. That is, it sets \( \sigma'_1, \text{mem}[\sigma_1, \text{rcx}], \ldots, \sigma_1, \text{rcx} + 16 \) and \( \sigma'_2, \text{mem}[\sigma_2, \text{rcx}], \ldots, \sigma_2, \text{rcx} + 16 \) to true, where two address ranges may not be identical. From induction hypothesis, we still derive ∀i, j. \( (\neg t_1[i], \text{mem}[j] \land \neg t_2[i], \text{mem}[j]) \Rightarrow (t_1[i], \text{mem}[j] = t_2[i], \text{mem}[j]) \).

Since egetkey does not modify regs, we derive ∀i, r \( \in \text{regs.} \) \( (\neg t_1[i], \text{C}_\text{r}, \land \neg t_2[i], \text{C}_\text{r}) \Rightarrow (t_1[i], r = t_2[i], r) \). Combining with the reasoning on \( \text{mem} \), we prove that property 12 holds inductively.

For the proof of property 13, recall that SGX prevents the programmer from providing a rcx value that points to \( \text{mem} \) i.e. egetkey is guaranteed to write the key to \( \text{mem} \). This constraint is captured in the axioms defining egetkey in \( \text{mem}' \Rightarrow \text{egetkey}(\text{mem}, \text{r}, \text{rcx}) \). Therefore, property 13 holds trivially — \( \text{mem} \) is not updated for any location \( j \) for which \( \neg \text{enc}(j) \) holds.

(ereportH).

A map assignment performs one transition step, hence traces \( t_1 \) and \( t_2 \) contain two states each. \( t_1 = [\sigma_1, \sigma'_1] \), where \( (\sigma_1, \sigma'_1) \in \mathcal{R} (\text{mem}' \Rightarrow \text{ereport}(\text{mem}, \text{rbx}, \text{rcx}, \text{rds})) \). Similarly, \( t_2 = [\sigma_2, \sigma'_2] \), where \( (\sigma_2, \sigma'_2) \in \mathcal{R} (\text{mem}' \Rightarrow \text{ereport}(\text{mem}, \text{rbx}, \text{rcx}, \text{rds})) \).
The ereportL rule assigns $\sigma'_1, C_{\text{mem}}[\sigma_1, \text{rdx} ... \sigma_1, \text{rdx} + 432]$ and $\sigma'_2, C_{\text{case}}[\sigma_2, \text{rdx} ... \sigma_2, \text{rdx} + 432]$ to true. This allows us to derive $\forall j. (\neg \sigma'_1, C_{\text{mem}}[j] \land \neg \sigma'_2, C_{\text{case}}[j]) \Rightarrow (\sigma'_1, \text{mem}[j] = \sigma'_2, \text{mem}[j])$. Furthermore, the ereport instruction doesn’t affect regs, allowing us to derive property 12.

For the proof of property 13, recall that SGX prevents the programmer from providing a rdx value that points to mem-enc i.e. ereport is guaranteed to write the key to mem-enc. This constraint is captured in the axioms defining ereport in mem' = ereport(mem, rbx, rcx, rdx). Therefore, property 13 holds trivially — C_{\text{mem}} is not updated for any location $j$ for which $\neg \text{enc}(j)$ holds.

(ereportL). We first try proving property 12. rcx is a pointer to the 64 bytes of data that will be included in the report. rdx is the base address of the output report. Regarding the update to mem, we have 2 cases:

1. $\neg \sigma_1, C_{\text{enc}}$, and $\neg \sigma_2, C_{\text{enc}}$, and $\neg \sigma_1, C_{\text{db}}$, and $\neg \sigma_2, C_{\text{db}}$.

Since rcx and rdx have the same value in $\sigma_1$ and $\sigma_2$ (from induction hypothesis), the same region of memory will be updated by the instruction. Furthermore, the secrecy level is retained from the input 64-byte region; hence property 12 holds inductively in this case.

2. $\sigma_1, C_{\text{enc}}$ or $\sigma_2, C_{\text{enc}}$, or $\sigma_1, C_{\text{db}}$, or $\sigma_2, C_{\text{db}}$.

Since rcx or rdx may evaluate to different values in $\sigma_1$ and $\sigma_2$. ereportL marks 432 bytes starting at $\sigma_1, \text{rdx}$ and 432 bytes starting at $\sigma_2, \text{rdx}$ as secret. Thus, property 12 holds inductively in this case.

For the proof of property 13, recall that SGX prevents the programmer from providing a rdx value that points to mem-enc i.e. ereport is guaranteed to write the key to mem-enc. This constraint is captured in the axioms defining ereport in mem' = ereport(mem, rbx, rcx, rdx). Therefore, property 13 holds trivially — C_{\text{mem}} is not updated for any location $j$ for which $\neg \text{enc}(j)$ holds.

(exit). exit affects a TCS page in mem-enc, but this page is not accessible by enclave code — it is used by hardware to keep track of security-critical metadata regarding an enclave thread. Furthermore, exit does not affect regs. This constraint is captured in the axioms defining exit in mem' = exit(mem). Therefore, we derive $\forall i, r \in \text{regs}. ((\neg t_1[i], C, \land \neg t_2[i], C) \Rightarrow (t_1[i], r = t_2[i], r))$. The updated enclave page remains at security level $\bot$ because its contents (i.e. the metadata values) are independent of enclave data, and hence independent of enclave secrets — we prove this using our model of SGX. Type $\bot$ implies that the new contents of this page is equivalent in both $\sigma'_1$ and $\sigma'_2$. Therefore, we derive $\forall i, j. ((\neg t_1[i], C_{\text{mem}}[j] \land \neg t_2[i], C_{\text{mem}}[j]) \Rightarrow (t_1[i], \text{mem}[j] = t_2[i], \text{mem}[j])).$ Combining the two derivations, we derive property 12.

As we mentioned above, exit only updates an enclave page in mem-enc — we prove using our model of SGX that this instruction does not write to mem-enc. Therefore, $\forall i, j. \neg \text{enc}(j) \Rightarrow (\neg t_1[i], C_{\text{mem}}[j] \land \neg t_2[i], C_{\text{mem}}[j])$ holds trivially, thus deriving property 13.

(ite). An if-then-else statement is of the form if (e) {$s_1$} else {$s_2$}. We have 2 cases:

1. $[\bot] \vdash (e) \{ s_1 \}$ else $\{ s_e \}$

Hence, $e : \bot$ in $\sigma_1$ and $\sigma_2$ i.e. $\bigwedge_{s \in \text{Vars}(e)} \neg \sigma_1, C_s \land \bigwedge_{s \in \text{Vars}(e)} \neg \sigma_2, C_s$. In this case, the inductive hypothesis implies that $e$ evaluates to the same value in states $\sigma_1$ and $\sigma_2$ — execution follows the same branch in both states. Using the structural induction hypothesis on $s_1$ and $s_e$, we prove that (ite) preserves properties 12 and 13.

2. $[\top] \vdash (e) \{ s_1 \}$ else $\{ s_e \}$

Hence, $e : \top$ in $\sigma_1$ or $\sigma_2$ i.e. $\bigvee_{s \in \text{Vars}(e)} \sigma_1, C_s \lor \bigvee_{s \in \text{Vars}(e)} \sigma_2, C_s$. Since $e$ may evaluate to different values in $\sigma_1$ and $\sigma_2$, execution may follow different branches in the two states. For each register $r$ modified by evaluating if (e) {$s_1$} else {$s_e$} in $\sigma_1$ and $\sigma_2$, the ite rule sets $C_r$ to true in either $\sigma'_1$ or $\sigma'_2$. Therefore, we derive $\forall i, r \in \text{regs}. ((\neg t_1[i], C, \land \neg t_2[i], C) \Rightarrow (t_1[i], r = t_2[i], r))$. Similarly, for each address $j$ in mem modified by evaluating if (e) {$s_1$} else {$s_e$} in $\sigma_1$ and $\sigma_2$, ite sets $C_{\text{mem}}[j]$ to true in either $\sigma'_1$ or $\sigma'_2$. Therefore, we derive $\forall j. ((\neg \sigma'_1, C_{\text{mem}}[j] \land \neg \sigma'_2, C_{\text{mem}}[j]) \Rightarrow (\sigma'_1, \text{mem}[j] = \sigma'_2, \text{mem}[j])).$ Combining the derivations, we prove that property 12 is preserved by ite. Since we must typecheck both $s_1$ and $s_e$ in $\top$ context, ite generates assertions checking that writes are not made to mem-enc. Therefore, we also prove that property 13 is preserved by ite.

(seq). A sequential statement is of the form $s; s'$. $t_1 = [\sigma_1, \sigma'_1, \sigma''_1]$, where $([\sigma_1, \sigma'_1] \in \text{Rs}$ and $([\sigma'_1, \sigma''_1] \in \text{R}s')$. Similarly, $t_2 = [\sigma_2, \sigma'_2, \sigma''_2]$, where $([\sigma_2, \sigma'_2] \in \text{Rs}$ and $([\sigma'_2, \sigma''_2] \in \text{R}s')$. From structural induction, we derive the following:

$$\forall j. (\neg \sigma'_1, C_{\text{mem}}[j] \land \neg \sigma'_2, C_{\text{mem}}[j]) \Rightarrow (\sigma'_1, \text{mem}[j] = \sigma'_2, \text{mem}[j])$$

$$\land \ \forall r \in \text{regs}. ((\neg \sigma'_1, C, \land \neg \sigma'_2, C) \Rightarrow (\sigma'_1, r = \sigma'_2, r))$$

$$\forall j. (\neg \sigma''_1, C_{\text{mem}}[j] \land \neg \sigma''_2, C_{\text{mem}}[j]) \Rightarrow (\sigma''_1, \text{mem}[j] = \sigma''_2, \text{mem}[j])$$

$$\land \ \forall r \in \text{regs}. ((\neg \sigma''_1, C, \land \neg \sigma''_2, C) \Rightarrow (\sigma''_1, r = \sigma''_2, r))$$
Similarly, we leverage structural induction to prove the following:

\[ \forall j. \neg \text{enc}(j) \Rightarrow (\neg \sigma'_1.\text{C}_{\text{mem}}[j] \land \neg \sigma'_2.\text{C}_{\text{mem}}[j]) \] (14)

\[ \forall j. \neg \text{enc}(j) \Rightarrow (\neg \sigma''_1.\text{C}_{\text{mem}}[j] \land \neg \sigma''_2.\text{C}_{\text{mem}}[j]) \] (15)

Therefore, we prove both property 12 and property 13.