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

Sriram Rajamani
Microsoft Research, India
sriram@microsoft.com

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

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 kind 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. Hoekstra 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 `egetkey` 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,

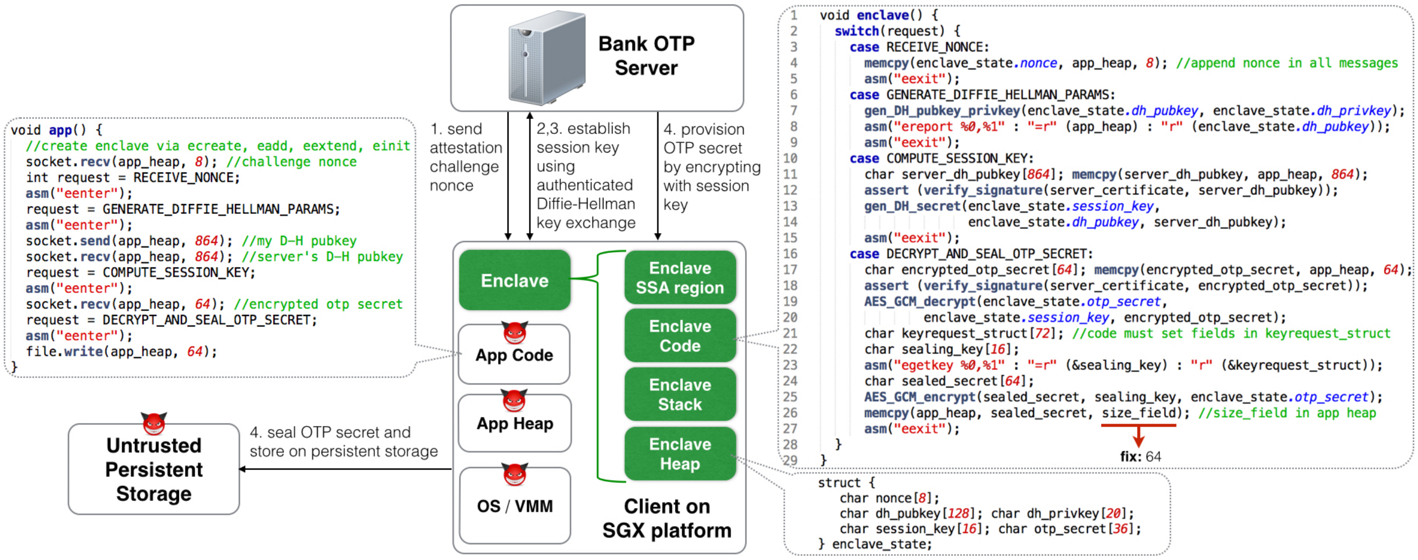

Figure 1: Running OTP Example. The enclave performs trusted cryptographic operations, and the host application performs untrusted tasks such as UI handling and network communications with the OTP server.

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 `eenter` and `eexit` to transfer control back and forth. The host application transfers control to the enclave by invoking `eenter` (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 needs 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 adversary 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 sealing_key. 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.

```
egetkey                     mem := egetkey(mem, rbx, rcx);
movl $0x8080AC,0x8(%esp)    mem := store(mem,add(esp,0x8),0x8080AC);
lea -0x6e0(%ebp),%eax       eax := sub(ebp, 0x6e0);
mov %eax,0x4(%esp)          mem := store(mem,add(esp,0x4),eax);
lea -0x720(%ebp),%eax       eax := sub(ebp, 0x720);
mov %eax,(%esp)             mem := store(mem,esp,eax);
call <AES_GCM_encrypt>      mem := AES_GCM_encrypt(mem, esp);
mov 0x700048,%eax           eax := load(mem,0x700048);
movl %eax,0x8(%esp)         mem := store(mem,add(esp,0x8),eax);
lea -0x720(%ebp),%eax       eax := sub(ebp, 0x720);
mov %eax,0x4(%esp)          mem := store(mem,add(esp,0x4),eax);
movl $0x701000,(%esp)       mem := store(mem,esp,0x701000);
call 802080 <memcpy>        mem := memcpy(mem, esp);
```

Figure 2: OTP enclave snippet (left) and p_{enc} (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 p_{enc}. p_{enc} 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. p_{enc} 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 p_{enc} executes in the presence of an active adversary, we must model the effects of adversarial operations on p_{enc}'s execution. Section 5 defines an active adversary $\mathcal{H}$, which can perform the operation "havoc $\text{mem}_{\neg\text{epc}}$" *once* between consecutive instructions along any execution of p_{enc}. Here, mem_{epc} denotes memory reserved by the SGX processor for enclave use, and $\text{mem}_{\neg\text{epc}}$ is all other memory; havoc $\text{mem}_{\neg\text{epc}}$ updates each address in $\text{mem}_{\neg\text{epc}}$ with a non-deterministically chosen value. We define $\mathcal{H}$ this way because a privileged software adversary can interrupt p_{enc} at any point, perform havoc $\text{mem}_{\neg\text{epc}}$, and then resume p_{enc}. We model the effect of $\mathcal{H}$ on p_{enc}'s behavior by instrumenting p_{enc} to obtain $p_{enc-\mathcal{H}}$ (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 vulnerabilitiy 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, $\mathcal{H}$ uses havoc $\text{mem}_{\neg\text{epc}}$ (in line 8) to choose the number of bytes that $p_{enc-\mathcal{H}}$ 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, $\mathcal{H}$ causes $p_{enc-\mathcal{H}}$ to leak the stack contents, which includes the sealing_key. We can assume for now that writing sealed_secret to the unprotected app_heap is safe because it is encrypted. We formalize a confidentiality property in § 6 that prevents such vulnerabilities, and build a static type system in § 7 which only admits programs that satisfy confidentiality. Confidentiality enforces that for any

```
1  havoc mem¬epc; mem := egetkey(mem, rbx, rcx);
2  havoc mem¬epc; mem := store(mem,add(esp,0x8),0x8080AC);
3  havoc mem¬epc; eax := sub(ebp, 0x6e0);
4  havoc mem¬epc; mem := store(mem,add(esp,0x4),eax);
5  havoc mem¬epc; eax := sub(ebp, 0x720);
6  havoc mem¬epc; mem := store(mem,esp,eax);
7  havoc mem¬epc; mem := AES_GCM_encrypt(mem, esp);
8  havoc mem¬epc; eax := load(mem,0x700048);
9  havoc mem¬epc; mem := store(mem,add(esp,0x8),eax);
10 havoc mem¬epc; eax := sub(ebp, 0x720);
11 havoc mem¬epc; mem := store(mem,add(esp,0x4),eax);
12 havoc mem¬epc; mem := store(mem,esp,0x701000);
13 havoc mem¬epc; mem := memcpy(mem, esp);
```

Figure 3: $p_{enc-\mathcal{H}}$ constructed using OTP p_{enc}

pair of traces of $p_{enc-\mathcal{H}}$ that differ in the values of *Secrets*, if $\mathcal{H}$'s operations along the two traces are equivalent, then $\mathcal{H}$'s observations along the two traces must also be equivalent.

Our type system checks confidentiality by instrumenting $p_{enc-\mathcal{H}}$ 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 $p_{enc-\mathcal{H}}$. 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 independant of *Secrets* (details in § 7). For instance, $\mathsf{C}_{\mathsf{mem}}[\mathsf{esp}]$ in line 13 is assigned to $\mathsf{C}_{\mathsf{eax}} \vee \mathsf{C}_{\mathsf{esp}}$ because either secret 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 $\mathsf{C}_{\mathsf{mem}}$ to *true*. For instance, lines 1-3 assign *true* to those 16 bytes in $\mathsf{C}_{\mathsf{mem}}$ where egetkey places the secret sealing_key. Information leaks can only happen via store to $\text{mem}_{\neg\text{enc}}$, where mem_{enc} is a subset of mem_{epc} that is reserved for use by p_{enc}, and $\text{mem}_{\neg\text{enc}}$ 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 $\text{mem}_{\neg\text{enc}}$. For a program to be well-typed, all assertions in the instrumented $p_{enc-\mathcal{H}}$ 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 $\mathsf{C}_{\mathsf{mem}}$ 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

```
1  C_mem^old := C_mem; havoc C_mem;
2  assume ∀i. (ecx ≤ i < ecx + 16) → C_mem[i];
3  assume ∀i. ¬(ecx ≤ i < ecx + 16) → C_mem[i] ↔ C_mem^old[i];
4  havoc mem_¬epc; mem := egetkey(mem, ebx, ecx);
5  C_mem[add(esp, 0x8)] := C_esp;
6  havoc mem_¬epc; mem := store(mem,add(esp,0x8),0x8080AC);
7  C_eax := C_ebp;
8  havoc mem_¬epc; eax := sub(ebp, 0x6e0);
9  assert ¬enc(add(esp, 0x4)) → ¬C_eax; C_mem[add(esp, 0x4)] := C_eax ∨ C_esp;
10 havoc mem_¬epc; mem := store(mem,add(esp,0x4),eax);
11 C_eax := C_ebp;
12 havoc mem_¬epc; eax := sub(ebp, 0x720);
13 assert ¬enc(esp) → ¬C_eax; C_mem[esp] := C_eax ∨ C_esp;
14 havoc mem_¬epc; mem := store(mem,esp,eax);
15 C_mem := C_AES_GCM_encrypt(C_mem, esp);
16 havoc mem_¬epc; mem := AES_GCM_encrypt(mem, esp);
17 C_eax := C_mem[0x700048];
18 havoc mem_¬epc; eax := load(mem,0x700048);
19 assert ¬enc(add(esp, 0x8)) → ¬C_eax; C_mem[add(esp, 0x8)] := C_eax ∨ C_esp;
20 havoc mem_¬epc; mem := store(mem,add(esp,0x8),eax);
21 C_eax := C_ebp;
22 havoc mem_¬epc; eax := sub(ebp, 0x720);
23 assert ¬enc(add(esp, 0x4)) → ¬C_eax; C_mem[add(esp, 0x4)] := C_eax ∨ C_esp;
24 havoc mem_¬epc; mem := store(mem,add(esp,0x4),eax);
25 C_mem[esp] := C_esp;
26 havoc mem_¬epc; mem := store(mem,esp,0x7001000);
27 C_mem := C_memcpy(C_mem, esp);
28 arg1 := load(mem, esp); arg3 := load(mem, add(esp, 8));
29 assert ∀i. ((arg1 ≤ i < add(arg1, arg3)) ∧ ¬enc(i)) → ¬C_mem[i];
30 havoc mem_¬epc; mem := memcpy(mem, esp);
```

Figure 4: $p_{enc-\mathcal{H}}$ 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 *Declassifed* 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 $\texttt{mem}_{\neg\texttt{epc}}$.

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 combination of *Secrets* and *Declassafied* annotations is refered to as the policy, and is an additional input to Moat.

3.2 Protecting from Infrastructure Attack

We mandate that the enclave be created with the following sequence that measures all pages in $\texttt{mem}_{\texttt{enc}}$.

$$\texttt{ecreate}(size(\texttt{mem}_{\texttt{enc}}));\quad \texttt{foreach}\ page \in \texttt{mem}_{\texttt{enc}} : \{\texttt{eadd}(page);\ \texttt{eextend}(page)\};\quad \texttt{einit} \tag{1}$$

If some component of enclave state is not measured, then the adversary may havoc that component of state during initialization without being detected. We assume that any enclave with the right measurement is equivalent to p_{enc} (by collision resistance assumption). The proof of Theorem 1 in § 5 guarantees that an enclave initialized using the sequence in (1) is protected from privileged adversarial actions such as interrupts, modification of page tables, havoc $\texttt{mem}_{\neg\texttt{epc}}$, and invocation of any x86+SGX instruction — we model each SGX instruction and the adversary as part of this proof (see § 4.2 and § 4.3). The proof guarantees that p_{enc}'s execution (i.e. its set of reachable states) is not affected by the adversary with a caveat that the p_{enc} may read $\texttt{mem}_{\neg\texttt{epc}}$ for inputs. However, we do not consider this to be an infrastructure attack because inputs should be treated as untrusted. Therefore, initializing using the above instruction sequence is sufficient for protecting the enclave from *infrastructure attacks*.

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`, `egetkey`, and `eexit`). We choose BAP for its precise modeling of x86 instructions, which includes updating of CPU flags. We have added a minimal extension to BAP in order to decode SGX instructions. Each microarchitectural instruction from above is modeled in p_{enc} as a sequential composition of BoogiePL [3] statements (syntax described in Figure 5). BoogiePL is an intermediate verification language supporting assertions that can be statically checked for validity using automated theorem provers. Within p_{enc}, Moat uses uninterpreted *Functions* constrained with axioms (described in § 4.2) to model the semantics of each microarchitectural instruction. These axioms describe the effect of microarchitectural instructions on machine state variables *Vars*: main memory `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 $\mathcal{R}(s) \subseteq 2^{\Sigma \times \Sigma}$ over pairs of pre and post states, where $(\sigma, \sigma') \in \mathcal{R}(s)$ if and only if there is an execution of s starting at σ and ending in σ'. We use standard axiomatic semantics for each *Stmt* in Figure 5 [4].

Enclaves have an entrypoint 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 enclave entrypoint, 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}.

$$
\begin{array}{llll}
\mathsf{x}, \mathsf{X} & \in & \mathit{Vars} & \\
q & \in & \mathit{Relations} & \\
f, g, h & \in & \mathit{Functions} & \\
e & \in & \mathit{Expr} & ::= \; \mathsf{x} \mid \mathsf{X} \mid \mathsf{X}[e] \mid f(e, \ldots, e) \\
\phi & \in & \mathit{Formula} & ::= \; \mathsf{true} \mid \mathsf{false} \mid e == e \mid \\
 & & & \quad\;\; q(e, \ldots, e) \mid \phi \wedge \phi \mid \neg\phi \\
s & \in & \mathit{Stmt} & ::= \; \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \\
 & & & \quad\;\; \mathsf{X} := e \mid \mathsf{x} := e \mid \mathsf{X}[e] := e \mid \\
 & & & \quad\;\; \mathsf{if}\ (e)\ \{s\}\ \mathsf{else}\ \{s\} \mid s; s
\end{array}
$$

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 σ is a valuation of all *Vars*, which consists of `mem`, `regs`, and `epcm`. As their names suggest, physical memory σ.`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 ($\mathtt{mem}_{\mathsf{epc}}$), and unprotected memory ($\mathtt{mem}_{\neg\mathsf{epc}}$). For any physical address `a`, `epc(a)` is true iff `a` is an address in $\mathtt{mem}_{\mathsf{epc}}$. Furthermore, $\mathtt{mem}_{\mathsf{enc}}$ is a subset of $\mathtt{mem}_{\mathsf{epc}}$ that is reserved for use by p_{enc} — $\mathtt{mem}_{\mathsf{enc}}$ is virtually addressed and it belongs to the host application's virtual address space. For any virtual address `a`, `enc(a)` is true iff `a` resides in $\mathtt{mem}_{\mathsf{enc}}$. The `epcm` is a finite sized array of hardware-managed structures, where each structure stores security critical metadata about a page in $\mathtt{mem}_{\mathsf{epc}}$. $\mathtt{epcm}_{\mathsf{enc}}$ is a subset of `epcm` that stores metadata about each page in $\mathtt{mem}_{\mathsf{enc}}$ — 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 σ, 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 $\mathsf{x} := \mathsf{X}[\mathsf{y}]$ and writes $\mathsf{X}[\mathsf{y}] := \mathsf{x}$, which are interpreted by the Theory of Arrays [5]. The boolean `ea` denotes whether this access is made by enclave code to $\mathtt{mem}_{\mathsf{enc}}$. If `ea` is true, then `load` asserts (line 14) that the following security checks succeed:

- the translated physical address pa resides in $\texttt{mem}_{\texttt{epc}}$ (line 9)
- **epcm** contains a valid entry for address pa (lines 10 and 11)
- enclave'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 $\texttt{mem}_{\texttt{epc}}$, or if p_{enc} is accessing some other enclave's memory (i.e. within $\texttt{mem}_{\texttt{epc}}$ but outside $\texttt{mem}_{\texttt{enc}}$), 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]bv8, va:bv32)
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 :=  epc(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;
14   assert (ea => check); //EPCM security checks
15   assert ...; //read bit set and pagetable has valid mapping
16   if (!ea && epc(pa)) {return 0xff;} else {return mem[pa];}
17 }
18 function eexit(mem:[bv32]bv8, rbx:bv32)
19 {
20   var mem' := mem; var regs' := regs;
21   regs'[rip] := rbx; regs'[CR_ENCLAVE_MODE] := false;
22   mem'[CR_TCS_PA] := 0x00;
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. p_{enc}'s execution is interleaved with the host application — host application transfers control to p_{enc} via **eenter** or **eresume**, and p_{enc} returns control back to the host application via **eexit**. Control may also transfer from p_{enc} 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 p_{enc}:

1. Havoc all non-enclave memory (denoted by havoc $\texttt{mem}_{\neg\texttt{epc}}$): While the CPU protects the **epc** region, a privileged software adversary can write to any memory location in $\texttt{mem}_{\neg\texttt{epc}}$ region. havoc $\texttt{mem}_{\neg\texttt{epc}}$ is encoded in BoogiePL as:

$$\texttt{assume}\ \forall a.\ \texttt{epc}(a) \rightarrow \texttt{mem}_{new}[a] == \texttt{mem}[a];$$
$$\texttt{mem} := \texttt{mem}_{new};$$

where $\texttt{mem}_{new}$ 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 $\texttt{mem}_{\neg\texttt{epc}}$, havoc $\texttt{mem}_{\neg\texttt{epc}}$ 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:

$$\texttt{regs} := \texttt{regs}_{new};$$

where each register (e.g. $\texttt{rax} \in \texttt{regs}$) 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 p_{enc}, $\mathcal{G}$ may execute an unbounded number of transitions of type* havoc $\texttt{mem}_{\neg\texttt{epc}}$, havoc **regs**, **interrupt**, *or* **call sgx**, *thereby modifying a component $\sigma|_{\mathcal{G}}$ of machine state σ. Following each p_{enc} microarchitectural instruction, $\mathcal{G}$ observes a projection $\sigma|_{\text{obs}}$ of machine state σ. Here, $\sigma|_{\text{obs}} \doteq (\sigma.\texttt{mem}_{\neg\texttt{epc}})$, and $\sigma|_{\mathcal{G}} \doteq (\sigma.\texttt{mem}_{\neg\texttt{enc}}, \sigma.\texttt{regs}, \sigma.\texttt{epcm}_{\neg\texttt{enc}})$.*

DEFINITION 2. **Passive Adversary $\mathcal{P}$.** *The passive adversary $\mathcal{P}$ observes a projection $\sigma|_{\text{obs}}$ of machine state σ after each microarchitectural instruction in p_{enc}. Here, $\sigma|_{\text{obs}} \doteq (\sigma.\texttt{mem}_{\neg\texttt{epc}})$ 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 an absence of exception-causing errors in p_{enc} 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 p_{enc}'s execution in the presence of an adversary ($\mathcal{P}$ or $\mathcal{G}$) by composing their state transition systems. An execution of p_{enc} 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 **cjmp**, unconditional **jmp**, or a usermode SGX instruction (**ereport**, **egetkey**, or **eexit**). Since p_{enc} is loop-free, each statement s_i has a distinct label 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 $\mathcal{P}$.
In the presence of $\mathcal{P}$, p_{enc} undergoes a deterministic sequence of state transitions starting from initial state σ_0. $\mathcal{P}$ cannot update $Vars$, therefore $\mathcal{P}$ affects p_{enc}'s execution only via the initial state σ_0. We denote this sequence of states as trace $t = [\sigma_0, \sigma_1, \ldots, \sigma_n]$, where $(\sigma_i, \sigma_{i+1}) \in \mathcal{R}(s_i)$ for each $i \in 0, \ldots, n-1$. We also write this as $\langle p_{enc}, \sigma_0\rangle \Downarrow t$.

Composing enclave p_{enc} with active adversary $\mathcal{G}$.
$\mathcal{G}$ 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 $\mathcal{G}$, we consider the following composition of p_{enc} and $\mathcal{G}$. For each p_{enc} statement $l: s$, we transform it to:

$$adv_1;\ \ldots;\ adv_k;\ l:\ s \tag{2}$$

This instrumentation guarantees that between any consecutive statements along an execution of p_{enc}, $\mathcal{G}$ 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 $\mathtt{mem}_{\neg\mathsf{epc}}$, havoc **regs**, **interrupt**, or **call sgx**. This composed model, hereby called $p_{enc-\mathcal{G}}$, encodes all possible behaviours of p_{enc} in the presence of $\mathcal{G}$. An execution of $p_{enc-\mathcal{G}}$ is described by a sequence of states i.e. trace $t = [\alpha_0, \sigma_0, \alpha_1, \sigma_1, \ldots, \alpha_n, \sigma_n]$, where each $\alpha_i \in t$ denotes the state after the last adversary transition adv_k (right before execution resumes in the enclave). We coalesce the effect of all adversary transitions into a single state α_i for cleaner notation. Following adv_k, the composed model $p_{enc-\mathcal{G}}$ executes an enclave statement $l: s$, taking the system from a state α_i to state σ_i.
Given a trace $t = [\alpha_0, \sigma_0, \alpha_1, \sigma_1, \ldots, \alpha_n, \sigma_n]$, we define

$$t|_{\mathsf{obs}} \doteq [\sigma_0|_{\mathsf{obs}}, \sigma_1|_{\mathsf{obs}}, \ldots, \sigma_n|_{\mathsf{obs}}]$$

denoting the adversary-observable projection of trace t, ignoring the adversary controlled α states. Correspondingly, we define

$$t|_{\mathcal{G}} \doteq [\alpha_0|_{\mathcal{G}}, \alpha_1|_{\mathcal{G}}, \ldots, \alpha_n|_{\mathcal{G}}]$$

capturing the adversary's effects within a trace t. We define the enclave projection of σ to be

$$\sigma|_{\mathsf{enc}} \doteq (\sigma.\mathtt{mem}_{\mathsf{enc}}, \sigma.\mathtt{regs}, \sigma.\mathtt{epcm}_{\mathsf{enc}})$$

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

$$t|_{\mathsf{enc}} \doteq [\sigma_0|_{\mathsf{enc}}, \sigma_1|_{\mathsf{enc}}, \ldots, \sigma_n|_{\mathsf{enc}}]$$

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 $p_{enc-\mathcal{G}}$, it unsoundly limits the $\mathcal{G}$'s capabilities.
- Use alternative adversary models in lieu of $\mathcal{G}$ 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 $\mathcal{P}$ for $\mathcal{G}$. This would be the equivalent of making k equal 0, and thus $p_{enc-\mathcal{G}}$ bounded in length. However, for this to be sound, we must prove that $\mathcal{G}$'s operations can be removed without affecting p_{enc}'s execution, as required by the following property.

$$\forall \sigma \in \Sigma.\ \forall t_i, t_j \in \Sigma^*.\ \langle p_{enc-\mathcal{G}}, \sigma\rangle \Downarrow t_i \wedge \langle p_{enc}, \sigma\rangle \Downarrow t_j \ \Rightarrow \ \forall i.\ t_i|_{\mathsf{enc}}[i] = t_j|_{\mathsf{enc}}[i] \tag{3}$$

If property (3) holds, then we can substitute $\mathcal{P}$ for $\mathcal{G}$ while proving any safety (or k-safety [12]) property of p_{enc}. While attempting to prove this property in the Boogie verifier [3], we quite expectedly discovered counter-examples that illustrate the different ways in which $\mathcal{G}$ affects p_{enc}'s execution:

1. Enclave instruction $\mathtt{load}(\mathtt{mem}, a)$, where a is an address in $\mathtt{mem}_{\neg\mathsf{epc}}$. $\mathcal{G}$ havocs $\mathtt{mem}_{\neg\mathsf{epc}}$ and p_{enc} reads $\mathtt{mem}_{\neg\mathsf{epc}}$ for inputs, so this counter-example is not surprising.
2. $\mathtt{load}(\mathtt{mem}, a)$, where a is an address within SSA pages. $\mathcal{G}$ 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 $\mathtt{mem}_{\neg\mathsf{epc}}$ 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 $\mathtt{mem}_{\neg\mathsf{epc}}$) is too restrictive in practice because p_{enc} must read $\mathtt{mem}_{\neg\mathsf{epc}}$ to receive inputs. Therefore, we must explore alternative adversary models. Instead of replacing $\mathcal{G}$ with $\mathcal{P}$, we attempt replacing $\mathcal{G}$ with $\mathcal{H}$ defined below.

Definition 3. **Havocing Active Adversary $\mathcal{H}$.** *Between any consecutive statements along an execution of p_{enc}, $\mathcal{H}$ may execute a single* havoc $\mathtt{mem}_{\neg\mathsf{epc}}$ *operation, thereby modifying a component $\sigma|_{\mathcal{H}}$ of machine state σ. Following each p_{enc} microarchitectural instruction, $\mathcal{H}$ observes a projection $\sigma|_{\mathsf{obs}}$ of machine state σ. Here, $\sigma|_{\mathsf{obs}} \doteq (\sigma.\mathtt{mem}_{\neg\mathsf{epc}})$, and $\sigma|_{\mathcal{H}} \doteq (\sigma.\mathtt{mem}_{\neg\mathsf{enc}})$.*

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

$$\mathsf{havoc}\ \mathtt{mem}_{\neg\mathsf{epc}};\ l:\ s \tag{4}$$

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

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

$$\forall \sigma \in \Sigma.\ \forall t_1 \in \Sigma^*.\ \langle p_{enc-\mathcal{G}}, \sigma\rangle \Downarrow t_1 \Rightarrow \exists t_2 \in \Sigma^*.\ \langle p_{enc-\mathcal{H}}, \sigma\rangle \Downarrow t_2 \ \wedge\ \forall i.\ t_1|_{\mathsf{enc}}[i] = t_2|_{\mathsf{enc}}[i]$$

Validity of this theorem implies that we can replace $\mathcal{G}$ with $\mathcal{H}$ 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 $\mathcal{G}$ by instrumenting an unbounded number of adversary operations $adv_1; \ldots; adv_k$ before each statement in p_{enc}. Now, let us further instrument havoc $\texttt{mem}_{\neg\mathsf{epc}}$ after each $adv_i \in \{adv_1; \ldots; adv_k\}$ — this is sound because a havoc on $\texttt{mem}_{\neg\mathsf{epc}}$ does not restrict the allowed values of $\texttt{mem}_{\neg\mathsf{epc}}$. The resulting instrumentation for each statement $l:\ s$ is:

$$adv_1; \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}}; \ldots; adv_k; \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}};\ l:\ s \tag{5}$$

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 $\texttt{mem}_{\neg\mathsf{epc}}$. In order to define lemma 1, we introduce the following transformation on each statement $l : s$ of p_{enc}:

$$\mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}}; \ldots; \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}};\ l:\ s \tag{6}$$

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

$$\begin{aligned}&\forall \sigma \in \Sigma.\ \forall t_1 \in \Sigma^*.\ \langle p_{enc-\mathcal{G}*}, \sigma\rangle \Downarrow t_1 \Rightarrow \\ &\qquad \exists t_2 \in \Sigma^*.\ \langle p_{enc-\mathcal{H}*}, \sigma\rangle \Downarrow t_2 \wedge \forall i.\ t_1|_{\mathsf{enc}}[i] = t_2|_{\mathsf{enc}}[i]\end{aligned}$$

Proof: The intuition behind this proof is that the other adversarial transitions do not affect p_{enc} in any way that is not simulated by havoc $\texttt{mem}_{\neg\mathsf{epc}}$. 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 (σ_i, σ_j) that is indistinguishable to the enclave, we prove that after one transition, the new pair of states (σ_i', σ_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 $p_{enc-\mathcal{G}*}$ and s_j is executed by $p_{enc-\mathcal{H}*}$. The state predicate $Init$ represents an enclave state after invoking $\texttt{einit}$ in the prescribed initialization sequence in (1).

$$\forall \sigma_i, \sigma_j. Init(\sigma_i) \wedge Init(\sigma_j) \Rightarrow \sigma_i|_{\mathsf{enc}} = \sigma_j|_{\mathsf{enc}} \tag{7}$$

$$\begin{aligned}&\forall \sigma_i, \sigma_j, \sigma_i', \sigma_j', s_i, s_j. \\ &\sigma_i|_{\mathsf{enc}} = \sigma_j|_{\mathsf{enc}} \wedge (\sigma_i, \sigma_i') \in \mathcal{R}(s_i) \wedge (\sigma_j, \sigma_j') \in \mathcal{R}(s_j) \wedge p(s_i, s_j) \\ &\Rightarrow \sigma_i'|_{\mathsf{enc}} = \sigma_j'|_{\mathsf{enc}}\end{aligned} \tag{8}$$

where

$$p(s_i, s_j) \doteq \begin{cases} s_i \in \{\texttt{egetkey}, \texttt{ereport}, \texttt{eexit}, \texttt{load}, \texttt{store}\} \wedge s_j = s_i \\ s_i = s; \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}} \wedge s_j = \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}} \\ \quad \text{where } s \in \{\mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}}, \ldots, \texttt{interrupt}, \texttt{call sgx}\}\end{cases}$$

Lemma 2. *A sequential composition of unbounded number of* havoc $\texttt{mem}_{\neg\mathsf{epc}}$ *statements can be simulated by a single* havoc $\texttt{mem}_{\neg\mathsf{epc}}$ *statement.*

Combining lemma 1 and lemma 2, we prove that the effect of $adv_1; \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}}; \ldots; adv_k; \mathsf{havoc}\ \texttt{mem}_{\neg\mathsf{epc}}$ (or adv_1; adv_2;...; adv_n) on enclave's execution can be simulated by havoc $\texttt{mem}_{\neg\mathsf{epc}}$. By theorem 1, it is sound to prove any safety (or k-safety) property on $p_{enc-\mathcal{H}}$ because $p_{enc-\mathcal{H}}$ allows all traces allowed by $p_{enc-\mathcal{G}}$. The benefits of composing with $\mathcal{H}$ are (1) $p_{enc-\mathcal{H}}$ 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) $\mathcal{H}$ gives a convenient mental 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 $\mathcal{H}$ cannot distinguish between p_{enc} and an enclave that executes skip in each step. However, such definition prevents p_{enc} from writing to $\texttt{mem}_{\neg\mathsf{epc}}$, 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 $\texttt{mem}_{\neg\mathsf{epc}}$, but constraining the values to be independent of the secrets. An input to Moat is a policy that defines $Secrets = \{(l, v) \mid l \in L, v \in Vars\}$, 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 $p_{enc-\mathcal{H}}$ to $p_{enc-\mathcal{H}-sec}$ for formalizing confidentiality. For each $(l, v) \in Secrets$, we transform the statement $l : s$ to:

$$l:\ s;\ \mathsf{havoc}\ v; \tag{9}$$

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 $p_{enc-\mathcal{H}-sec}$ that potentially differ in the values of the Secret variables, if $\mathcal{H}$'s operations along the two traces are equivalent, then $\mathcal{H}$'s observations along the two traces must also be equivalent.*

$$\begin{aligned}&\forall \sigma \in \Sigma, t_1, t_2 \in \Sigma^*.(\langle p_{enc-\mathcal{H}-sec}, \sigma\rangle \Downarrow t_1 \wedge \langle p_{enc-\mathcal{H}-sec}, \sigma\rangle \Downarrow t_2 \wedge \\ &\qquad \forall i. t_1|_{\mathcal{H}}[i] = t_2|_{\mathcal{H}}[i]) \Rightarrow (\forall i. t_1|_{\mathsf{obs}}[i] = t_2|_{\mathsf{obs}}[i])\end{aligned} \tag{10}$$

The havoc on $Secrets$ cause the secret variables to take potentially differing symbolic values in t_1 and t_2. However, property (10) requires $t_1|_{\mathsf{obs}}$ and $t_2|_{\mathsf{obs}}$ to be equivalent, which is achieved only if secrets do not leak to $\mathcal{H}$-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 $Declassified = \{(l, v) \mid l \in L, v \in Vars\}$, 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 $p_{enc-\mathcal{H}-sec}$ that potentially differ in the values of the Secret variables, if $\mathcal{H}$'s operations along the two traces are equivalent, then $\mathcal{H}$'s observations (ignoring Declassified outputs) along the two traces must also be equivalent.*

$$\begin{aligned}&\forall \sigma \in \Sigma, t_1, t_2 \in \Sigma^*.(\langle p_{enc-\mathcal{H}-sec}, \sigma\rangle \Downarrow t_1\ \wedge \langle p_{enc-\mathcal{H}-sec}, \sigma\rangle \Downarrow t_2 \wedge \\ &\qquad \forall i. t_1|_{\mathcal{H}}[i] = t_2|_{\mathcal{H}}[i]) \Rightarrow \\ &\qquad \forall i, j.\ \neg\mathsf{epc}(j) \Rightarrow ((i, \texttt{mem}[j]) \in Declassified \\ &\qquad \vee\ t_1|_{\mathsf{obs}}[i].\texttt{mem}[j] = t_2|_{\mathsf{obs}}[i].\texttt{mem}[j])\end{aligned} \tag{11}$$

7. PROVING CONFIDENTIALITY

Our goal is to automatically check if $p_{enc-\mathcal{H}}$ satisfies confidentiality (property 11), which would ensure safety against *application attacks*. Since confidentiality is a 2-safety hyperproperty (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 $p_{enc-\mathcal{H}}$ satisfies confidentiality. We avoid a self-composition approach because of complications in encoding equivalence assumptions over adversary operations in the two traces of $p_{enc-\mathcal{H}-sec}$ (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 $\mathcal{H}$-observable state i.e. $\sigma|_{\mathsf{obs}}$. For instance, the program $\mathsf{mem} := \mathtt{store}(\mathsf{mem}, \mathsf{y}, \mathsf{x})$ is ill-typed if x's value depends on a secret and $\mathsf{enc}(\mathsf{y})$ 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 $\mathcal{H}$-visible state in either branch. For instance, the program $\mathsf{if}\ (\mathsf{x} == 42)\ \{\mathsf{mem} := \mathtt{store}(\mathsf{mem}, \mathsf{y}, 1)\}\ \mathsf{else}\ \{\mathsf{skip}\}$ is ill-typed if x's value depends on a secret and $\mathsf{enc}(\mathsf{y})$ is false. In both examples above, $\mathcal{H}$ learns the secret value x by reading mem at address y. In addition to the `store` instruction, explicit and implicit leaks may also be caused by unsafe use of SGX instructions. For instance, `egetkey` returns a secret sealing key, which must not be leaked from the enclave. Similarly, `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 $p_{enc-\mathcal{H}}$ only if it has no implicit or explicit leaks.

A security type is either $\top$ (secret) or $\perp$ (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 `regs`) may alternate between holding secret and public values. As explained later in this section, Moat needs the security types in order to decide whether a `store` instruction causes implicit or explicit leaks. $p_{enc-\mathcal{H}}$ accompanies a policy containing $Secrets = \{(l, v)\}$ and $Declassified = \{(l, v)\}$, where a tuple (l, v) denotes that variable v at program location l contains a secret and declassified value, respectively. However, there are no other type declarations; therefore, 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.

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

For a statement s to have type τ, every assignment in s must update a state variable whose security class is τ or higher. We write this typing judgment as $[\tau] \vdash s \Rightarrow \langle \psi, \mathcal{F} \rangle$, where ψ is a SMT formula and $\mathcal{F}$ is a set of SMT formulae. Each satisfiable interpretation of ψ corresponds to a feasible execution of s. $\mathcal{F}$ contains a SMT formula for each `store` instruction in s, such that the formula is valid iff the `store` does not leak secrets. We present our typing rules in Figure 7, which assume that $p_{enc-\mathcal{H}}$ is first converted to single static assignment form. s has type τ if we derive $[\tau] \vdash s \Rightarrow \langle \psi, \mathcal{F} \rangle$ using the typing rules, and prove that all formulae in $\mathcal{F}$ are valid. If s has type $\top$, then s does not update $\mathcal{H}$-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 $\perp$ implies that s either does not update $\mathcal{H}$-observable state or the update is independent of *Secrets*.

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

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

Moat implements this type system by replacing each statement s in $p_{enc-\mathcal{H}}$ by $I(s)$ using the instrumentation rules in Figure 8. Observe that we introduce $\mathsf{C_{pc}}$ 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 $\mathsf{C_{pc}}$ to *true* within the then and else branches. Moat invokes $I(p_{enc-\mathcal{H}})$ and applies the instrumentation rules in Figure 8 recursively. Figure 4 demonstrates an example of instrumenting $p_{enc-\mathcal{H}}$. Moat then feeds the instrumented program $I(p_{enc-\mathcal{H}})$ to an off-the-shelf program verifier, which proves validity all assertions or finds a counter-example. Our implementation uses the Boogie [3] program verifier, which receives $I(p_{enc-\mathcal{H}})$ 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, Moat's type system is inspired by the type-based approach for information flow checking by Volpano et

$$\frac{}{\vdash e : \top \Rightarrow \mathit{true}}\,(\mathit{exp1}) \qquad \frac{}{\vdash e : \bot \Rightarrow \bigwedge_{\mathsf{v} \in Vars(e)} \neg \mathsf{C_v}}\,(\mathit{exp2})$$

$$\frac{[\top] \vdash s \Rightarrow \langle \psi, A \rangle}{[\bot] \vdash s \Rightarrow \langle \psi, A \rangle}\,(\mathit{coercion}) \qquad \frac{}{[\top] \vdash \mathsf{skip} \Rightarrow \langle \mathit{true}, \{\emptyset\} \rangle}\,(\mathit{skip})$$

$$\frac{}{[\tau] \vdash \mathtt{assume}\ \phi \Rightarrow \langle \phi, \{\emptyset\} \rangle}\,(\mathit{assume}) \qquad \frac{}{[\tau] \vdash \mathtt{assert}\ \phi \Rightarrow \langle \phi, \{\emptyset\} \rangle}\,(\mathit{assert})$$

$$\frac{}{[\tau] \vdash \mathsf{x'} := e \Rightarrow \langle (\mathsf{x'} = e) \wedge (\mathsf{C_{x'}} \leftrightarrow \bigvee_{\mathsf{v} \in Vars(e)} \mathsf{C_v}), \{\emptyset\} \rangle}\,(\mathit{scalar})$$

$$\frac{}{\begin{array}{l}[\tau] \vdash \mathsf{x'} := \mathtt{load}(\mathtt{mem}, e) \Rightarrow \\ \langle (\mathsf{x'} = \mathtt{load}(\mathtt{mem}, e)) \wedge (\mathsf{C_{x'}} \leftrightarrow \mathsf{C_{mem}}[e] \vee \bigvee_{\mathsf{v} \in Vars(e)} \mathsf{C_v}), \{\emptyset\} \rangle\end{array}}\,(\mathit{load})$$

$$\frac{}{\begin{array}{l}[\top] \vdash \mathtt{mem'} := \mathtt{store}(\mathtt{mem}, \mathsf{y}, e) \Rightarrow \\ \langle \mathtt{mem'} = \mathtt{store}(\mathtt{mem}, \mathsf{y}, e) \ \wedge \mathsf{C_{mem'}} = \mathsf{C_{mem}}[\mathsf{y} := \mathit{true}], \\ \{\mathsf{enc}(\mathsf{y})\} \rangle\end{array}}\,(\mathit{storeH})$$

$$\frac{\vdash e : \bot \Rightarrow \psi_1 \qquad \vdash y : \bot \Rightarrow \psi_2}{\begin{array}{l}[\bot] \vdash \mathtt{mem'} := \mathtt{store}(\mathtt{mem}, \mathsf{y}, e) \Rightarrow \\ \langle \mathtt{mem'} = \mathtt{store}(\mathtt{mem}, \mathsf{y}, e) \wedge \mathsf{C_{mem'}} = \mathsf{C_{mem}}[\mathsf{y} := \psi_1 \wedge \psi_2], \\ \{\neg \mathsf{enc}(\mathsf{y}) \rightarrow (\psi_1 \wedge \psi_2)\} \rangle\end{array}}\,(\mathit{storeL})$$

$$\frac{}{\begin{array}{l}[\bot] \vdash \mathtt{mem'} := \mathtt{ereport}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx}) \Rightarrow \\ \langle \mathtt{mem'} = \mathtt{ereport}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx}) \ \wedge \\ \forall i.\ (\mathsf{rdx} \leq i < \mathsf{rdx} + 320) \rightarrow \mathsf{C_{mem'}}[i] \leftrightarrow \mathsf{C_{rdx}} \\ \ \wedge \ (\mathsf{rdx} + 320 \leq i < \mathsf{rdx} + 384) \rightarrow \\ \qquad \mathsf{C_{mem'}}[i] \leftrightarrow (\mathsf{C_{rcx}} \vee \mathsf{C_{rdx}} \vee \mathsf{C_{mem}}[\mathsf{rcx} + i - \mathsf{rdx} - 320]) \\ \ \wedge \ (\mathsf{rdx} + 384 \leq i < \mathsf{rdx} + 432) \rightarrow \mathsf{C_{mem'}}[i] \leftrightarrow \mathsf{C_{rdx}} \\ \ \wedge \ \neg(\mathsf{rdx} \leq i < \mathsf{rdx} + 432) \rightarrow \mathsf{C_{mem'}}[i] \leftrightarrow \mathsf{C_{mem}}[i], \\ \{\emptyset\} \rangle\end{array}}\,(\mathit{ereportL})$$

$$\frac{}{\begin{array}{l}[\top] \vdash \mathtt{mem'} := \mathtt{ereport}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx}) \Rightarrow \\ \langle \mathtt{mem'} = \mathtt{ereport}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx}) \ \wedge \\ \forall i.\ (\mathsf{rdx} \leq i < \mathsf{rdx} + 320) \rightarrow \mathsf{C_{mem'}}[i] \\ \ \wedge \ (\mathsf{rdx} + 320 \leq i < \mathsf{rdx} + 384) \rightarrow \mathsf{C_{mem'}}[i] \\ \ \wedge \ (\mathsf{rdx} + 384 \leq i < \mathsf{rdx} + 432) \rightarrow \mathsf{C_{mem'}}[i] \\ \ \wedge \ \neg(\mathsf{rdx} \leq i < \mathsf{rdx} + 432) \rightarrow \mathsf{C_{mem'}}[i] \leftrightarrow \mathsf{C_{mem}}[i], \\ \{\emptyset\} \rangle\end{array}}\,(\mathit{ereportH})$$

$$\frac{}{\begin{array}{l}[\tau] \vdash \mathtt{mem'} := \mathtt{egetkey}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}) \Rightarrow \\ \langle \mathtt{mem'} = \mathtt{egetkey}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}) \ \wedge \\ \forall i.\ (\mathsf{rcx} \leq i < \mathsf{rcx} + 16) \rightarrow \mathsf{C_{mem'}}[i] \\ \ \wedge \ \neg(\mathsf{rcx} \leq i < \mathsf{rcx} + 16) \rightarrow \mathsf{C_{mem'}}[i] \leftrightarrow \mathsf{C_{mem}}[i], \\ \{\emptyset\} \rangle\end{array}}\,(\mathit{egetkey})$$

$$\frac{}{\begin{array}{l}[\tau] \vdash \mathtt{mem'}, \mathtt{regs'} := \mathtt{eexit}(\mathtt{mem}) \Rightarrow \\ \langle (\mathtt{mem'}, \mathtt{regs'}) = \mathtt{eexit}(\mathtt{mem}), \{\forall \mathsf{r} \in \mathtt{regs}.\ \neg \mathsf{C_r}\} \rangle\end{array}}\,(\mathit{eexit})$$

$$\frac{[\tau] \vdash s_1 \Rightarrow \langle \psi_1, \mathcal{F}_1 \rangle \qquad [\tau] \vdash s_2 \Rightarrow \langle \psi_2, \mathcal{F}_2 \rangle}{[\tau] \vdash s_1; s_2 \Rightarrow \langle \psi_1 \wedge \psi_2, \mathcal{F}_1 \cup \{\psi_1 \rightarrow f_2 \mid f_2 \in \mathcal{F}_2\} \rangle}\,(\mathit{seq})$$

$$\frac{\vdash e : \tau \Rightarrow \psi \qquad [\tau] \vdash s_1 \Rightarrow \langle \psi_1, \mathcal{F}_1 \rangle \qquad [\tau] \vdash s_2 \Rightarrow \langle \psi_2, \mathcal{F}_2 \rangle}{\begin{array}{l}[\tau] \vdash \mathsf{if}\ (e)\ \{s_1\}\ \mathsf{else}\ \{s_2\} \Rightarrow \\ \langle (e \rightarrow \psi_1) \wedge (\neg e \rightarrow \psi_2), \\ \{\psi\} \cup \{e \rightarrow f_1 \mid f_1 \in \mathcal{F}_1\} \cup \{\neg e \rightarrow f_2 \mid f_2 \in \mathcal{F}_2\} \rangle\end{array}}\,(\mathit{ite})$$

Figure 7: Typing Rules for $p_{enc-\mathcal{H}}$

Statement s	Instrumented Statement $I(s)$
`assert` ϕ	`assert` ϕ
`assume` ϕ	`assume` ϕ
skip	skip
$\mathsf{x} := e$	$\mathsf{C_x} := \mathsf{C_{pc}} \vee \bigvee_{v \in Vars(e)} \mathsf{C_v}$; $\mathsf{x} := e$
$\mathsf{x} := \mathtt{load}(\mathtt{mem}, e)$	$\mathsf{C_x} := \mathsf{C_{pc}} \vee \mathsf{C_{mem}}[e] \vee \bigvee_{v \in Vars(e)} \mathsf{C_v}$; $\mathsf{x} := \mathtt{load}(\mathtt{mem}, e)$
$\mathtt{mem} :=$ $\mathtt{store}(\mathtt{mem}, \mathsf{y}, e)$	`assert` $\mathsf{C_{pc}} \rightarrow \mathsf{enc}(\mathsf{y})$; `assert` $(\neg \mathsf{C_{pc}} \wedge \neg \mathsf{enc}(\mathsf{y})) \rightarrow (\bigwedge_{v \in Vars(e) \cup Vars(\mathsf{y})} \neg \mathsf{C_v})$; $\mathsf{C_{mem}}[\mathsf{y}] := \mathsf{C_{pc}} \vee \bigvee_{v \in Vars(e) \cup Vars(\mathsf{y})} \mathsf{C_v}$; $\mathtt{mem} := \mathtt{store}(\mathtt{mem}, \mathsf{y}, e)$
$\mathtt{mem} :=$ `ereport` $(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx})$	$\mathsf{C^{old}_{mem}} := \mathsf{C_{mem}}$; havoc $\mathsf{C_{mem}}$; `assume` $\forall i.\ (\mathsf{rdx} \leq i < \mathsf{rdx} + 320) \rightarrow \mathsf{C_{mem}}[i] = \mathsf{C_{pc}} \vee \mathsf{C_{rdx}}$; `assume` $\forall i.\ (\mathsf{rdx} + 320 \leq i < \mathsf{rdx} + 384) \rightarrow$ $\mathsf{C_{mem}}[i] = (\mathsf{C_{pc}} \vee \mathsf{C_{rcx}} \vee \mathsf{C_{rdx}} \vee \mathsf{C^{old}_{mem}}[\mathsf{rcx} + i - \mathsf{rdx} - 320])$; `assume` $\forall i.\ (\mathsf{rdx} + 384 \leq i < \mathsf{rdx} + 432) \rightarrow$ $\mathsf{C_{mem}}[i] = \mathsf{C_{pc}} \vee \mathsf{C_{rdx}}$; `assume` $\forall i.\ \neg(\mathsf{rdx} \leq i < \mathsf{rdx} + 432) \rightarrow \mathsf{C_{mem}}[i] = \mathsf{C^{old}_{mem}}[i]$; $\mathtt{mem} := \mathtt{ereport}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx})$
$\mathtt{mem} :=$ `egetkey` $(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx})$	$\mathsf{C^{old}_{mem}} := \mathsf{C_{mem}}$; havoc $\mathsf{C_{mem}}$; `assume` $\forall i.\ (\mathsf{rcx} \leq i < \mathsf{rcx} + 16) \rightarrow \mathsf{C_{mem}}[i]$; `assume` $\forall i.\ \neg(\mathsf{rcx} \leq i < \mathsf{rcx} + 16) \rightarrow \mathsf{C_{mem}}[i] = \mathsf{C^{old}_{mem}}[i]$; $\mathtt{mem} := \mathtt{egetkey}(\mathtt{mem}, \mathsf{rbx}, \mathsf{rcx})$
$\mathtt{mem}, \mathtt{regs} :=$ $\mathtt{eexit}(\mathtt{mem}, \mathsf{rbx})$	`assert` $\forall \mathsf{r} \in \mathtt{regs}.\ \neg \mathsf{C_r}$; $\mathtt{mem}, \mathtt{regs} := \mathtt{eexit}(\mathtt{mem})$
$s_1; s_2$	$I(s_1); I(s_2)$
$\mathsf{if}(e)\{s_1\}\mathsf{else}\{s_2\}$	$\mathsf{C^{in}_{pc}} := \mathsf{C_{pc}}$; $\mathsf{C_{pc}} := \mathsf{C_{pc}} \vee \bigvee_{v \in Vars(e)} \mathsf{C_v}$; if (e) $\{I(s_1)\}$ else $\{I(s_2)\}$; $\mathsf{C_{pc}} := \mathsf{C^{in}_{pc}}$

Figure 8: Instrumentation rules for $p_{enc-\mathcal{H}}$

al. [26]. The core modifications to their system are as follows:

- Our type system includes rules for SGX instructions `ereport`, `egetkey`, and `eexit`. The rules precisely model the memory locations written by these these instructions, and whether the produced data is public or confidential.

- Our type system is flow-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 program verifier for exploring all executions of the instrumented $p_{enc-\mathcal{H}}$.

- Our type system includes rules updating unbounded array variables (e.g. `mem`), without requiring that all indices in the array take the same security type.

8. EVALUATION AND EXPERIENCE

Moat's implementation comprises (1) translation from x86 + SGX program to p_{enc} using BAP, (2) transformation to $p_{enc-\mathcal{H}}$ using instrumentation in (4), (3) transformation to $I(p_{enc-\mathcal{H}})$ using Figure 8, and (4) invoking Boogie/Z3 Theorem Prover to prove validity of all assertions in $I(p_{enc-\mathcal{H}})$ (modulo declassifications from the protocol verification step). Although the enclave code contains calls to the cryptographic library (we use cryptopp), p_{enc} abstracts them away as uninterpreted functions i.e. we do not verify the cryptographic implementation. Having said that, this was merely a pragmatic choice, and not a fundamental weakness of our approach. We now describe some case studies which we verified using Moat and ProVerif in tandem. We upload these case studies at [1], and summarize the results in Figure 9.

We use the following standard crytpographic notation and assumptions. $m_1 | \ldots | m_n$ denotes tagged concatenation of n

messages. We use a keyed-hash message authentication function $\mathsf{MAC}_k(text)$ and hash function $\mathsf{H}(text)$, 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 $\mathsf{Sig}_k(text)$; we assume unforgeability under chosen message attacks. Intel provisions each SGX processor with a unique private key K_{SGX}^{-1} 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 $\mathsf{Quote}_e(text)$, which is equivalent to $\mathsf{Sig}_{K_{SGX}^{-1}}(\mathsf{H}(text) \mid M_e)$ — measurement of enclave e is written as M_e. N is used to denote nonce. Finally, we write $\mathsf{Enc}_k(text)$ for the encryption of $text$, for which we assume indistinguishability under chosen plaintext attack. We also use $\mathsf{AEnc}_k(text)$ 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 provisiong protocol (from § 2), where *client* runs in a SGX enclave, *bank* is an uncompromised service, and *disk* is under adversary control:

$$bank \rightarrow client : N$$
$$client \rightarrow bank : N \mid g^c \mid \mathsf{Quote}_{client}(N \mid g^c)$$
$$bank \rightarrow client : N \mid g^b \mid \mathsf{Sig}_{K_{bank}^{-1}}(N \mid g^b) \mid \mathsf{AEnc}_{\mathsf{H}(g^{bc})}(secret)$$
$$client \rightarrow disk : \mathsf{AEnc}_{K_{seal}}(secret)$$

First, we use Moat to prove that g^{bc} and K_{seal} are not leaked to $\mathcal{H}$. Next, ProVerif uses secrecy assumption on g^{bc} and K_{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 nondeterministically 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.

$$user \rightarrow notary : \mathsf{connect} \mid N$$
$$notary \rightarrow user : \mathsf{Quote}_{notary}(N)$$
$$user \rightarrow notary : \mathsf{notarize} \mid \mathsf{H}(text)$$
$$notary \rightarrow user : \mathsf{counter} \mid \mathsf{H}(text) \mid \mathsf{Sig}_{K_{notary}^{-1}}(\mathsf{counter} \mid \mathsf{H}(text))$$

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

$$A \rightarrow B : g^{a_1} \mid \mathsf{Sig}_{K_A^{-1}}(g^{a_1}) \mid \mathsf{Quote}_A(\mathsf{Sig}_{K_A^{-1}}(g^{a_1}))$$
$$B \rightarrow A : g^{b_1} \mid \mathsf{Sig}_{K_B^{-1}}(g^{b_1}) \mid \mathsf{Quote}_B(\mathsf{Sig}_{K_B^{-1}}(g^{b_1}))$$
$$A \rightarrow B : g^{a_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_1 b_1})}(m_1) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_1 b_1}))}(g^{a_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_1 b_1})}(m_1))$$
$$B \rightarrow A : g^{b_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_1})}(m_2) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_2 b_1}))}(g^{b_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_1})}(m_2))$$
$$A \rightarrow B : g^{a_3} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_2})}(m_3) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_2 b_2}))}(g^{a_3} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2 b_2})}(m_3))$$

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 p_{enc} (i.e. $g^{a_1 b_1}$, $g^{a_2 b_1}$, $g^{a_2 b_2}$) are not leaked to $\mathcal{H}$. Next, ProVerif uses this secrey 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.

Benchmark	x86+SGX instructions	BoogiePL statements	Moat proof	Policy Annotations
OTP	188	1774	5.8 sec	4
Notary	147	1222	3.2 sec	2
OTR IM	251	2191	5.6 sec	7
Query	575	4727	61 sec	9

Figure 9: Summary of experimental results. Columns are (1) instructions analyzed by Moat not including crypto library, (2) size of $I(p_{enc-\mathcal{H}})$, (3) proof time, (4) number of secret and declassifed annotations

9. RELATED WORK

Our work relates three somewhat distinct areas in security.

Secure Systems on Trusted Hardware. In recent years, there has been growing interest in building secure systems on top of trusted hardware. Sancus [23] is a security architecture for networked embedded devices that seeks to provide security guarantees without trusting any infrastructural software, only relying on trusted hardware. Intel SGX [17] seeks to provide similar guarantees via extension to the x86 instruction set. There are some recent efforts on using SGX for trusted computation. Haven [7] is a system that exploits Intel SGX for shielded execution of unmodified legacy applications. VC3 [25] uses SGX to run map-reduce computations while protecting data 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 p_{enc}, a binary, and a protocol model.

11. REFERENCES

[1] Available at https://devmoat.github.io.

[2] ARM Security Technology - Building a Secure System using TrustZone Technology. ARM Technical White Paper.

[3] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO '05*, LNCS 4111, pages 364–387, 2005.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87, 2005.

[5] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[6] G. Barthe and L. P. Nieto. Secure information flow for a concurrent language with scheduling. In *Journal of Computer Security*, pages 647–689. IOS Press, 2007.

[7] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[8] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001.

[9] B. Blanchet. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the link between formal and computational models*, Paris, France, June 2005.

[10] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, 2011.

[12] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, Sept. 2010.

[13] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, pages 337–340, 2008.

[14] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488, 2014.

[15] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings 35th Symposium on Principles of Programming Languages.* G. Smith, 2008.

[16] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: end-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 165–181. USENIX Association, 2014.

[17] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade, and J. Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[18] Intel Software Guard Extensions Programming Reference. Available at https://software.intel.com/sites/default/files/329298-001.pdf.

[19] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, USA, 2009.

[20] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.

[21] J. Mclean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1:37–58, 1992.

[22] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 129–142, New York, USA, 1997. ACM.

[23] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22nd USENIX Conference on Security*, pages 479–494, 2013.

[24] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *In Proc. International Symp. on Software Security*, pages 174–191. Springer-Verlag, 2004.

[25] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud. Technical Report MSR-TR-2014-39, February 2014.

[26] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.

[27] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 2010 Conference on Programming Language Design and Implementation*, pages 99–110, 2010.

APPENDIX

A. PROOF OF TYPE SOUNDNESS THEOREM IN MOAT

Theorem: Suppose

1. $[\tau] \vdash p_{enc-\mathcal{H}} \Rightarrow (\psi, A)$
2. $\langle p_{enc-\mathcal{H}-sec}, \sigma\rangle \Downarrow t_1$
3. $\langle p_{enc-\mathcal{H}-sec}, \sigma\rangle \Downarrow t_2$
4. $\forall i. t_1|_{\mathcal{H}}[i] = t_2|_{\mathcal{H}}[i]$

Then $\forall i. t_1|_{\mathsf{obs}}[i] = t_2|_{\mathsf{obs}}[i]$.

Proof:

We prove this by induction on the structure of p_{enc}. Furthermore, instead of $(\forall i.\ t_1|_{\mathsf{obs}}[i] = t_2|_{\mathsf{obs}}[i])$, we use the following two properties 12 and 13 which (in conjunction) imply $(\forall i.\ t_1|_{\mathsf{obs}}[i] = t_2|_{\mathsf{obs}}[i])$. Assuming the 4 conditions in the theorem above, we prove that each typing rule preserves both properties 12 and 13.

$$\begin{aligned} &\forall i, j.\ ((\neg t_1[i].\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg t_2[i].\mathsf{C}_{\mathtt{mem}}[j]) \Rightarrow (t_1[i].\mathtt{mem}[j] = t_2[i].\mathtt{mem}[j])) \\ &\wedge\ \ \forall i, r \in \mathtt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)) \end{aligned} \tag{12}$$

$$\forall i, j.\ \neg\mathtt{enc}(j) \Rightarrow (\neg t_1[i].\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg t_2[i].\mathsf{C}_{\mathtt{mem}}[j]) \tag{13}$$

(scalar).

A scalar assignment performs one transition step, hence traces t_1 and t_2 contain two states each. $t_1 = [\sigma_1, \sigma_1']$, where $(\sigma_1, \sigma_1') \in \mathcal{R}(\mathsf{x}' := e)$. Similarly, $t_2 = [\sigma_2, \sigma_2']$, where $(\sigma_2, \sigma_2') \in \mathcal{R}(\mathsf{x}' := e)$. Scalar assignments only update $\mathtt{regs}$, therefore $\sigma_1.\mathtt{mem} = \sigma_1'.\mathtt{mem}$ and $\sigma_2.\mathtt{mem} = \sigma_2'.\mathtt{mem}$. For the same reason, $\sigma_1.\mathsf{C}_{\mathtt{mem}} = \sigma_1'.\mathsf{C}_{\mathtt{mem}}$ and $\sigma_2.\mathsf{C}_{\mathtt{mem}} = \sigma_2'.\mathsf{C}_{\mathtt{mem}}$. By induction hypothesis, we know $\forall j.\ (\neg\sigma_1.\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg\sigma_2.\mathsf{C}_{\mathtt{mem}}[j]) \Rightarrow (\sigma_1.\mathtt{mem}[j] = \sigma_2.\mathtt{mem}[j])$. Propagating these equalties, we derive $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\mathtt{mem}}[j]) \Rightarrow (\sigma_1'.\mathtt{mem}[j] = \sigma_2'.\mathtt{mem}[j])$. Regarding update to $\mathtt{regs}$, we have two cases:

1. $\vdash e : \bot$ in both σ_1 and σ_2 i.e. $\bigwedge_{v \in Vars(e)} \neg\sigma_1.\mathsf{C}_{\mathsf{v}}\ \wedge\ \bigwedge_{v \in Vars(e)} \neg\sigma_2.\mathsf{C}_{\mathsf{v}}$.
   In this case, property 12 dictates that e evaluates to the same value in states σ_1 and σ_2. Therefore, the updated register x' has the same value in both states σ_1' and σ_2'. We derive $\forall i, r \in \mathtt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)$.
2. $\vdash e : \top$ in σ_1 or σ_2 or both i.e. $\bigvee_{v \in Vars(e)} \sigma_1.\mathsf{C}_{\mathsf{v}}\ \vee\ \bigvee_{v \in Vars(e)} \sigma_2.\mathsf{C}_{\mathsf{v}}$.
   In this case, our type system sets $\mathsf{C}_{\mathsf{x}'}$ to *true* in either σ_1' and σ_2' or both. Therefore, we derive $\forall i, r \in \mathtt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)$.

In both cases, we derive $\forall i, r \in \mathtt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)$. Conjuncting with our earlier derivation on $\mathtt{mem}$, we show that *scalar* preserves property 12.

Now we prove that *scalar* also preserves property 13. By induction hypothesis, we know $\forall j.\ \neg\mathtt{enc}(j) \Rightarrow (\neg\sigma_1.\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg\sigma_2.\mathsf{C}_{\mathtt{mem}}[j])$. Since scalar assignments do not update $\mathtt{mem}$, $\mathsf{C}_{\mathtt{mem}}$ also retains its value. Therefore, we prove $\forall j.\ \neg\mathtt{enc}(j) \Rightarrow (\neg\sigma_1'.\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\mathtt{mem}}[j])$. Property 13 is preserved by *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 $(\sigma_1, \sigma_1') \in \mathcal{R}(\mathsf{x}' := \mathtt{load}(\mathtt{mem}, e))$. Similarly, $t_2 = [\sigma_2, \sigma_2']$, where $(\sigma_2, \sigma_2') \in \mathcal{R}(\mathsf{x}' := \mathtt{load}(\mathtt{mem}, e))$. Here x' is a scalar register variable i.e. $\mathsf{x}' \in \mathtt{regs}$. Since $\mathtt{load}$ can only update $\mathtt{regs}$, we derive $\sigma_1.\mathtt{mem} = \sigma_1'.\mathtt{mem}$ and $\sigma_2.\mathtt{mem} = \sigma_2'.\mathtt{mem}$. For the same reason, $\sigma_1.\mathsf{C}_{\mathtt{mem}} = \sigma_1'.\mathsf{C}_{\mathtt{mem}}$ and $\sigma_2.\mathsf{C}_{\mathtt{mem}} = \sigma_2'.\mathsf{C}_{\mathtt{mem}}$. By induction hypothesis, we know $\forall j.\ (\neg\sigma_1.\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg\sigma_2.\mathsf{C}_{\mathtt{mem}}[j]) \Rightarrow (\sigma_1.\mathtt{mem}[j] = \sigma_2.\mathtt{mem}[j])$. Propagating these equalties, we derive $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\mathtt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\mathtt{mem}}[j]) \Rightarrow (\sigma_1'.\mathtt{mem}[j] = \sigma_2'.\mathtt{mem}[j])$. Regarding update to $\mathtt{regs}$, we have two cases:

1. $\vdash e : \bot$ in both σ_1 and σ_2 i.e. $\bigwedge_{v \in Vars(e)} \neg\sigma_1.\mathsf{C}_{\mathsf{v}}\ \wedge\ \bigwedge_{v \in Vars(e)} \neg\sigma_2.\mathsf{C}_{\mathsf{v}}$.
   In this case, property 12 dictates that e evaluates to the same value in states σ_1 and σ_2 i.e. $\mathtt{load}$ happens from the same address. We have two nested cases:
   (a) $\mathsf{C}_{\mathsf{mem}}[e]$ is $false$ in both σ_1 and σ_2
   Therefore, register x' is updated to the same value in both states σ_1' and σ_2'. We derive $\forall r \in \mathtt{regs}.\ ((\neg\sigma_1'.\mathsf{C}_{\mathsf{r}} \wedge \neg\sigma_2'.\mathsf{C}_{\mathsf{r}}) \Rightarrow (\sigma_1'.r = \sigma_2'.r)$.
   (b) $\mathsf{C}_{\mathsf{mem}}[e]$ is *true* in either σ_1 or σ_2 or both
   We appropriately set $\mathsf{C}_{\mathsf{x}'}$ to *true*, making making $\forall r \in \mathtt{regs}.\ ((\neg\sigma_1'.\mathsf{C}_{\mathsf{r}} \wedge \neg\sigma_2'.\mathsf{C}_{\mathsf{r}}) \Rightarrow (\sigma_1'.r = \sigma_2'.r)$ trivially *true*.
2. $\vdash e : \top$ in σ_1 or σ_2 or both i.e. $\bigvee_{v \in Vars(e)} \sigma_1.\mathsf{C}_{\mathsf{v}}\ \vee\ \bigvee_{v \in Vars(e)} \sigma_2.\mathsf{C}_{\mathsf{v}}$.
   In this case, our type system sets $\mathsf{C}_{\mathsf{x}'}$ to *true* in either σ_1' and σ_2' or both. Therefore, we trivially derive $\forall i, r \in \mathtt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)$.

In all cases, we derive $\forall i, r \in \mathtt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)$. Conjuncting with our earlier derivation on $\mathtt{mem}$, we show that *scalar* preserves property 12.

Now we prove that *load* also preserves property 13. By induction hypothesis, we know $\forall j.\ \neg\texttt{enc}(j) \Rightarrow (\neg\sigma_1.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2.\mathsf{C}_{\texttt{mem}}[j])$. Since load does not update mem, $\mathsf{C}_{\texttt{mem}}$ also retains its value. Therefore, we prove $\forall j.\ \neg\texttt{enc}(j) \Rightarrow (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j])$. 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 $(\sigma_1, \sigma_1') \in \mathcal{R}(\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e))$. Similarly, $t_2 = [\sigma_2, \sigma_2']$, where $(\sigma_2, \sigma_2') \in \mathcal{R}(\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e))$. The *storeH* rule assigns $\sigma_1'.\mathsf{C}_{\texttt{mem}}[\sigma_1.\mathsf{y}]$ and $\sigma_2'.\mathsf{C}_{\texttt{mem}}[\sigma_2.\mathsf{y}]$ to *true*. This allows us to derive $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j])$. Furthermore, $\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e)$ 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 $\texttt{enc}(\mathsf{y})$. Therefore, $\mathsf{C}_{\texttt{mem}}$ retains its value for any location y for which $\neg\texttt{enc}(\mathsf{y})$ holds, allowing us to derive property 13. In other words, if this $\texttt{enc}(\mathsf{y})$ 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 $(\sigma_1, \sigma_1') \in \mathcal{R}(\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e))$. Similarly, $t_2 = [\sigma_2, \sigma_2']$, where $(\sigma_2, \sigma_2') \in \mathcal{R}(\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e))$. Furthermore, note that *storeL* generates $\texttt{assert}\ \neg\texttt{enc}(\mathsf{y}) \rightarrow (\psi_1 \wedge \psi_2)$.

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

1. $\texttt{enc}(\mathsf{y})$
   (a) $\vdash y : \bot$ in both σ_1 and σ_2, and $\vdash e : \bot$ in both σ_1 and σ_2
   i.e. $\bigwedge_{v \in Vars(e) \cup Vars(\mathsf{y})} \neg\sigma_1.\mathsf{C}_{\mathsf{v}}\ \wedge\ \bigwedge_{v \in Vars(e) \cup Vars(\mathsf{y})} \neg\sigma_2.\mathsf{C}_{\mathsf{v}}$.
   By induction hypothesis, e and y evaluate to the same value in both σ_1 and σ_2; hence, the assignment updates mem and $\mathsf{C}_{\texttt{mem}}$ at the same address with the same value in both σ_1 and σ_2. Therefore, we derive $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j])$.
   (b) $\vdash e : \top$ in σ_1 or σ_2 or both, or $\vdash y : \top$ in σ_1 or σ_2 or both
   i.e. $\bigvee_{v \in Vars(e) \cup Vars(\mathsf{y})} \sigma_1.\mathsf{C}_{\mathsf{v}}\ \vee\ \bigvee_{v \in Vars(e) \cup Vars(\mathsf{y})} \sigma_2.\mathsf{C}_{\mathsf{v}}$.
   Since this statement updates $\sigma_1.\texttt{mem}$ at $\sigma_1.\mathsf{y}$ and $\sigma_2.\texttt{mem}$ at $\sigma_2.\mathsf{y}$, we update $\sigma_1.\mathsf{C}_{\texttt{mem}}[\sigma_1.\mathsf{y}]$ or $\sigma_2.\mathsf{C}_{\texttt{mem}}[\sigma_2.\mathsf{y}]$ or both to *true*. Therefore, $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j])$.
2. $\neg\texttt{enc}(\mathsf{y})$
   The *storeL* rule generates an assertion $\neg\texttt{enc}(\mathsf{y}) \rightarrow \bigwedge_{v \in Vars(e) \cup Vars(\mathsf{y})} \neg\mathsf{C}_{\mathsf{v}}$. Since $\neg\texttt{enc}(\mathsf{y})$ holds in this case, this assertion checks that both the address y and value e have type $\bot$. By induction hypothesis, both y and e must evaluate to the same value in σ_1 and σ_2. If this assertion is *valid*, then there does not exist any execution of $\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e)$ that violates $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j])$.

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

Next, we prove that *storeL* also preserves property 13. The *storeL* rule generates $\texttt{assert}\ \neg\texttt{enc}(\mathsf{y}) \rightarrow \bigwedge_{v \in Vars(e) \cup Vars(\mathsf{y})} \neg\mathsf{C}_{\mathsf{v}}$. If this statement writes to $\texttt{mem}_{\neg\mathsf{enc}}$, then the assertion checks that both the address y and value e have type $\bot$. If this assertion is *valid*, then there does not exist any execution of $\texttt{mem}' := \texttt{store}(\texttt{mem}, \mathsf{y}, e)$ that violates $\forall j.\ \neg\texttt{enc}(j) \Rightarrow (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j])$. This is because both ($\sigma_1'.\mathsf{C}_{\texttt{mem}}[\mathsf{y}]$ and $\sigma_2'.\mathsf{C}_{\texttt{mem}}[\mathsf{y}]$) are set to $false$. Hence, we prove that *storeL* also preserves property 13.

(egetkey).

We first try proving property 12. Regarding the update to mem, we have 2 cases:

1. $\neg\sigma_1.\mathsf{C}_{\mathsf{rcx}}$ and $\neg\sigma_2.\mathsf{C}_{\mathsf{rcx}}$:
   Although rcx evaluates to the same value in σ_1 and σ_2 (from induction hypothesis), the *egetkey* rule marks 16 bytes starting at $\sigma_1.\mathsf{rcx}$ as secret. That is, it sets $\sigma_1'.\mathsf{C}_{\mathsf{mem}}[\sigma_1.\mathsf{rcx} \ldots \sigma_1.\mathsf{rcx} + 16]$ and $\sigma_2'.\mathsf{C}_{\mathsf{mem}}[\sigma_2.\mathsf{rcx} \ldots \sigma_2.\mathsf{rcx} + 16]$ to *true*, where $\sigma_1.\mathsf{rcx} = \sigma_2.\mathsf{rcx}$. Therefore, we derive $\forall i, j.\ ((\neg t_1[i].\mathsf{C}_{\texttt{mem}}[j] \wedge \neg t_2[i].\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (t_1[i].\texttt{mem}[j] = t_2[i].\texttt{mem}[j]))$.
2. $\sigma_1.\mathsf{C}_{\mathsf{rcx}}$ or $\sigma_2.\mathsf{C}_{\mathsf{rcx}}$:
   Although rcx may evaluate to different values in σ_1 and σ_2, the *egetkey* rule marks 16 bytes starting at $\sigma_1.\mathsf{rcx}$ and 16 bytes starting at $\sigma_2.\mathsf{rcx}$ as secret. That is, it sets $\sigma_1'.\mathsf{C}_{\mathsf{mem}}[\sigma_1.\mathsf{rcx} \ldots \sigma_1.\mathsf{rcx} + 16]$ and $\sigma_2'.\mathsf{C}_{\mathsf{mem}}[\sigma_2.\mathsf{rcx} \ldots \sigma_2.\mathsf{rcx} + 16]$ to *true*, where two address ranges may not be identical. From induction hypothesis, we still derive $\forall i, j.\ ((\neg t_1[i].\mathsf{C}_{\texttt{mem}}[j] \wedge \neg t_2[i].\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (t_1[i].\texttt{mem}[j] = t_2[i].\texttt{mem}[j]))$.

Since egetkey does not modify regs, we derive $\forall i, r \in \texttt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r))$. Combining with the reasoning on 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 $\texttt{mem}_{\neg\mathsf{enc}}$ i.e. egetkey is guaranteed to write the key to $\texttt{mem}_{\mathsf{enc}}$. This constraint is captured in the axioms defining egetkey in $\texttt{mem}' = \texttt{egetkey}(\texttt{mem}, \mathsf{rbx}, \mathsf{rcx})$. Therefore, property 13 holds trivially — $\mathsf{C}_{\mathsf{mem}}$ is not updated for any location j for which $\neg\texttt{enc}(\mathsf{j})$ 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 $(\sigma_1, \sigma_1') \in \mathcal{R}(\texttt{mem}' := \texttt{ereport}(\texttt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx}))$. Similarly, $t_2 = [\sigma_2, \sigma_2']$, where $(\sigma_2, \sigma_2') \in \mathcal{R}(\texttt{mem}' := \texttt{ereport}(\texttt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx}))$.

The *ereportH* rule assigns $\sigma_1'.\mathsf{C}_{\texttt{mem}}[\sigma_1.\mathsf{rdx} \ldots \sigma_1.\mathsf{rdx} + 432]$ and $\sigma_2'.\mathsf{C}_{\texttt{mem}}[\sigma_2.\mathsf{rdx} \ldots \sigma_2.\mathsf{rdx} + 432]$ to *true*. This allows us to derive $\forall j.\ (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j])$. 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 $\texttt{mem}_{\neg\texttt{enc}}$ i.e. `ereport` is guaranteed to write the key to $\texttt{mem}_{\texttt{enc}}$. This constraint is captured in the axioms defining `ereport` in $\texttt{mem}' = \texttt{ereport}(\texttt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx})$. Therefore, property 13 holds trivially — $\mathsf{C}_{\mathsf{mem}}$ is not updated for any location j for which $\neg\texttt{enc}(\mathsf{j})$ 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. $\neg\sigma_1.\mathsf{C}_{\mathsf{rcx}}$ and $\neg\sigma_2.\mathsf{C}_{\mathsf{rcx}}$ and $\neg\sigma_1.\mathsf{C}_{\mathsf{rdx}}$ and $\neg\sigma_2.\mathsf{C}_{\mathsf{rdx}}$:
   Since rcx and rdx have the same value in σ_1 and σ_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. $\sigma_1.\mathsf{C}_{\mathsf{rcx}}$ or $\sigma_2.\mathsf{C}_{\mathsf{rcx}}$ or $\sigma_1.\mathsf{C}_{\mathsf{rdx}}$ or $\sigma_2.\mathsf{C}_{\mathsf{rdx}}$:
   Since rcx or rdx may evaluate to different values in σ_1 and σ_2, *ereportL* marks 432 bytes starting at $\sigma_1.\mathsf{rdx}$ and 432 bytes starting at $\sigma_2.\mathsf{rdx}$ 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 $\texttt{mem}_{\neg\texttt{enc}}$ i.e. `ereport` is guaranteed to write the key to $\texttt{mem}_{\texttt{enc}}$. This constraint is captured in the axioms defining `ereport` in $\texttt{mem}' = \texttt{ereport}(\texttt{mem}, \mathsf{rbx}, \mathsf{rcx}, \mathsf{rdx})$. Therefore, property 13 holds trivially — $\mathsf{C}_{\mathsf{mem}}$ is not updated for any location j for which $\neg\texttt{enc}(\mathsf{j})$ holds.

(eexit).

`eexit` affects a TCS page in $\texttt{mem}_{\texttt{enc}}$, 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, `eexit` does not affect `regs`. This constraint is captured in the axioms defining `eexit` in $\texttt{mem}' = \texttt{eexit}(\texttt{mem})$. Therefore, we derive $\forall i, r \in \texttt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r))$. The updated enclave page remains at security level $\perp$ 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 $\perp$ implies that the new contents of this page is equivalent in both σ_1' and σ_2'. Therefore, we derive $\forall i, j.\ ((\neg t_1[i].\mathsf{C}_{\texttt{mem}}[j] \wedge \neg t_2[i].\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (t_1[i].\texttt{mem}[j] = t_2[i].\texttt{mem}[j]))$. Combining the two derivations, we derive property 12.

As we mention above, `eexit` only updates an enclave page in $\texttt{mem}_{\texttt{enc}}$ — we prove using our model of SGX that this instruction does not write to $\texttt{mem}_{\neg\texttt{enc}}$. Therefore, $\forall i, j.\ \neg\texttt{enc}(j) \Rightarrow (\neg t_1[i].\mathsf{C}_{\texttt{mem}}[j] \wedge \neg t_2[i].\mathsf{C}_{\texttt{mem}}[j])$ holds trivially, thus deriving property 13.

(ite).

An if-then-else statement is of the form $\mathsf{if}\ (e)\ \{s_t\}\ \mathsf{else}\ \{s_e\}$. We have 2 cases:

1. $[\perp] \vdash \mathsf{if}\ (e)\ \{s_t\}\ \mathsf{else}\ \{s_e\}$
   Hence, $\vdash e : \perp$ in σ_1 and σ_2 i.e. $\bigwedge_{v \in Vars(e)} \neg\sigma_1.\mathsf{C}_{\mathsf{v}}\ \wedge\ \bigwedge_{v \in Vars(e)} \neg\sigma_2.\mathsf{C}_{\mathsf{v}}$. In this case, the inductive hypothesis implies that e evaluates to the same value in states σ_1 and σ_2 — execution follows the same branch in both states. Using the structural induction hypothesis on s_t and s_e, we prove that *(ite)* preserves properties 12 and 13.

2. $[\top] \vdash \mathsf{if}\ (e)\ \{s_t\}\ \mathsf{else}\ \{s_e\}$
   Hence, $\vdash e : \top$ in σ_1 or σ_2 i.e. $\bigvee_{v \in Vars(e)} \sigma_1.\mathsf{C}_{\mathsf{v}}\ \vee\ \bigvee_{v \in Vars(e)} \sigma_2.\mathsf{C}_{\mathsf{v}}$. Since e may evaluate to different values in σ_1 and σ_2, execution may follow different branches in the two states. For each register r modified by evaluating $\mathsf{if}\ (e)\ \{s_t\}\ \mathsf{else}\ \{s_e\}$ in σ_1 and σ_2, the *ite* rule sets C_{r} to *true* in either σ_1' or σ_2'. Therefore, we derive $\forall i, r \in \texttt{regs}.\ ((\neg t_1[i].\mathsf{C}_{\mathsf{r}} \wedge \neg t_2[i].\mathsf{C}_{\mathsf{r}}) \Rightarrow (t_1[i].r = t_2[i].r)$. Similarly, for each address j in `mem` modified by evaluating $\mathsf{if}\ (e)\ \{s_t\}\ \mathsf{else}\ \{s_e\}$ in σ_1 and σ_2, *ite* sets $\mathsf{C}_{\texttt{mem}}[j]$ to *true* in either σ_1' or σ_2'. Therefore, we derive $\forall j.\ ((\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j]))$. Combining the derivations, we prove that property 12 is preserved by *ite*. Since we must typecheck both s_t and s_e in $\top$ context, *ite* generates assertions checking that writes are not made to $\texttt{mem}_{\neg\texttt{enc}}$. 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 $(\sigma_1, \sigma_1') \in \mathcal{R}(s)$ and $(\sigma_1', \sigma_1'') \in \mathcal{R}(s')$. Similarly, $t_2 = [\sigma_2, \sigma_2', \sigma_2'']$, where $(\sigma_2, \sigma_2') \in \mathcal{R}(s)$ and $(\sigma_2', \sigma_2'') \in \mathcal{R}(s')$. From structural induction, we derive the following:

$$\forall j.\ ((\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1'.\texttt{mem}[j] = \sigma_2'.\texttt{mem}[j])) \\ \wedge\ \ \forall r \in \texttt{regs}.\ ((\neg\sigma_1'.\mathsf{C}_{\mathsf{r}} \wedge \neg\sigma_2'.\mathsf{C}_{\mathsf{r}}) \Rightarrow (\sigma_1'.r = \sigma_2'.r))$$

$$\forall j.\ ((\neg\sigma_1''.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2''.\mathsf{C}_{\texttt{mem}}[j]) \Rightarrow (\sigma_1''.\texttt{mem}[j] = \sigma_2''.\texttt{mem}[j])) \\ \wedge\ \ \forall r \in \texttt{regs}.\ ((\neg\sigma_1''.\mathsf{C}_{\mathsf{r}} \wedge \neg\sigma_2''.\mathsf{C}_{\mathsf{r}}) \Rightarrow (\sigma_1''.r = \sigma_2''.r))$$

Similarly, we leverage structural induction to prove the following:.

$$\forall j.\ \neg\texttt{enc}(j) \Rightarrow (\neg\sigma_1'.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2'.\mathsf{C}_{\texttt{mem}}[j]) \tag{14}$$

$$\forall j.\ \neg\texttt{enc}(j) \Rightarrow (\neg\sigma_1''.\mathsf{C}_{\texttt{mem}}[j] \wedge \neg\sigma_2''.\mathsf{C}_{\texttt{mem}}[j]) \tag{15}$$

Therefore, we prove both property 12 and property 13.