Moat: Verifying Confidentiality of Enclave Programs

Rohit Sinha
UC Berkeley
rsinha@berkeley.edu

Sriram Rajamani
Microsoft Research
sriram@microsoft.com

Sanjit A. Seshia
UC Berkeley
sseshia@berkeley.edu

Kapil Vaswani
Microsoft Research
kapilv@microsoft.com

ABSTRACT

Security-critical applications constantly face threats from exploits in lower computing layers such as the operating system, virtual machine monitors, or even attacks from malicious 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) instructions. These primitives enable hardware protection of memory regions containing code and data, and provide a root of trust for measurement, remote attestation, and cryptographic sealing. However, vulnerabilities in the application itself, such as the incorrect use of SGX instructions or memory safety errors, can be exploited to divulge secrets. In this paper, we introduce a new approach to formally model these primitives and formally verify properties of so-called enclave programs that use them. More specifically, we create formal models of relevant aspects of SGX, develop several adversary models, and present a sound verification methodology (based on automated theorem proving and information flow analysis) for proving that an enclave program running on SGX does not contain a vulnerability that causes it to reveal secrets to the adversary. We introduce Moat, a tool which formally verifies confidentiality properties of applications running on SGX. We evaluate Moat on several applications, including a one time password scheme, off-the-record messaging, notary service, and secure query processing.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection — *Information flow controls, Verification*; D.2.4 [**Software Engineering**]: Software/Program Verification — *Formal methods, Model checking*

Keywords

Enclave Programs; Secure Computation; Confidentiality; Formal Verification

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

ACM 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813608 .

1. INTRODUCTION

Building applications that do not leak secrets, i.e., provide confidentiality guarantees, is a non-trivial task. There are at least three kinds of attacks a developer must guard against. The first kind of attack, which we call *protocol attack*, is relevant for distributed applications involving client nodes and cloud-based services, and can arise from 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., the Heartbleed bug [18]). 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 has full control of the CPU, memory, I/O devices, etc. Infrastructure attacks can result in an attacker gaining control of any application's memory and reading secrets at will.

Several mitigation strategies have been proposed for each of these kinds of attacks. In order to guard against protocol attacks, we can use protocol verifiers (e.g., ProVerif [11], CryptoVerif [12]) 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 [28]. 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. [21, 25]), 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 [23, 24] 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. Using these extensions it is possible to write applications with a small trusted computing base, which includes only the enclave code and the SGX processor. All

other layers in the software stack including the operating system and the hypervisor can be excluded from the trusted computing base. However, establishing such a guarantee requires precise understanding of the contract between the hardware and software. In particular, it requires specifying a formal API-level semantics of SGX instructions, an adversary model, and a verification methodology (with tool support) to detect insecure use of SGX instructions. Our work addresses each of these aspects, as we describe below.

Semantics of SGX instructions. SGX provides instructions which enable applications to create enclaves, transfer control to and from enclaves, perform remote attestation, and seal and unseal secrets so that they can persist on the platform. We give a formal semantic model of the interface between the programmer and the implementation of SGX, with an eye towards automatic formal verification of the enclave code's security. Our approach is similar in spirit to previous work on finding API-level exploits [20] using the UCLID modeling and verification system [15].

Adversary model. We consider a powerful adversary who can compromise the infrastructure code (OS, hypervisor, etc.) and perform both passive and active attacks by accessing any non-enclave memory (i.e. memory not protected by SGX), modifying page tables, generating interrupts, controlling I/O interaction, etc. It is non-trivial to reason about enclave behaviors (and potential vulnerabilities) in the presence of such an adversary. As a first step, using our formal ISA-level semantics of the SGX instructions, we prove that (if the application follows certain guidelines during enclave creation) any combination of the above adversarial actions can be simply modeled as an arbitrary update of non-enclave memory. This simplifies our reasoning of enclave execution in the presence of privileged adversaries. In particular, we show that a so-called *havocing adversary* who symbolically modifies all of the non-enclave memory after every instruction of the enclave code, and is able to observe all non-enclave memory, is powerful enough to model all passive and active adversaries. We consider this to be one of the key contributions of this paper. It greatly simplifies the construction of automated verifiers for checking security properties of enclave code, and potentially even programming and reasoning about enclaves.

Verification methodology and tool support. Even though SGX offers protection from infrastructure attacks, the developer must take necessary steps to defend against protocol and application attacks by using SGX instructions correctly, using safe cryptographic protocols, avoiding traditional bugs due to memory safety violations, etc. For instance, the enclave may suffer from exploits like Heartbleed [18] by using vulnerable SSL implementations, and these exploits have been shown to leak secret cryptographic keys from memory. To that end, our next step is to prove that enclaves satisfy confidentiality i.e. there is no execution that leaks a secret to the adversary-visible, non-enclave memory. Proving confidentiality involves tracking the flow of secrets within the application's memory, and proving that the adversary does not observe values that depend on secrets. While past research has produced several type systems that verify information flows (e.g. Jif [28], Volpano et al. [34], Balliu et al. [5]), they make a fundamental assumption that the infrastructure (OS/VMM, etc.) on which the code runs is safe, which is unrealistic due to privileged malware attacks. Furthermore, extending traditional type systems to enclave programs is non-trivial because the analysis must faithfully model the semantics of SGX instructions and infer information flows to individual addresses within enclave memory. Therefore, we develop a static verifier called Moat that analyzes the instruction-level behavior of the enclave binary program. Moat employs a flow- and path-sensitive type checking algorithm (based on automated theorem proving using satisfiability modulo theories solving [8]) for automatically verifying whether an enclave program (in the presence of an active adversary) provides confidentiality guarantees. For ill-typed programs, Moat returns an exploit demonstrating a potential leak of secret to non-enclave memory. By analyzing the binary, we remove the compiler from our trusted computing base, and relax several memory safety assumptions that are common in traditional information flow type systems.

Although we do not focus on protocol attacks in this paper, we briefly describe how one can compose protocol-level analysis (performed by a verifier such as ProVerif [11]) with Moat to achieve end-to-end confidentiality guarantees against infrastructure, application, and protocol attacks.

In summary, the goal of this paper is to explore the contract between the SGX hardware and the enclave developer and provide a methodology and tool support for the programmer to write secure enclaves. We make the following specific contributions:

- We develop the first semantic API model of the SGX platform and its new instruction set, working from publicly-available documentation [24].
- We formally study active and passive adversaries for SGX enclaves, and show that a havocing adversary who observes and havocs non-enclave memory after every instruction in the enclave is both powerful enough to model all such adversaries, and amenable to be used in automated symbolic verification tools.
- We develop Moat, a system for statically verifying confidentiality properties of an enclave program in the face of application and infrastructure attacks.

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 [4] and Sancus [29], 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

2.1 SGX

The SGX instructions allow a user-level *host application* to instantiate a protected execution context, called an enclave, containing code and data. An enclave's memory resides within the untrusted host application's virtual address space, but is protected from accesses by that host application or any privileged software — only the enclave code is allowed to access enclave memory. Furthermore, to protect against certain hardware attacks, the cache lines belonging

to enclave memory are encrypted and integrity protected by the CPU prior to being written to off-chip memory.

The host application creates an enclave using a combination of instructions: `ecreate`, `eadd`, `eextend`, and `einit`. The 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 (and measurement) from non-enclave code. The host application transfers control to the enclave by invoking `eenter`, which targets a programmer defined entry point inside the enclave (via a callgate-like mechanism). The enclave executes 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 in enclave memory, and can resume the enclave in the same state once the OS / VMM handles the event. Although a compromised OS may launch denial of service attacks, we show that an enclave can still guarantee properties such as data confidentiality.

The reader may have observed that before enclave initialization, code and data is open 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 an attestation primitive called `ereport` to defend against this class of attacks. The enclave participates in attestation by invoking `ereport`, which generates a hardware-signed report of the enclave's measurement, and then sending the report to the verifying party. The enclave can also use `ereport` to bind data to its measurement, thereby adding authenticity to that data. The enclave can use `egetkey` to attain a hardware-generated sealing key, and store sealed secrets to untrusted storage.

2.2 Example

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 displayed 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.

The necessary primitives 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. Hoekstra et al. [23] propose the following OTP scheme based on SGX, which we implement (Figure 1) and verify using Moat. In this protocol, a bank OTP server provisions the pre-shared secret to a client, which is running on a potentially infected machine with SGX hardware.

0. The host application on the client sets up an enclave that contains trusted code for the client side of the protocol.
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 an Intel SGX CPU, while the measurement guarantees that the reporting enclave was not tampered during initialization. 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.

A typical application (such as our OTP client) uses enclave code to implement trusted computation such as the cryptographic operations, stores secrets in the enclave heap, and uses non-enclave code (host application, OS, VMM, etc.) for untrusted computation. In fact, SGX prevents the enclave code from invoking any privileged instructions such as system calls, thus 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, the 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 untrusted app, and (4) waits for app to invoke the socket system calls to send the report to the bank server. Over their lifetimes, app and enclave perform several `eenter` and `eexit` while alternating between trusted and untrusted computation. To facilitate the interaction between an enclave and non-enclave code, SGX allows the enclave to access the entire address space of the host application.

While SGX implements the necessary primitives for doing trusted computation, we need a methodology for writing secure enclave programs. Confidentiality requires protecting

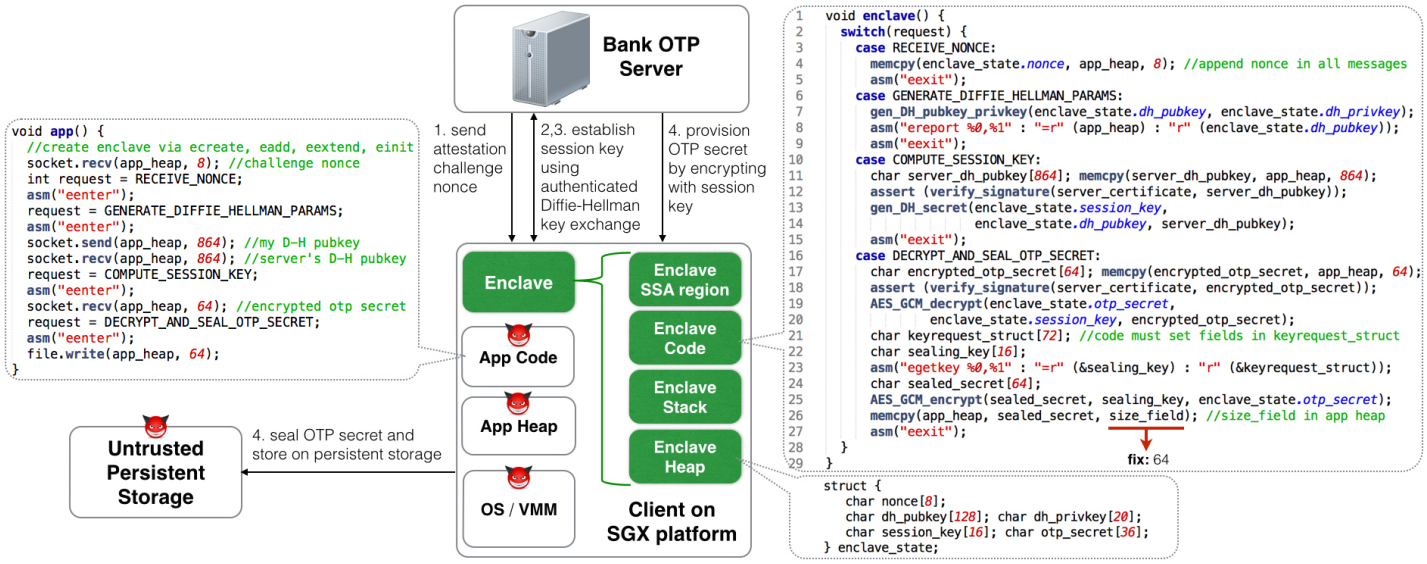

Figure 1: Running OTP Example. The enclave performs trusted cryptographic operations, and the host application performs untrusted tasks such as UI handling and network communications with the OTP server.

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 § 3.1) 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. For example, the enclave code in Figure 1 has such a vulnerability. The enclave code here copies encrypted data from enclave memory to non-enclave memory, but the size of the data copied is determined by a variable size_field, which resides in non-enclave memory. Thus, by manipulating the value of this variable the adversary can trick the enclave code into leaking secrets to non-enclave memory. Avoiding such attacks is non-trivial. In this paper, we present a methodology and tool support for detecting such errors, and proving that enclaves are secure.

The rest of this paper is structured as follows. We provide an overview of our approach in § 3. § 4 describes how Moat constructs a formal model of the enclave program (including its use of x86 and SGX instructions), and also formalizes an active and a passive adversary. § 5 introduces the *havocing adversary*, and shows how it can be used to model the enclave's execution in the presence of an active adversary. The results presented in § 5 allow us soundly verify any safety property of enclave programs. Next, we focus our attention on proving confidentiality by first formalizing it in § 6, and then developing a verification algorithm in § 7. § 8 describes several case studies where we apply Moat.

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*. Our goal is to prove that, even in the presence of such attacks, the enclave does not leak its secrets to the adversary. Moat defends against application and infrastructure attacks. Furthermore, we combine Moat with off-the-shelf protocol verifiers to defend against protocol attacks as well.

3.1 Protecting from Infrastructure Attacks

An infrastructure attack can generate interrupts, modify page tables, modify any non-enclave memory, invoke any x86 and SGX instruction — § 4.3 formalizes the threat model. We mandate that the enclave be created with the following sequence that measures all pages in $\texttt{mem}_{\texttt{enc}}$, which denotes memory reserved for the enclave. We do not find this to be a restriction in practice.

$$\begin{aligned}&\texttt{ecreate}(\mathit{size}(\texttt{mem}_{\texttt{enc}}));\\&\texttt{foreach } \mathit{page} \in \texttt{mem}_{\texttt{enc}} : \{\texttt{eadd}(\mathit{page});\ \texttt{eextend}(\mathit{page})\};\\&\texttt{einit}\end{aligned} \tag{1}$$

If some component of enclave state is not measured, then the adversary may havoc that component of state during initialization without being detected. This precondition on the initialization sequence lets us prove that the SGX hardware provides some very useful guarantees. For instance, we prove Theorem 1 in § 5 which guarantees that an enclave initialized using this sequence is protected from each adversarial operation in our threat model — we formally model each SGX instruction (see § 4.2) to perform this proof. Specifically, we prove a non-interference property [16] that the enclave's execution (i.e. its set of reachable states) is independent of the adversarial operations, with the caveat that the enclave may read non-enclave memory for inputs. However, we do not consider this to be an attack because the enclave must read non-enclave memory for inputs, which are

untrusted by design. The utility of this proof is that all infrastructure attacks can now be modeled by a so-called *havocing adversary* that is only allowed to update non-enclave memory, and this simplifies our reasoning of enclave execution in the presence of privileged adversaries. We call this havocing adversary $\mathcal{H}$ (defined in § 5), and we allow $\mathcal{H}$ to update all addresses in non-enclave memory between any consecutive instructions executed by the enclave. Going forward, we focus primarily on application attacks, and model $\mathcal{H}$'s operations only for the purpose of reads from non-enclave memory.

3.2 Protecting from Application Attacks

In this section, we give an overview of Moat's approach for proving confidentiality properties of enclave code (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 1) program points where secrets values are generated (e.g. after decryption), and 2) memory locations where those secret values are stored. In the OTP example, the *Secrets* include 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's proof methodology on a snippet of OTP enclave code containing lines 22-26 from Figure 1, which is first compiled to x86+SGX Assembly in 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 buffer overrun vulnerability can be exploited by the adversary, causing the enclave to leak secrets from its stack.

```
egetkey                    mem := egetkey(mem, ebx, ecx);
movl $0x8080AC,0x8(%esp)   mem := store(mem,add(esp,8),8080AC);
lea -0x6e0(%ebp),%eax      eax := sub(ebp, 6e0);
mov %eax,0x4(%esp)         mem := store(mem,add(esp,4),eax);
lea -0x720(%ebp),%eax      eax := sub(ebp, 720);
mov %eax,(%esp)            mem := store(mem, esp, eax);
call <AES_GCM_encrypt>     mem := AES_GCM_encrypt(mem, esp);
mov 0x700048,%eax          eax := load(mem, 700048);
movl %eax,0x8(%esp)        mem := store(mem, add(esp,8), eax);
lea -0x720(%ebp),%eax      eax := sub(ebp, 720);
mov %eax,0x4(%esp)         mem := store(mem, add(esp,4), eax);
movl $0x701000,(%esp)      mem := store(mem, esp, 701000);
call <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 imperative 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} uses BAP to model x86 [14] precisely, including updates to CPU flags. For brevity, Figure 2 omits all updates to flags as they are irrelevant to this code snippet. For reasons explained later, Moat does not model the implementation of cryptographic routines (such as AES_GCM_encrypt in Figure 2). It replaces all calls to the cryptographic library with their specifications. In the case of AES_GCM_encrypt, the specification ensures memory safety: only the output ciphertext buffer and memory allocated to the library can be modified by this call.

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 3.1 introduces an active adversary $\mathcal{H}$ (formalized in 5), which can perform the operation "havoc $\texttt{mem}_{\neg\texttt{epc}}$" *once* between consecutive instructions along any execution of p_{enc}. Here, $\texttt{mem}_{\texttt{epc}}$ denotes memory reserved by the SGX processor for enclave use, and $\texttt{mem}_{\neg\texttt{epc}}$ is all of untrusted, non-enclave memory; havoc $\texttt{mem}_{\neg\texttt{epc}}$ updates each address in $\texttt{mem}_{\neg\texttt{epc}}$ 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 $\texttt{mem}_{\neg\texttt{epc}}$, and then resume p_{enc}. We model the effect of $\mathcal{H}$ on p_{enc}'s behavior by instrumenting havoc $\texttt{mem}_{\neg\texttt{epc}}$ in p_{enc} (see Figure 3). The instrumented program is called $p_{enc-\mathcal{H}}$.

```
1  havoc mem¬epc; mem := egetkey(mem, ebx, ecx);
2  havoc mem¬epc; mem := store(mem, add(esp,8), 8080AC);
3  havoc mem¬epc; eax := sub(ebp, 6e0);
4  havoc mem¬epc; mem := store(mem, add(esp,4), eax);
5  havoc mem¬epc; eax := sub(ebp, 720);
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, 700048);
9  havoc mem¬epc; mem := store(mem, add(esp,8), eax);
10 havoc mem¬epc; eax := sub(ebp, 720);
11 havoc mem¬epc; mem := store(mem, add(esp,4), eax);
12 havoc mem¬epc; mem := store(mem, esp, 701000);
13 havoc mem¬epc; mem := memcpy(mem, esp);
```

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

As mentioned before, the OTP enclave implementation is vulnerable. The size argument to memcpy (line 26 in Figure 1) is a field within a data structure in non-enclave memory. This vulnerability manifests as a **load** (line 8 of Figure 3), which reads a value from non-enclave memory and passes that value as the size argument to memcpy. To perform the exploit, $\mathcal{H}$ uses havoc $\texttt{mem}_{\neg\texttt{epc}}$ (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 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. In other words, $\mathcal{H}$'s observation of p_{enc}'s execution is independent of *Secrets*. Note that we omit side channels from $\mathcal{H}$'s observation in this work.

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. Moat tracks both implicit and explicit information flows [30]. 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 independent of *Secrets* (details in § 7). For instance, $\mathsf{C}_{\mathsf{mem}}[\mathsf{esp}]$ in line 13 is assigned to $\mathsf{C}_{\mathsf{eax}}$ because a secret in the eax register makes the written memory location also 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 $\mathtt{mem}_{\neg\mathtt{enc}}$, where $\mathtt{mem}_{\mathtt{enc}}$ is a subset of $\mathtt{mem}_{\mathtt{epc}}$ that is reserved for use by $p_{\mathtt{enc}}$, and $\mathtt{mem}_{\neg\mathtt{enc}}$ is either non-enclave memory or memory used by other enclaves. enc(i) is *true* if i is an address in $\mathtt{mem}_{\mathtt{enc}}$. For each `store` instruction, the type system instruments an `assert` checking that a secret value is not written to $\mathtt{mem}_{\neg\mathtt{enc}}$ (with special treatment of memcpy for efficiency). 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 [6], which uses SMT solving to explore all executions (i.e. all reachable states) and verify that the assertions are valid along all executions. The assertion in line 30 is invalid because $\mathsf{C}_{\mathsf{mem}}$ is *true* for memory locations that hold the sealing_key. Our type system rejects this enclave program. A fix to the OTP implementation is to replace size_field with the correct size, which is 64 bytes. Although memory safety vulnerabilities can be found using simpler static analysis, Moat can identify several classes of vulnerabilities using these typing assertions. The analysis in Moat is sound (if p_{enc} terminates) i.e. Moat finds any vulnerable execution of p_{enc} that leaks a secret to $\mathtt{mem}_{\neg\mathtt{enc}}$.

```
1  assert ¬C_ecx; C^old_mem := C_mem; havoc C_mem;
2  assume ∀i. (ecx ≤ i < ecx + 16) → C_mem[i] ↔ true;
3  assume ∀i. ¬(ecx ≤ i < ecx + 16) → C_mem[i] ↔ C^old_mem[i];
4  havoc mem_¬epc; mem := egetkey(mem, ebx, ecx);
5  assert ¬C_esp; C_mem[add(esp, 8)] := false;
6  havoc mem_¬epc; mem := store(mem, add(esp,8), 8080AC);
7  C_eax := C_ebp;
8  havoc mem_¬epc; eax := sub(ebp, 6e0);
9  assert ¬C_esp ∧ (¬enc(add(esp, 4)) → ¬C_eax); C_mem[add(esp, 4)] := C_eax;
10 havoc mem_¬epc; mem := store(mem, add(esp,4), eax);
11 C_eax := C_ebp;
12 havoc mem_¬epc; eax := sub(ebp, 720);
13 assert ¬C_esp ∧ (¬enc(esp) → ¬C_eax); C_mem[esp] := C_eax;
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[700048];
18 havoc mem_¬epc; eax := load(mem, 700048);
19 assert ¬C_esp ∧ (¬enc(add(esp, 8)) → ¬C_eax); C_mem[add(esp, 8)] := C_eax;
20 havoc mem_¬epc; mem := store(mem, add(esp,8), eax);
21 C_eax := C_ebp;
22 havoc mem_¬epc; eax := sub(ebp, 720);
23 assert ¬C_esp ∧ (¬enc(add(esp, 4)) → ¬C_eax); C_mem[add(esp, 4)] := C_eax;
24 havoc mem_¬epc; mem := store(mem, add(esp,4), eax);
25 assert ¬C_esp; C_mem[esp] := false;
26 havoc mem_¬epc; mem := store(mem, esp, 7001000);
27 C_mem := C_memcpy(C_mem, esp);
28 arg1 := load(mem, esp); arg3 := load(mem, add(esp, 8));
29 havoc mem_¬epc; mem := memcpy(mem, esp);
30 assert ∀i. ((arg1 ≤ i < add(arg1, arg3)) ∧ ¬enc(i)) → ¬C_mem[i];
```

Figure 4: $p_{enc-\mathcal{H}}$ instrumented with typing assertions

Declassification.

In the previous section, we 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, the verification would need to reason about the authentication and attestation scheme used in that Diffie-Hellman exchange in order to derive that the key can be safely used for encryption. Protocol verifiers (e.g. ProVerif [11], CryptoVerif [12]) excel at this form of reasoning. Therefore, 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 plain-text or encryption key is secret. This conservative axiomatization is very unnecessary because a secret encrypted with a key (that is 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 intentional information leak of the program, which may be proven to be a safe information leak using other proof techniques. In our experiments, we safely eliminate declassified outputs from information leakage checking if the protocol verifier has already proven them to be safe outputs.

To collect the *Declassified* annotations, we manually model the cryptographic protocol to verify using an off-the-shelf protocol verifier. The choice of protocol verifier is orthogonal to our work. 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 of protocol-level secrets. 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 $\mathtt{mem}_{\neg\mathtt{epc}}$.

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 of Figure 1 is safe because its axiomatization of aes_gcm_encrypt is imprecise. We proceed by first proving in Moat that the sealing_key (obtained using `egetkey`) 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 30 of Figure 4 is valid.

This illustrates that protocol verification not only provides *Declassified* annotations, but also specifies which values must be kept secret by the enclave to ensure that the protocol is safe. The combination of *Secrets* and *Declassified* annotations is called a policy, and this policy forms an input to Moat in addition to the enclave program.

3.3 Assumptions and Limitations

Our work has the following fundamental limitations:

- The Intel SGX hardware is in our trusted computing base. Specifically, we assume that all x86 + SGX instructions fulfill the ISA-defined semantics. This eliminates a class of attacks such as physical tampering of the CPU and supply chain attacks.
- To make static analysis feasible, Moat assumes that the enclave code cannot be modified at runtime (enforced using page permissions), and is statically linked.
- We do not consider attacks from observing side channels such as memory access patterns, timing, etc.
- Although SGX allows an enclave to have multiple CPU threads, we only consider single-threaded enclaves for simplicity.

The current implementation of Moat makes the following additional assumptions:

- Moat's implementation uses the Boogie [6] program verifier, Z3 [17] SMT solver, and BAP [14] for modeling x86 instructions. All these dependencies are in our trusted computing base.
- We use trusted implementation of cryptographic routines (cryptopp library [1]) to develop our benchmarks. Since Moat does not model their implementation, they are in our trusted computing base.
- Moat assumes that the enclave program has control flow integrity. Moat does not find vulnerabilities that exploit the control flow behavior (such as ROP attacks). This assumption is not fundamental, and can be removed using modern runtime defenses (e.g. [3]).
- We assume that the enclave code cannot cause exceptions, apart from page fault exceptions which are handled seamlessly by the OS/VMM. In other words, we terminate the enclave in the event of all other exceptions (such as divide by 0).
- Moat assumes that the enclave code does not read (via `load` instruction) from static save area (SSA). Note that this assumption does not prevent the untrusted code from invoking `eresume` (which is necessary for resuming from asynchronous exits). We have not yet found this to be a limiting assumption in our benchmarks.

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 [14] 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 user-mode 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 [6] 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*, which include main memory `mem`, ISA-visible CPU registers `regs`, etc. 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 [7].

Enclaves have an entrypoint which is configured at compile time and enforced at runtime by a callgate-like mechanism. Therefore, Moat makes BAP disassemble instructions in the code region starting from the enclave entrypoint. 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. We omit further details on the translation from the microarchitectural instructions to p_{enc} (in the language presented in Figure 5) because it is mostly syntactic and standard — lack of indirect control flow transfers (due to inlining) makes control flow reconstruction simpler. Our p_{enc} model is sound under the following assumptions: (1) control flow integrity, (2) code pages are not modified (which is enforced using page permissions), and (3) the trusted cryptographic library implementation is memory safe (i.e. it does not modify any memory outside of the allocated result buffer or memory allocated to the library).

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 [7] 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 [17]. In the case of assertion failures, the SMT solver also constructs a counter-example execution of p_{enc} demonstrating the assertion failure. In § 7, we show how Moat uses assertions and verification-condition generation to prove confidentiality properties of p_{enc}.

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

Figure 5: Syntax of programs.

4.2 Formal Model of x86 and SGX instructions

While formal models of x86 instructions using BoogiePL has been done before (see for instance [35]), 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 ($\texttt{mem}_{\texttt{epc}}$), and unprotected memory ($\texttt{mem}_{\neg\texttt{epc}}$). For any physical address `a`, `epc(a)` is true iff `a` is an address in $\texttt{mem}_{\texttt{epc}}$. Furthermore, $\texttt{mem}_{\texttt{enc}}$ is a subset of $\texttt{mem}_{\texttt{epc}}$ that is reserved for use by $p_{\texttt{enc}}$ — $\texttt{mem}_{\texttt{enc}}$ 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` is within $\texttt{mem}_{\texttt{enc}}$. The `epcm` is a finite sized array of hardware-managed structures, where each structure stores security critical metadata about a page in $\texttt{mem}_{\texttt{epc}}$. $\texttt{epcm}_{\texttt{enc}}$ is a subset of `epcm` that stores metadata about each page in $\texttt{mem}_{\texttt{enc}}$ — other `epcm` structures are either free or in use by other enclaves. `regs` is the set of ISA-visible CPU registers such as `eax`, `esp`, 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 `x := X[y]` and writes `X[y] := x`, which are interpreted by the Theory of Arrays [8]. The boolean variable `ea` denotes whether this access is made by enclave code to $\texttt{mem}_{\texttt{enc}}$. 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 $p_{\texttt{enc}}$ 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 [26] 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 [2].

```
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, ebx:bv32)
19 {
20  var mem' := mem; var regs' := regs;
21  regs'[rip] := 0bv32 ++ ebx;
22  regs'[CR_ENCLAVE_MODE] := false;
23  mem'[CR_TCS_PA] := 0x00;
24  return (mem',regs');
25 }
```

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 the 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 $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathtt{epc}}$): While the CPU protects the `epc` region, a privileged software adversary can write to any memory location in $\mathtt{mem}_{\neg\mathtt{epc}}$ region. $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathtt{epc}}$ is encoded in BoogiePL as:

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

   where $\mathtt{mem}_{new}$ 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 $\mathtt{mem}_{\neg\mathtt{epc}}$, $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathtt{epc}}$ models havoc on page tables.

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

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

   where each register (e.g. $\mathtt{eax} \in \mathtt{regs}$) 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`, and `eresume` 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-length 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. Furthermore, the active adversary is quite powerful in this model. It may control the entire software stack, from the host application upto the OS/VMM layers, thereby modeling attacks from malicious administrators and privileged malware. The adversary controls all hardware peripherals, and may insert, delete, modify and replay all communication with the external world — these attacks can be modeled using the above 5 transitions. Side-channels are out of scope, where typical side channels may include memory access patterns, timing leaks, and I/O traffic patterns.

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* $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathtt{epc}}$, $\mathsf{havoc}\ \mathtt{regs}$, `interrupt`, *or* `call sgx`, *thereby modifying a component $\sigma|_{\mathcal{G}}$ of machine state σ. Following each microarchitectural instruction in p_{enc}, $\mathcal{G}$ observes a projection $\sigma|_{\mathsf{obs}}$ of machine state σ. Here, $\sigma|_{\mathsf{obs}} \doteq (\sigma.\mathtt{mem}_{\neg\mathtt{epc}})$, and $\sigma|_{\mathcal{G}} \doteq (\sigma.\mathtt{mem}_{\neg\mathtt{enc}}, \sigma.\mathtt{regs}, \sigma.\mathtt{epcm}_{\neg\mathtt{enc}})$.*

Definition 2. **Passive Adversary $\mathcal{P}$.** *The passive adversary $\mathcal{P}$ observes a projection $\sigma|_{\mathsf{obs}}$ of machine state σ after each microarchitectural instruction in p_{enc}. Here, $\sigma|_{\mathsf{obs}} \doteq (\sigma.\mathtt{mem}_{\neg\mathtt{epc}})$ includes the non-enclave memory. $\mathcal{P}$ does not modify any state.*

Note that we omit $\sigma.\mathtt{regs}$ from $\sigma|_{\mathsf{obs}}$ because they cannot be accessed by the adversary while the CPU operates in enclave mode — asynchronous exits clear their values, while `eexit` removes the CPU from enclave mode. Enclave execution may result in exceptions (such as divide by 0 and page fault exception) or faults (such as general protection fault), in which case the exception codes are conveyed to the adversarial OS. We omit exception codes from $\sigma|_{\mathsf{obs}}$ for verification ease, and terminate the enclave (at runtime) on an exception, with the caveat of page fault exceptions which are allowed. Since $\mathcal{G}$ can havoc page tables, it can cause page fault exceptions at runtime. However, a page fault only reveals memory access patterns (at the page granularity), which we consider to be a side-channel observation.

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 $\mathtt{x} := e$, user-mode SGX instruction (`ereport`, `egetkey`, or `eexit`), or a conditional statement. Since p_{enc} is made to be loop-free (by sound loop unrolling), each statement s_i has a distinct label l_i that relates to the program counter. We assume that each microarchitectural instruction (not the x86 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 (where k is unbounded) 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 $\texttt{mem}_{\neg\texttt{epc}}$, 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

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

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 [16]) property of p_{enc}. While attempting to prove this property in the Boogie verifier [6], we quite expectedly discovered counter-examples that illustrate the different ways in which $\mathcal{G}$ affects p_{enc}'s execution:

1. p_{enc} invokes `load`(`mem`, a), where a is an address in $\texttt{mem}_{\neg\texttt{epc}}$. $\mathcal{G}$ havocs $\texttt{mem}_{\neg\texttt{epc}}$ and p_{enc} reads $\texttt{mem}_{\neg\texttt{epc}}$ for inputs, so this counter-example is not surprising.

2. p_{enc} invokes `load`(`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 $\texttt{mem}_{\neg\texttt{epc}}$ 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 limiting in our case studies. Note that this restriction does not prevent the use of `eresume` instruction, which causes the CPU to access the SSA pages directly. However, the former constraint (not reading $\texttt{mem}_{\neg\texttt{epc}}$) is too restrictive in practice because p_{enc} must read $\texttt{mem}_{\neg\texttt{epc}}$ 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 $\texttt{mem}_{\neg\texttt{epc}}$ *operation, thereby modifying a component $\sigma|_{\mathcal{H}}$ of machine state σ. Following each microarchitectural instruction in p_{enc}, $\mathcal{H}$ observes a projection $\sigma|_{\mathsf{obs}}$ of machine state σ. Here, $\sigma|_{\mathsf{obs}} \doteq (\sigma.\texttt{mem}_{\neg\texttt{epc}})$, and $\sigma|_{\mathcal{H}} \doteq (\sigma.\texttt{mem}_{\neg\texttt{epc}})$.*

Composing enclave p_{enc} with active adversary $\mathcal{H}$.

To construct $p_{enc-\mathcal{H}}$, we transform each p_{enc} statement $l:\ s$ to:

$$\text{havoc } \texttt{mem}_{\neg\texttt{epc}};\ l:\ s \tag{4}$$

Figure 3 shows a sample transformation from p_{enc} to $p_{enc-\mathcal{H}}$. Similar to our previous exercise 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 [16]. We prove theorem 1 with the use of lemma 1 and lemma 2, described next.

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}. Let us further instrument havoc $\texttt{mem}_{\neg\texttt{epc}}$ after each $adv_i \in \{adv_1;\ldots;adv_k\}$ — this is sound because a havoc on $\texttt{mem}_{\neg\texttt{epc}}$ does not restrict the allowed values of $\texttt{mem}_{\neg\texttt{epc}}$. The resulting instrumentation for each statement $l:\ s$ is:

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

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 $\texttt{mem}_{\neg\texttt{epc}}$.

In order to define lemma 1, we introduce the following transformation on each statement $l : s$ of p_{enc}:

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

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,*

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

Proof: The intuition is that (if the enclave makes progress) the other adversarial transitions do not affect p_{enc} in any way that is not already simulated by $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathsf{epc}}$. For example, an interrupt causes the enclave to resume in the state prior to the interrupt. In addition, the CPU detects modifications to the page tables that affect the enclave pages, and prevents the enclave from progressing. Other enclaves on the machine may affect execution, but only by sending messages via non-enclave memory, which is simulated by $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathsf{epc}}$. We prove lemma 1 by induction as follows, and also machine-check the proof in Boogie [6] (proof available at [2]). The inductive proof makes use of our modeling of SGX instructions, and is setup as follows. This property in Lemma 1 is a predicate over a pair of traces, making it a 2-safety hyperproperty [16]. 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 the property 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 `einit` in the prescribed initialization sequence in (1). Property 7 is the base case and property 8 is the inductive step in the proof by induction of lemma 1.

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

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

where

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

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

Proof: Lemma 2 requires a straightforward proof as it follows naturally from the semantics of havoc.

Combining lemma 1 and lemma 2, we prove that the effect of $adv_1; \mathsf{havoc}\ \mathtt{mem}_{\neg\mathsf{epc}}; \ldots; adv_k; \mathsf{havoc}\ \mathtt{mem}_{\neg\mathsf{epc}}$ (or $adv_1; adv_2; \ldots; adv_n$) on enclave's execution can be simulated by $\mathsf{havoc}\ \mathtt{mem}_{\neg\mathsf{epc}}$. 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 standard verification techniques to prove safety (or k-safety) properties of enclave programs, and (2) $\mathcal{H}$ gives a convenient mental model of an active adversary's effects on enclave execution. While we focus on proving confidentiality from hereon, we note that theorem 1 is valuable for soundly proving any safety property of enclave programs.

6. FORMALIZING CONFIDENTIALITY

Moat's definition of confidentiality is inspired by standard non-interference definition [27], 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 $\mathtt{mem}_{\neg\mathsf{epc}}$, which it must write in order to return outputs or send messages to remote parties. To that end, we weaken this definition to allow for writes to $\mathtt{mem}_{\neg\mathsf{epc}}$, 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 symbolic 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.*

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

The havoc on *Secrets* cause the secret variables to take potentially differing symbolic values in t_1 and t_2. Property (10) requires $t_1|_{\mathsf{obs}}$ and $t_2|_{\mathsf{obs}}$ to be equivalent, which is achieved only if the enclave does not leak secrets 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). This is a false alarm if the encryption uses a secret key to produce the ciphertext. To remove such false alarms, we take the standard approach of extending the policy with $Declassified = \{(l, v) \mid l \in L, v \in Vars\}$, where a tuple (l, v) denotes that variable v at location l contains a declassified value. In practice, since outputs typically occupy several bytes in memory, v is a range of symbolic addresses in the enclave heap. We can safely eliminate declassified outputs from information leaks as the protocol verifier has already proven them to be safe outputs (see Section 3.2). 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.

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

7. PROVING CONFIDENTIALITY

Moat automatically checks if $p_{enc-\mathcal{H}}$ satisfies confidentiality (property 11). 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. Hence, 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 in many type-based systems [34, 28], the typing rules prevent programs that have 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, mem := store(mem,a,d) is ill-typed if d's value depends on a secret and enc(a) 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, if (d == 42) {mem := store(mem, a, 1)} else {skip} is ill-typed if d's value depends on a secret and enc(a) is false. In both examples above, $\mathcal{H}$ learns the secret value d by reading mem at address a. 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 bind upto 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 $\bot$ (public). At each program location, 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 program location because variables (especially `regs`) may alternate between holding secret and public values. As explained later in this section, Moat uses the security types in order to decide whether instructions in $p_{enc-\mathcal{H}}$ have implicit or explicit leaks. $p_{enc-\mathcal{H}}$ has $Secrets = \{(l, v)\}$ and $Declassified = \{(l, v)\}$ annotations. 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, at each program location — flow sensitivity and type inference are key requirements of type checking machine code.

Moat's type inference algorithm computes first-order 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. That is, an expression may have type $\bot$ 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 first-order (SMT) formula and $\mathcal{F}$ is a set of first-order (SMT) formulae. Each satisfiable interpretation of ψ corresponds to a feasible execution of s. $\mathcal{F}$ contains a SMT formula for each instruction in s, such that the formula is valid iff that instruction 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 $\bot$ 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, and hence 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 scalar register v, $\mathsf{C_v}$ is a boolean; for an array variable v, $\mathsf{C_v}$ (e.g. $\mathsf{C_{mem}}$) is an array and $\mathsf{C_v}$[i] denotes the security type for each location i. *exp1* rule allows inferring the type of any expression e as $\top$. *exp2* rule allows inferring an expression type e as $\bot$ if we derive $\mathsf{C_v}$ to be false for all variables v in the expression e. *storeL* rule marks the memory location as secret if the stored data is secret. In case of secret data, we assert that the updated location is within $\mathtt{mem}_{\neg\mathtt{enc}}$; we also assert that the address is public to prevent implicit leaks. Since *storeH* rule types `store` instructions as $\top$, it unconditionally marks the memory location as secret. This is necessary because the `store` may execute in a context where a secret is implicitly known through the guard of a conditional statement. *load* marks the updated register as secret if the memory location contains a secret value. *ereportL* rule types the updated memory locations as per SGX semantics. `ereport` takes 64 bytes of data (that the programmer intends to bind to the measurement) at address in ecx, and copies them to memory starting at address edx + 320; the rest of the report has public data such as the MAC, measurement, etc. Hence, $\mathsf{C_{mem}}$ retains the secrecy level for the 64 bytes of data, and assumes $false$ for the public data. Similar to *storeH*, *ereportH* unconditionally marks all 432 bytes of the report as secret. `egetkey` stores 16 bytes of the sealing key at address ecx, hence the *egetkey* rule marks those 16 bytes in $\mathsf{C_{mem}}$ as secret. Notice that we do not assert that `ereport` and `egetkey` writes to enclave memory since this is enforced by SGX. `eexit` jumps to the host application without clearing `regs`. Hence, the *eexit* rule asserts that those `regs` hold public values. We prove the following type soundness theorem (machine-checked proof available at [2]).

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 $\bot$) and all formulae in $\mathcal{F}$ are valid, $p_{enc-\mathcal{H}}$ satisfies property 11.*

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

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

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

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

$$\frac{\vdash e_a : \bot \Rightarrow \psi_a}{\begin{array}{l}[\tau] \vdash \mathsf{x}' := \mathtt{load}(\mathtt{mem}, e_a) \Rightarrow \\ \langle (\mathsf{x}' = \mathtt{load}(\mathtt{mem}, e_a)) \wedge (\mathsf{C}_{\mathsf{x}'} \leftrightarrow \mathsf{C}_{\mathtt{mem}}[e_a]), \{\psi_a\} \rangle\end{array}}(load)$$

$$\frac{\vdash e_a : \bot \Rightarrow \psi_a}{\begin{array}{l}[\top] \vdash \mathtt{mem}' := \mathtt{store}(\mathtt{mem}, e_a, e_d) \Rightarrow \\ \langle \mathtt{mem}' = \mathtt{store}(\mathtt{mem}, e_a, e_d) \ \wedge \mathsf{C}_{\mathtt{mem}'} = \mathsf{C}_{\mathtt{mem}}[e_a := true], \\ \{\psi_a \wedge \mathsf{enc}(\mathsf{e_a})\} \rangle\end{array}}(storeH)$$

$$\frac{\vdash e_a : \bot \Rightarrow \psi_a \qquad \vdash e_d : \bot \Rightarrow \psi_d}{\begin{array}{l}[\bot] \vdash \mathtt{mem}' := \mathtt{store}(\mathtt{mem}, e_a, e_d) \Rightarrow \\ \langle \mathtt{mem}' = \mathtt{store}(\mathtt{mem}, e_a, e_d) \wedge \mathsf{C}_{\mathtt{mem}'} = \mathsf{C}_{\mathtt{mem}}[e_a := \psi_d], \\ \{\psi_a \wedge (\neg \mathsf{enc}(e_a) \rightarrow \psi_d)\} \rangle\end{array}}(storeL)$$

$$\frac{}{\begin{array}{l}[\bot] \vdash \mathtt{mem}' := \mathtt{ereport}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \Rightarrow \\ \langle \mathtt{mem}' = \mathtt{ereport}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \ \wedge \\ \forall i.\ (\mathsf{edx} \le i < \mathsf{edx} + 320) \rightarrow \mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow false \\ \wedge\ (\mathsf{edx} + 320 \le i < \mathsf{edx} + 384) \rightarrow \\ \qquad \mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow \mathsf{C}_{\mathtt{mem}}[\mathsf{ecx} + i - \mathsf{edx} - 320] \\ \wedge\ (\mathsf{edx} + 384 \le i < \mathsf{edx} + 432) \rightarrow \mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow false \\ \wedge\ \neg(\mathsf{edx} \le i < \mathsf{edx} + 432) \rightarrow \mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow \mathsf{C}_{\mathtt{mem}}[i], \\ \{\neg \mathsf{C}_{\mathsf{edx}}\} \rangle\end{array}}(ereportL)$$

$$\frac{}{\begin{array}{l}[\top] \vdash \mathtt{mem}' := \mathtt{ereport}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \Rightarrow \\ \langle \mathtt{mem}' = \mathtt{ereport}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}, \mathsf{edx}) \ \wedge \\ \forall i.\ (\mathsf{edx} \le i < \mathsf{edx} + 432) \rightarrow (\mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow true) \\ \wedge\ \neg(\mathsf{edx} \le i < \mathsf{edx} + 432) \rightarrow (\mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow \mathsf{C}_{\mathtt{mem}}[i]), \\ \{\neg \mathsf{C}_{\mathsf{edx}}\} \rangle\end{array}}(ereportH)$$

$$\frac{}{\begin{array}{l}[\tau] \vdash \mathtt{mem}' := \mathtt{egetkey}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}) \Rightarrow \\ \langle \mathtt{mem}' = \mathtt{egetkey}(\mathtt{mem}, \mathsf{ebx}, \mathsf{ecx}) \ \wedge \\ \forall i.\ (\mathsf{ecx} \le i < \mathsf{ecx} + 16) \rightarrow \mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow true \\ \wedge\ \neg(\mathsf{ecx} \le i < \mathsf{ecx} + 16) \rightarrow \mathsf{C}_{\mathtt{mem}'}[i] \leftrightarrow \mathsf{C}_{\mathtt{mem}}[i], \\ \{\neg \mathsf{C}_{\mathsf{ecx}}\} \rangle\end{array}}(egetkey)$$

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

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

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

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

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 [6] program verifier, which receives $I(p_{enc-\mathcal{H}})$ and generates verification conditions in the SMT format. Boogie uses the Z3 [17] 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.

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

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

In summary, Moat's type system is inspired by the type-based approach for information flow checking by Volpano et al. [34]. The core modifications 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 and path-sensitive, and performs type inference. A program is well-typed if the typing assertions are valid in all feasible executions. We ensure soundness by using a sound program verifier to explore all feasible executions of the instrumented $p_{enc-\mathcal{H}}$.

- Our type system includes rules for 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 the Boogie verifier to prove validity of all assertions in $I(p_{enc-\mathcal{H}})$ (modulo declassifications from the protocol verification step).

Optimizations: Our primary objective was to build a sound verifier for proving confidentiality in the presence of an active adversary. However, due to certain scalability challenges, we implement the following optimizations. First, we only introduce havoc $\texttt{mem}_{\neg\texttt{epc}}$ prior to `load` instructions because only a `load` can be used to read $\texttt{mem}_{\neg\texttt{epc}}$. Furthermore, we axiomatize specific library calls such as memcpy and memset, because their loopy implementations incur significant verification overhead.

We now describe some case studies which we verified using Moat and ProVerif in tandem, 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 private key K_{SGX}^{-1}) 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. Recall that although the enclave code contains calls to these cryptographic primitives, Moat abstracts them as uninterpreted functions with basic memory safety axioms. We use cryptopp [1] in our case studies, and we do not verify its implementation.

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 a network 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.

Notary Service.

We implement a notary service introduced by [21] 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 confidentiality.

End-to-End Encrypted Instant Messaging.

We implement the off-the-record messaging protocol [13], 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 ephemeral Diffie-Hellman keys, which encrypt and integrity-protect the messages between A and B. We only present a synchronous form of communication here for simplicity.

$$A \rightarrow B : g^{a_1} \mid \mathsf{Sig}_{K_A^{-1}}(g^{a_1}) \mid \mathsf{Quote}_A(\mathsf{Sig}_{K_A^{-1}}(g^{a_1}))$$
$$B \rightarrow A : g^{b_1} \mid \mathsf{Sig}_{K_B^{-1}}(g^{b_1}) \mid \mathsf{Quote}_B(\mathsf{Sig}_{K_B^{-1}}(g^{b_1}))$$
$$A \rightarrow B : g^{a_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_1b_1})}(m_1) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_1b_1}))}(g^{a_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_1b_1})}(m_1))$$
$$B \rightarrow A : g^{b_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2b_1})}(m_2) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_2b_1}))}(g^{b_2} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2b_1})}(m_2))$$
$$A \rightarrow B : g^{a_3} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2b_2})}(m_3) \mid \mathsf{MAC}_{\mathsf{H}(\mathsf{H}(g^{a_2b_2}))}(g^{a_3} \mid \mathsf{Enc}_{\mathsf{H}(g^{a_2b_2})}(m_3))$$

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. $g^{a_1b_1}$, $g^{a_2b_1}$, $g^{a_2b_2}$) 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.

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).

Benchmark	x86+SGX instructions	BoogiePL statements	Moat proof	Policy Annotations
OTP	188	1774	9.9 sec	4
Notary	147	1222	3.2 sec	2
OTR IM	251	2191	7.8 sec	7
Query	575	4727	55 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 [29] 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 [23] seeks to provide similar guarantees via extension to the x86 instruction set. There are some recent efforts on using SGX for trusted computation. Haven [10] is a system that exploits Intel SGX for shielded execution of unmodified Windows applications. It links the application together with a runtime library OS that implements the Windows 8 API. However, it does not provide any confidentiality or integrity guarantees, and includes a significant TCB. VC3 [33] 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. Santos et al. [32] seek to build a trusted language runtime for mobile applications based on ARM TrustZone [4]. These design efforts have thrown up very interesting associated verification questions, and our paper seeks to address these with a special focus on Intel SGX.

Verifying Information Flow on Programs. Checking implementation code for safety is also a well studied problem. Type systems proposed by Sabelfeld et al. [31], Barthe et al. [9], and Volpano et al. [34] enable the programmer to annotate variables that hold secret values, and ensure that these values do not leak. Balliu et al. [5] automate information flow analysis of ARMv7 machine code. Languages and verification techniques also exist for quantitative information flow (e.g., [22]). 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. 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. [11, 12]). 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 [19].

10. CONCLUSION

This paper introduces a technique for verifying information flow properties of 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, we plan to design a high-level language from which we generate machine code, enclave model, and a protocol model.

11. ACKNOWLEDGMENTS

This research is supported in part by SRC contract 2460.001 and NSF STARSS grant 1528108. We gratefully acknowledge Brent ByungHoon Kang and the anonymous reviewers for their insightful feedback.

12. REFERENCES

[1] Available at http://www.cryptopp.com/.
[2] https://devmoat.github.io.
[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.
[4] ARM Security Technology - Building a Secure System using TrustZone Technology. ARM Technical White Paper.
[5] M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1080–1091, New York, NY, USA, 2014. ACM.
[6] 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.
[7] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05*, pages 82–87, 2005.
[8] 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.
[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. In *Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

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

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

[18] 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.

[19] 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.

[20] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of API-level exploits. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 312–321, May 2005.

[21] 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, 2014.

[22] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 261–269. ACM, 2010.

[23] 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.

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

[25] 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.

[26] 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.

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

[28] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 129–142, New York, USA, 1997.

[29] 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.

[30] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[31] 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.

[32] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using ARM TrustZone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, pages 67–80. ACM, 2014.

[33] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 38–54, 2015.

[34] 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.

[35] J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st Conference on Programming Language Design and Implementation*, pages 99–110, 2010.