Moat: Verifying Confidentiality of Enclave Programs

Rohit Sinha\textsuperscript{1}, Sriram Rajamani\textsuperscript{2}, Sanjit Seshia\textsuperscript{1}, Kapil Vaswani\textsuperscript{2}

University of California, Berkeley\textsuperscript{1}
Microsoft Research\textsuperscript{2}

CCS 2015
Online Banking Today

![Diagram showing a bank server connected to a client on commodity hardware, with layers of browser code, browser heap, and OS/hypervisor]

- Bank Server
- Client on Commodity Hardware
  - Browser Code
  - Browser Heap
  - OS/Hypervisor
Online Banking Today

```c
// Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey,
                                 server_dh_pubkey);

// encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));

// receive user's account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
```
Online Banking Today

```c
//Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey,
                                server_dh_pubkey);
//encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));
//receive user's account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
```

```c
struct untrusted_heap {
  byte client_dh_pubkey[64];
  byte session_key[32];
  byte secret1[64];
  byte secret2[32];
}
```
Online Banking Today

---

```
//Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey,
                                 server_dh_pubkey);

//encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));
//receive user's account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
```

```
struct untrusted_heap {
    byte client_dh_pubkey[64];
    byte session_key[32];
    byte secret1[64];
    byte secret2[32];
}
```
Online Banking Today

//Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();

session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey, server_dh_pubkey);

//encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));

//receive user’s account secret
signed_ciphertext = IO_RECV();
assert(CRYPTO_VERIFY_SIGNATURE(signed_ciphertext));

secret2 = CRYPTO_DECRYPT(signed_ciphertext);

// Housekeeping struct
struct untrusted_heap {
    byte client_dh_pubkey[64];
    byte session_key[32];
    byte secret1[64];
    byte secret2[32];
}
Primitives for Online Banking

```c
// Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey,
                                server_dh_pubkey);

// encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));
// receive user's account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);

struct untrusted_heap {
    byte client_dh_pubkey[64];
    byte session_key[32];
    byte secret1[64];
    byte secret2[32];
};
```
Primitives for Online Banking

1. Trusted computation without interference from adversary

```c
//Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey, server_dh_pubkey);

//encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));

//receive user's account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);

struct untrusted_heap {
    byte client_dh_pubkey[64];
    byte session_key[32];
    byte secret1[64];
    byte secret2[32];
}
```
Primitives for Online Banking

1. Trusted computation without interference from adversary

```c
// Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey,
                                 server_dh_pubkey);
```

```c
// encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));
```

```c
// receive user’s account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
```

2. Protected memory for computing and storing secrets

```c
struct untrusted_heap {
    byte client_dh_pubkey[64];
    byte session_key[32];
    byte secret1[64];
    byte secret2[32];
}
```
Primitives for Online Banking

1. Trusted computation without interference from adversary

```c
//Diffie-Hellman (SSL) key exchange protocol
client_dh_pubkey = CRYPTO_DH_PUBKEY();
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey, server_dh_pubkey);

//encrypt and send username and password
secret1 = concatenate(username, password);
IO_SEND(CRYPTO_ENCRYPT(session_key, secret1));
//receive user's account secret
signed_ciphertext = IO_RECV();
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
```

2. Protected memory for computing and storing secrets

```c
struct untrusted_heap {
    byte client_dh_pubkey[64];
    byte session_key[32];
    byte secret1[64];
    byte secret2[32];
}
```

3. Trusted measurement and remote attestation
Can we use **trusted hardware primitives** to defend against **privileged malware**?
Can we use trusted hardware primitives to defend against privileged malware?

Yes. Intel SGX, ARM TrustZone, Sancus, etc.
Can we use trusted hardware primitives to defend against privileged malware?

Yes. Intel SGX, ARM TrustZone, Sancus, etc.

How do we ensure that the programmer uses these primitives in a provably correct way?
Can we use **trusted hardware primitives** to defend against **privileged malware**?

Yes. Intel SGX, ARM TrustZone, Sancus, etc.

How do we ensure that the programmer uses these primitives in a **provably correct** way?

Focus of this paper!
Background on SGX
Defending Against Privileged Malware using Intel SGX
Defending Against Privileged Malware using Intel SGX

Enclave can invoke SGX instructions:

- **ereport** (attestation)
- **egetkey** (enclave-specific encryption key)
Online Banking using SGX

asm ("ereport %0 : "r" (signed_report));
memcpy(untrusted_heap, signed_report);
asm ("eeexit");

client_dh_pubkey = CRYPTO_DH_PUBKEY();
memcpy(server_dh_pubkey, untrusted_heap);
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey, server_dh_privkey);
asm ("eeexit");

secret1 = concatenate(username, password);
memcpy(untrusted_heap,
    CRYPTO_ENCRYPT(session_key, secret1));
asm ("eeexit");

memcpy(signed_ciphertext, untrusted_heap);
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
asm ("eeexit");
Online Banking using SGX

enclave accesses untrusted memory for I/O

asm ("ereport " : "r" (signed_report));
memcpy(untrusted_heap, signed_report);
asm ("eexit");

client_dh_pubkey = CRYPTO_DH_PUBKEY();
memcpy(server_dh_pubkey, untrusted_heap);
session_key = CRYPTO_DH_PRIVKEY(client_dh_pubkey,
server_dh_privkey);
asm ("eexit");

secret1 = concatenate(username, password);
memcpy(untrusted_heap,
CRYPTO_ENCRYPT(session_key, secret1));
asm ("eexit");

memcpy(signed_ciphertext, untrusted_heap);
assert CRYPTO_VERIFY_SIGNATURE(signed_ciphertext);
secret2 = CRYPTO_DECRYPT(signed_ciphertext);
asm ("eexit");
Online Banking using SGX

... 
ecreate;
foreach page in enclave {
    eadd(page);
    eextend(page);
}
einit;

asm ("eenter");
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();

asm ("eenter");
IO_SEND(encrypted_secret1);

asm ("eenter");
signed_ciphertext = IO_RECV();
...

Bank Server

Client on SGX

Enclave

Enclave Stack

Enclave Code

Enclave Heap

App Code

App Heap

OS / Hypervisor
Online Banking using SGX

```c
... 
ecreate;
foreach page in enclave {
    eadd(page);
    eextend(page);
}
einit;
asm ("eenter");
IO_SEND(client_dh_pubkey);
server_dh_pubkey = IO_RECV();
asm ("eenter");
IO_SEND(encrypted_secret1);
asm ("eenter");
signed_ciphertext = IO_RECV();
... 
```

I/O operations are untrusted
Online Banking using SGX

enclave creation is untrusted, and verified by attestation

```c
... 
  ecreate;
  foreach page in enclave {
    eadd(page);
    eextend(page);
  }
  einit;

  asm ("eenter");
  IO_SEND(client_dh_pubkey);
  server_dh_pubkey = IO_RECV();

  asm ("eenter");
  IO_SEND(encrypted_secret1);

  asm ("eenter");
  signed_ciphertext = IO_RECV();
...```

I/O operations are untrusted
Goal: Proving Confidentiality
Goal: Proving Confidentiality

- SGX does not prevent an enclave from leaking secrets
Goal: Proving Confidentiality

• SGX does **not** prevent an enclave from leaking secrets

  ▸ `asm("egetkey %0" : "r" (sealing_key));
    memcpy(untrusted_memory, sealing_key, 16);`
Goal: Proving Confidentiality

- SGX does not prevent an enclave from leaking secrets

  - `asm("egetkey %0": "r" (sealing_key));`
    - `memcpy(untrusted_memory, sealing_key, 16);`

  - `memcpy(untrusted_memory, encrypted_value, untrusted_memory_struct.size_field);`
Goal: Proving Confidentiality

- SGX does **not** prevent an enclave from leaking secrets
  - `asm("egetkey %0" : "r" (sealing_key));`
  - `memcpy(untrusted_memory, sealing_key, 16);`
  - `memcpy(untrusted_memory, encrypted_value, untrusted_memory_struct.size_field);`

- We want to prove **confidentiality**: enclave program does not leak secrets in any potential execution
Goal: Proving Confidentiality

- SGX does not prevent an enclave from leaking secrets
  - asm("egetkey %0" : "r" (sealing_key));
    memcpy(untrusted_memory, sealing_key, 16);
  - memcpy(untrusted_memory, encrypted_value,
      untrusted_memory_struct.size_field);

- We want to prove confidentiality: enclave program does not leak secrets in any potential execution
  - Verifier must reason about enclave’s use of SGX and x86 instructions, and adversarial operations on the enclave
Contributions

**Moat**, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
- Formal Models of the Adversary
- Static analysis that proves that an enclave program (running in the presence of an adversary) satisfies confidentiality
Running Example

**C code**

```c
asm("egetkey %0" : "r" (sealing_key));
int *ptr = *((int **) 0xdead);
int secret = *sealing_key;
*ptr = secret;
```

**x86-64**

```assembly
egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %(%rdx)
```
Running Example

C code

```
asm("egetkey %0" : "r" (sealing_key));
int *ptr = *((int **) 0xdead);
int secret = *sealing_key;
*ptr = secret;
```

x86-64

```
egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %(%rdx)
```
Running Example

C code

```
asm("egetkey %0" : "r" (sealing_key));
int *ptr = *((int **) 0xdead);
int secret = *sealing_key;
*ptr = secret;
```

x86-64

```
egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %(%rdx)
```
Running Example

C code

```c
asm("egetkey %0": "r" (sealing_key));
int *ptr = *((int **) 0xdead);
int secret = *sealing_key;
*ptr = secret;
```

x86-64

```assembly
egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %0
```
Overview of Moat

<table>
<thead>
<tr>
<th>Enclave Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64 &amp; SGX instructions</td>
</tr>
</tbody>
</table>
Overview of Moat

Enclave Program

x86-64 & SGX instructions

instruction lifting to RISC, and formal modeling

Formal Model

RISC instructions in formal language
Overview of Moat

**Enclave Program**
- x86-64 & SGX instructions
  - Instruction lifting to RISC, and formal modeling

**Formal Model**
- RISC instructions in formal language

**Composed Model**
- Adversary Modeling
- RISC instructions & Adv. Operations
Overview of Moat

- **Enclave Program**
  - x86-64 & SGX instructions

- **Formal Model**
  - RISC instructions in formal language

- **Composed Model**
  - RISC instructions & Adv. Operations

- **Confidentiality Verifier**
  - proof or exploit

- Instruction lifting to RISC, and formal modeling

- Adversary Modeling

- Static program analysis of confidentiality property
Outline

Enclave Program

x86-64 & SGX instructions

instruction lifting to RISC, and formal modeling

Formal Model

RISC instructions in formal language

Composed Model

RISC instructions & Adv. Operations

Adversary Modeling

Confidentiality Verifier

proof or exploit

Static program analysis of confidentiality property
Outline

**Enclave Program**
- x86-64 & SGX instructions
  - Instruction lifting to RISC, and formal modeling

**Formal Model**
- RISC instructions in formal language

**Composed Model**
- RISC instructions & Adv. Operations
- Adversary Modeling

**Confidentiality**
- Static program analysis of confidentiality property
- Proof or exploit
Formal Model of Enclave

egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %(%rdx)
Formal Model of Enclave

egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %(%rdx)

mem := egetkey(mem, rbx, rcx);
rdx := load(mem, 0xdead);
rax := load(mem, rcx);
mem := store(mem, rdx, rax);
Formal Model of Enclave

SGX instructions:
gegetkey, ecreate, etc.
modeled using
Intel Programmer Reference Manual

x86 instructions:
add, jmp, etc.
modeled using
Binary Analysis Platform
[Brumley et al. CAV’11]

```assembly
egetkey
movl 0xdead, %rdx
movl (%rcx), %rax
movl %rax, %(%rdx)
```

```python
mem := egetkey(mem, rbx, rcx);
rdx := load(mem, 0xdead);
rax := load(mem, rcx);
mem := store(mem, rdx, rax);
```
Formal Model of Enclave

Assumptions
- no self-modifying code
- single-threaded
- control flow integrity

SGX instructions:
egetkey, ecreate, etc.
modeled using
Intel Programmer Reference Manual

x86 instructions:
add, jmp, etc.
modeled using
Binary Analysis Platform
[Brumley et al. CAV'11]
Formal Model of Enclave
SGX Instructions

create(laddr: bv64, secs: struct_secs): {
    paddr := page_table[laddr];
    measurement := hash(…)
    epcm[pa] := …
    mem[pa] := {Secs.baseaddr,
                measurement, …}
}
Formal Model of Enclave

SGX Instructions

`ecreate(laddr: bv64, secs: struct_secs): {
paddr := page_table[laddr];
measurement := hash(…)
epcm[pa] := …
mem[pa] := {Secs.baseaddr,
measurement, …}
}

x86 Instructions

`add(rbx: bv64, rax: bv64): {
rax := rbx +bv64 rax;
of := <overflow calculation>
zf := <zero flag calculation>
…
}`
Outline

**Enclave Program**
- x86-64 & SGX instructions

**Formal Model**
- RISC instructions in formal language

**Composed Model**
- RISC instructions & Adv. Operations

**Confidentiality Verifier**
- proof or exploit

- instruction lifting to RISC, and formal modeling

- Adversary Modeling

- Static program analysis of confidentiality property
Outline

Enclave Program

x86-64 & SGX instructions

Formal Model

RISC instructions in formal language

Composed Model

RISC instructions & Adv. Operations

Confidentiality Verifier

proof or exploit

Static program analysis of confidentiality property

Adversary Modeling

instruction lifting to RISC, and formal modeling
Adversary Model
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary **operations** after each enclave instruction:
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary operations after each enclave instruction:

1. Havoc non-enclave memory
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary **operations** after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode.

Adversary **operations** after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary operations after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
4. Control I/O devices and generate interrupts
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary operations after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
4. Control I/O devices and generate interrupts
5. Invoke any SGX instruction
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary operations after each enclave instruction:
1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
4. Control I/O devices and generate interrupts
5. Invoke any SGX instruction

Adversary observations after each enclave instruction:
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary operations after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
4. Control I/O devices and generate interrupts
5. Invoke any SGX instruction

Adversary observations after each enclave instruction:

1. non-enclave memory
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary **operations** after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
4. Control I/O devices and generate interrupts
5. Invoke any SGX instruction

Adversary **observations** after each enclave instruction:

1. non-enclave memory
Adversary Model

Attacker can execute in ring-0, ring-3 and system management mode

Adversary **operations** after each enclave instruction:

1. Havoc non-enclave memory
2. Havoc registers on non-enclave CPUs
3. Havoc page tables
4. Control I/O devices and generate interrupts
5. Invoke any SGX instruction

Adversary **observations** after each enclave instruction:

1. non-enclave memory
Composing Enclave with Adversary
Composing Enclave with Adversary

**Theorem**: any enclave state that is reachable in the presence of *General Adversary* is reachable in the presence of *Havocing Adversary* (modulo termination)
Composing Enclave with Adversary

**Theorem**: any enclave state that is reachable in the presence of General Adversary is reachable in the presence of Havocing Adversary (modulo termination)

Machine-checked proof with the following intuition:
Composing Enclave with Adversary

**Theorem:** any enclave state that is reachable in the presence of General Adversary is reachable in the presence of Havocing Adversary (modulo termination)

Machine-checked proof with the following intuition:

✓ CPU terminates enclave after detecting modification to page tables that control enclave memory
Composing Enclave with Adversary

**Theorem**: any enclave state that is reachable in the presence of General Adversary is reachable in the presence of Havocing Adversary (modulo termination)

Machine-checked proof with the following intuition:

- ✓ CPU terminates enclave after detecting modification to page tables that control enclave memory
- ✓ CPU enforces memory isolation between enclaves
Composing Enclave with Adversary

**Theorem**: any enclave state that is reachable in the presence of General Adversary is reachable in the presence of Havocing Adversary (modulo termination)

Machine-checked proof with the following intuition:

✓ CPU terminates enclave after detecting modification to page tables that control enclave memory

✓ CPU enforces memory isolation between enclaves

✓ …
Composing Enclave with Adversary

```plaintext
mem := egetkey(mem, rbx, rcx);
rdx := load(mem, Oxdead);
rax := load(mem, rcx);
mem := store(mem, rdx, rax);
```
Composing Enclave with Adversary

```plaintext
mem := egetkey(mem, rbx, rcx);
rdx := load(mem, 0xdead);
rax := load(mem, rcx);
mem := store(mem, rdx, rax);
```

```plaintext
havoc mem_enc;
mem := egetkey(mem, rbx, rcx);
mem := egetkey(mem, rbx, rcx);
mem := egetkey(mem, rbx, rcx);
mem := egetkey(mem, rbx, rcx);
```
Composing Enclave with Adversary

```plaintext
mem := egetkey(mem, rbx, rcx);
rdx := load(mem, 0xdead);
rax := load(mem, rcx);
mem := store(mem, rdx, rax);
```

 havocs untrusted memory

```plaintext
havoc mem_enc;
mem := egetkey(mem, rbx, rcx);
havoc mem_enc;
rdx := load(mem, 0xdead);
havoc mem_enc;
rax := load(mem, rcx);
havoc mem_enc;
mem := store(mem, rdx, rax);
```
Composing Enclave with Adversary

- Composed program models all potential executions in the presence of a General Adversary
- Enables sound proofs of safety properties

```plaintext
mem := egetkey(mem, rbx, rcx);
rax := load(mem, 0xdead);
mem := store(mem, rdx, rax);
```

```plaintext
havoc mem_enc;
mem := egetkey(mem, rbx, rcx);
havoc mem_enc;
rax := load(mem, 0xdead);
havoc mem_enc;
mem := store(mem, rdx, rax);
```
Outline

**Enclave Program**
- x86-64 & SGX instructions

**Formal Model**
- RISC instructions in formal language

**Composed Model**
- RISC instructions & Adv. Operations

**Adversary Modeling**

**Confidentiality Verifier**
- Proof or exploit

**Static program analysis of confidentiality property**
Outline

Enclave Program
x86-64 & SGX instructions

 Formal Model
RISC instructions in formal language

Composed Model
RISC instructions & Adv. Operations

Adversary Modeling

Static program analysis of confidentiality property

Confidentiality Verifier
proof or exploit
Non-interference: For any two traces of enclave code such that the two traces have different secret values and equivalent adversary operations, the adversary’s observations are equivalent in the two traces.

[Sabelfeld et al. JSAC’03]
Proving Confidentiality of Composed Program

Hyper Property
[Clarkson et al. JCS’10]

**Non-interference:** For any two traces of enclave code such that the two traces have different secret values and equivalent adversary operations, the adversary’s observations are equivalent in the two traces.

[Sabelfeld et al. JSAC’03]
Proving Confidentiality of Composed Program

Hyper Property
[Clarkson et al. JCS’10]

Non-interference: For any two traces of enclave code such that the two traces have different secret values and equivalent adversary operations, the adversary’s observations are equivalent in the two traces.

[Sabelfeld et al. JSAC’03]

ExplicitLeaks:
@secret rax
mem := store(mem, 0xdead, rax)
Proving Confidentiality of Composed Program

Non-interference: For any two traces of enclave code such that the two traces have different secret values and equivalent adversary operations, the adversary’s observations are equivalent in the two traces.

Explicit Leaks:
@secret rax
mem := store(mem, 0xdead, rax)

Implicit Leaks:
@secret rax
if (rax == 42) {
    mem := store(mem, 0xdead, 1)
} else {
    mem := store(mem, 0xdead, 2)
}

Hyper Property
[Clarkson et al. JCS'10]

[Sabelfeld et al. JSAC'03]
Proving Confidentiality of Composed Program

```plaintext
havoc mem_enc;
mem := egetkey(mem, rbx, rcx);
havoc mem_enc;
rdx := load(mem, 0xdead);
havoc mem_enc;
rax := load(mem, rcx);
havoc mem_enc;
mem := store(mem, rdx, rax);
```
Proving Confidentiality of Composed Program

```plaintext
assume \( \forall i. (rcx \leq i < rcx + 16) \rightarrow C_{\text{mem}[i]} \leftrightarrow \text{true} \);

havoc mem_{\text{enc}};
mem := egetkey(mem, rbx, rcx);

havoc mem_{\text{enc}};
rdx := load(mem, 0xdead);

havoc mem_{\text{enc}};
rax := load(mem, rcx);

havoc mem_{\text{enc}};
mem := store(mem, rdx, rax);

havoc mem_{\text{enc}};
mem := egetkey(mem, rbx, rcx);

havoc mem_{\text{enc}};
C_{rdx} := C_{\text{mem}[0xdead]};

havoc mem_{\text{enc}};
rdx := load(mem, 0xdead);

assert \neg C_{rcx};

C_{rax} := C_{\text{mem}[C_{rcx}]};

havoc mem_{\text{enc}};
rax := load(mem, rcx);

assert \neg C_{rdx};

assert \neg \text{enc}(rdx) \rightarrow \neg C_{rax};

C_{\text{mem}[C_{rdx}]} := C_{rax};

havoc mem_{\text{enc}};
mem := store(mem, rdx, rax);
```
assume $\forall i. \ (rcx \leq i < rcx + 16) \rightarrow C_{\text{mem}[i]} \leftrightarrow \text{true}$;

havoc $\text{mem}_{\text{enc}}$; $\text{mem} := \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx})$;

$C_{\text{rdx}} := C_{\text{mem}[0\times\text{dead}]}$;

havoc $\text{mem}_{\text{enc}}$; $\text{rdx} := \text{load}(\text{mem}, 0\times\text{dead})$;

assert $\neg C_{\text{rcx}}$;

$C_{\text{rax}} := C_{\text{mem}[C_{\text{rcx}}]}$;

havoc $\text{mem}_{\text{enc}}$; $\text{rax} := \text{load}(\text{mem}, \text{rcx})$;

assert $\neg C_{\text{rdx}}$;

assert $\neg \text{enc}(\text{rdx}) \rightarrow \neg C_{\text{rax}}$;

$C_{\text{mem}[C_{\text{rdx}}]} := C_{\text{rax}}$;

havoc $\text{mem}_{\text{enc}}$; $\text{mem} := \text{store}(\text{mem}, \text{rdx}, \text{rax})$;
Proving Confidentiality of Composed Program

\begin{verbatim}
assume ∀i. (rcx ≤ i < rcx + 16) → C\text{mem}[i] ↔ true;

havoc mem\_enc; mem := egetkey(mem, rbx, rcx);
C\_rdx := C\text{mem}[0xdead];

havoc mem\_enc; rdx := load(mem, 0xdead);
assert ¬C\_rcx;
C\_rax := C\text{mem}[C\_rcx];

havoc mem\_enc; rax := load(mem, rcx);
assert ¬C\_rdx;
assert ¬enc(rdx) → ¬C\_rax;
C\text{mem}[C\_rdx] := C\_rax;

havoc mem\_enc; mem := store(mem, rdx, rax);
\end{verbatim}
Proving Confidentiality of Composed Program

```
assume \( \forall i. \ (rcx \leq i < rcx + 16) \rightarrow C_{\text{mem}}[i] \leftrightarrow \text{true} \);

havoc \text{ mem}_{\text{enc}}; \text{ mem } := \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx});

C_{\text{rdx}} := C_{\text{mem}}[0\times\text{dead}];

havoc \text{ mem}_{\text{enc}}; \text{ rdx } := \text{load}(\text{mem}, 0\times\text{dead});

assert \neg C_{\text{rcx}};

C_{\text{rax}} := C_{\text{mem}}[C_{\text{rcx}}];

havoc \text{ mem}_{\text{enc}}; \text{ rax } := \text{load}(\text{mem}, \text{rcx});

assert \neg C_{\text{rdx}};

assert \neg \text{enc}(\text{rdx}) \rightarrow \neg C_{\text{rax}};

C_{\text{mem}}[C_{\text{rdx}}] := C_{\text{rax}};

havoc \text{ mem}_{\text{enc}}; \text{ mem } := \text{store}(\text{mem}, \text{rdx}, \text{rax});
```

16-byte sealing key at address in rcx

ghost variables track secrets
Proving Confidentiality of Composed Program

\[
\text{assume } \forall i. \ (\text{rcx} \leq i < \text{rcx} + 16) \rightarrow C_{\text{mem}}[i] \leftrightarrow \text{true};
\]

\[
\text{havoc } \text{mem}_{\text{enc}}; \quad \text{mem} := \text{egetkey}(\text{mem}, \text{rbx}, \text{rcx});
\]

\[
C_{\text{rdx}} := C_{\text{mem}}[0x\text{dead}];
\]

\[
\text{havoc } \text{mem}_{\text{enc}}; \quad \text{rdx} := \text{load}(\text{mem}, 0x\text{dead});
\]

\[
\text{assert } \neg C_{\text{rcx}};
\]

\[
C_{\text{rax}} := C_{\text{mem}}[C_{\text{rcx}}];
\]

\[
\text{havoc } \text{mem}_{\text{enc}}; \quad \text{rax} := \text{load}(\text{mem}, \text{rcx});
\]

\[
\text{assert } \neg C_{\text{rdx}};
\]

\[
\text{assert } \neg \text{enc}(\text{rdx}) \rightarrow \neg C_{\text{rax}};
\]

\[
C_{\text{mem}}[C_{\text{rdx}}] := C_{\text{rax}};
\]

\[
\text{havoc } \text{mem}_{\text{enc}}; \quad \text{mem} := \text{store}(\text{mem}, \text{rdx}, \text{rax});
\]
Proving Confidentiality of Composed Program

```plaintext
assume ∀i. (rcx ≤ i < rcx + 16) → C_{mem}[i] ↔ true;

havoc mem_{enc}; mem := egetkey(mem, rbx, rcx);
C_{rdx} := C_{mem}[0xdead];

havoc mem_{enc}; rdx := load(mem, 0xdead);
assert ¬C_{rcx};
C_{rax} := C_{mem}[C_{rcx}];

havoc mem_{enc}; rax := load(mem, rcx);
assert ¬C_{rdx};
assert ¬enc(rdx) → ¬C_{rax};
C_{mem}[C_{rdx}] := C_{rax};

havoc mem_{enc}; mem := store(mem, rdx, rax);
```

- 16-byte sealing key at address in rcx
- Propagate secret in memory
- Propagate secret in memory
- Assert secret is not leaked to non-enclave memory

Ghost variables track secrets
Proving Confidentiality of Composed Program

A program must satisfy all assertions in all executions.

```
assume ∀i. (rcx ≤ i < rcx + 16) → C_mem[i] ↔ true;

havoc mem_enc;  mem := egotkey(mem, rbx, rcx);
C_rdx := C_mem[0xdead];

havoc mem_enc;  rdx := load(mem, 0xdead);
assert ¬C_rcx;
C_rax := C_mem[C_rcx];

havoc mem_enc;  rax := load(mem, rcx);
assert ¬C_rdx;
assert ¬enc(rdx) → ¬C_rax;
C_mem[C_rdx] := C_rax;

havoc mem_enc;  mem := store(mem, rdx, rax);
```

- 16-byte sealing key at address in rcx
- propagate secret in memory
- propagate secret in memory
- assert secret is not leaked to non-enclave memory

ghost variables track secrets
Proving Confidentiality of Composed Program

A program must satisfy all assertions in all executions.

Verifier reasons about SGX instructions: egetkey, ereport, eexit
Proving Confidentiality of Composed Program

assume \( \forall i. (rcx \leq i < rcx + 16) \rightarrow C_{mem}[i] \leftrightarrow \text{true} \);

\text{havoc mem-enc; mem := egetkey(mem, rbx, rcx);}

\text{C_{rdx} := C_{mem}[0xdead];}

\text{havoc mem-enc; rdx := load(mem, 0xdead);}

\text{assert \neg C_{rcx};}

\text{C_{rax} := C_{mem}[C_{rcx}];}

\text{havoc mem-enc; rax := load(mem, rcx);}

\text{assert \neg C_{rdx};}

\text{assert \neg enc(rdx) \rightarrow \neg C_{rax};}

\text{C_{mem}[C_{rdx}] := C_{rax};}

\text{havoc mem-enc; mem := store(mem, rdx, rax);}
Proving Confidentiality of Composed Program

assume ∀i. (rcx ≤ i < rcx + 16) → C_{mem}[i] ↔ true;

havoc mem_{enc}; mem := egetkey(mem, rbx, rcx);
C_{rdx} := C_{mem}[0xdead];

havoc mem_{enc}; rdx := load(mem, 0xdead);
assert ¬C_{rcx};
C_{rax} := C_{mem}[C_{rcx}];

havoc mem_{enc}; rax := load(mem, rcx);
assert ¬C_{rdx};
assert ¬enc(rdx) → ¬C_{rax};
C_{mem}[C_{rdx}] := C_{rax};

havoc mem_{enc}; mem := store(mem, rdx, rax);
Proving Confidentiality of Composed Program

```
assume \( \forall i. \ (rcx \leq i < rcx + 16) \rightarrow C_{mem}[i] \leftrightarrow true; \)

havoc mem\_enc; mem := egetkey(mem, rbx, rcx);
C\_rdx := C_{mem}[0\times dead];

havoc mem\_enc; rdx := load(mem, 0\times dead);
assert \neg C_{rcx};
C_{rax} := C_{mem}[C_{rcx}];

havoc mem\_enc; rax := load(mem, rcx);
assert \neg C_{rdx};
assert \neg enc(rdx) \rightarrow \neg C_{rax};
C_{mem}[C_{rdx}] := C_{rax};

havoc mem\_enc; mem := store(mem, rdx, rax);
```

Boogie Program Verifier [Barnett et al. FMCO'05]
Z3 SMT Solver [de Moura et al. TACAS'08]
Caveat
Caveat

• Type system must handle encrypt differently
  ‣ For any function f, f(x) is marked secret if x is a secret

    @secret x
    memcpy(untrusted_heap, encrypt(x));
Caveat

• Type system must handle encrypt differently
  ‣ For any function \( f \), \( f(x) \) is marked secret if \( x \) is a secret

```c
@secret x
memcpy(untrusted_heap, encrypt(x));
```

• Declassify encrypted data by removing assertion on memcpy
Evaluation

- One time Password (OTP) scheme
- Notary Service
- Off-the-record (OTR) Messaging
- Query Processing on Encrypted Database

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>x86+SGX instructions</th>
<th>BoogiePL statements</th>
<th>Moat proof</th>
<th>Policy Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTP</td>
<td>188</td>
<td>1774</td>
<td>9.9 sec</td>
<td>4</td>
</tr>
<tr>
<td>Notary</td>
<td>147</td>
<td>1222</td>
<td>3.2 sec</td>
<td>2</td>
</tr>
<tr>
<td>OTR IM</td>
<td>251</td>
<td>2191</td>
<td>7.8 sec</td>
<td>7</td>
</tr>
<tr>
<td>Query</td>
<td>575</td>
<td>4727</td>
<td>55 sec</td>
<td>9</td>
</tr>
</tbody>
</table>
Related Work
## Related Work

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haven [Baumann et al, OSDI’14]</td>
<td>Legacy Windows Applications in Enclaves</td>
<td>High TCB, no formal guarantee</td>
</tr>
</tbody>
</table>
## Related Work

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Haven</strong> [Baumann et al, OSDI'14]</td>
<td>Legacy Windows Applications in Enclaves</td>
<td>High TCB, no formal guarantee</td>
</tr>
<tr>
<td><strong>VC3</strong> [Schuster et al, S&amp;P’15]</td>
<td>Map Reduce Computation in Enclaves</td>
<td>confidentiality and integrity with low TCB, but no formal guarantee</td>
</tr>
</tbody>
</table>
## Related Work

<table>
<thead>
<tr>
<th>Work</th>
<th>Application</th>
<th>Guarantees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Haven</strong></td>
<td>Legacy Windows Applications in Enclaves</td>
<td>High TCB, no formal guarantee</td>
</tr>
<tr>
<td>[Baumann et al, OSDI’14]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>VC3</strong></td>
<td>Map Reduce Computation in Enclaves</td>
<td>confidentiality and integrity with low TCB, but no formal guarantee</td>
</tr>
<tr>
<td>[Schuster et al, S&amp;P’15]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Jif</strong></td>
<td>Java applications Crypto protocols</td>
<td>no reasoning of SGX instructions no guarantee on machine code</td>
</tr>
<tr>
<td>[Myers et al, ‘06]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>[Swamy et al, ICFP’11]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary
Summary

Moat, an instruction-level verifier for enclave programs
Summary

Moat, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
Summary

Moat, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
- Formal Models of the Adversary
Summary

**Moat**, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
- Formal Models of the Adversary
- Verification algorithm that proves that an enclave program (running in the presence of an adversary) satisfies confidentiality
Summary

**Moat**, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
- Formal Models of the Adversary
- Verification algorithm that proves that an enclave program (running in the presence of an adversary) satisfies confidentiality

**Future Work**
Summary

**Moat**, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
- Formal Models of the Adversary
- Verification algorithm that proves that an enclave program (running in the presence of an adversary) satisfies confidentiality

**Future Work**

- Scalable verification using verified runtime APIs
Summary

**Moat**, an instruction-level verifier for enclave programs

- Formal Models of SGX instructions
- Formal Models of the Adversary
- Verification algorithm that proves that an enclave program (running in the presence of an adversary) satisfies confidentiality

**Future Work**

- Scalable verification using verified runtime APIs
- Relax assumptions: control flow integrity, single threaded enclaves
Questions?