Binary Code Retrofitting and Hardening Using SGX

Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang,
XiaoFeng Wang, and Dinghao Wu

The Pennsylvania State University, Indiana University Bloomington,
Institute of Information Engineering

Motivation

- Available in Intel Commercial CPUs
- Hardware isolated memory regions
- Protection under a strong adversary model
- A bit performance penalty (~10%)

Motivation

- Available in Intel Commercial CPUs
- Hardware isolated memory regions
- Protection under a strong adversary model
- A bit performance penalty

Can binary code hardening benefit from SGX?

Motivation

- Graphene-SGX, Haven
  - Large TCB (53 kloc for Graphene-SGX)

Shielding applications from an untrusted cloud with Haven

Andrew Baumann Marcus Peinado Galen Hunt
Microsoft Research

Abstract

Today's cloud computing infrastruc
tial trust. Cloud users rely on both t
its globally-distributed software/har
expose any of their private data.
We introduce the notion of shiel
protects the confidentiality and inte
its data from the platform on which
operator's OS, VM and firmware).
is the first system to achieve shiel
modified legacy applications, inclu
Apache, on a commodity OS (Wi
ity hardware. Haven leverages the
Intel SGX to defend against privil
cal attacks such as memory probes,
dual challenges of executing unmo
and protecting them from a malici
motivated recent changes in the SG

The current best practice for protecting secrets in the

Introduction to Intel SGX Support

chiache edited this page Jul 20, 2016 · 10 revisions

What is Intel SGX?

SGX (Software Guard Extenstion) is a new feature of the latest Intel CPUs. According to https://github.com/ayeks/SGX-hardware, SGX is available in CPUs that are launched after October 1st, 2015.

Intel SGX is designed to protection critical applications against potentially malicious system stack, from the operating systems to hardware (CPU itself excluded). SGX creates a hardware encrypted memory region (so-called **enclaves**) from the protected applications, that neither compromised operating systems, nor hardware attack such as **cold-boot attack** can retrieve the application secrets.

Why use Graphene Library OS for Intel SGX?

Porting applications to Intel SGX platform can be cumbersome. To secure an application with SGX, developers must recompile the application executable with the Intel SDK (Linux SDK: https://github.com/01org/linux-sgx). Moreover, the secured applications have *no* access to any OS features, such as opening a file, creating a network connection, or cloning a thread. For any interaction with the host, developers must define untrusted interfaces that the secure applications can call to leave the enclaves.

Graphene Library OS provides the OS features needed by the applications, right inside the SGX enclaves. To secure any applications, developers can directly load native, unmodified binaries into enclaves, with minimal porting efforts. Graphene Library OS provides signing tool to sign all binaries that are loaded into the enclaves, just like the Intel SGX SDK.

Basics

- Introduction to Graphene
- Quick Start
- Run Applications in Graphene
- Manifest Syntax
- Implemented System Calls
- Building Linux Kernel Support

Intel SGX Support

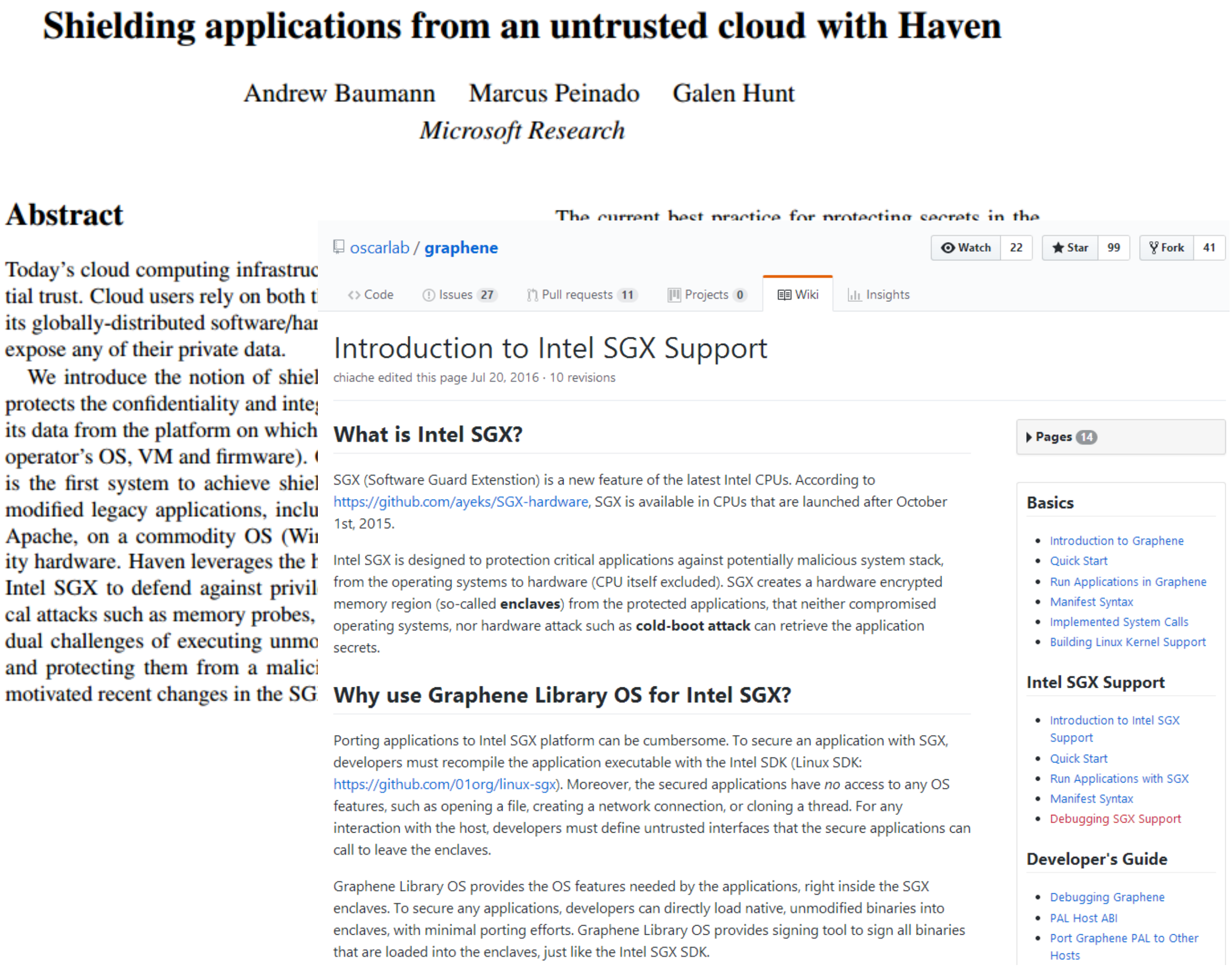
- Introduction to Intel SGX Support
- Quick Start
- Run Applications with SGX
- Manifest Syntax
- Debugging SGX Support

Developer's Guide

- Debugging Graphene
- PAL Host ABI
- Port Graphene PAL to Other Hosts

Motivation

- Graphene-SGX, Haven
  - Large TCB (53 kloc for Graphene-SGX)
- Our solution
  - Techniques to dissect binary code into multiple components
  - Put into separated enclaves



Shielding applications from an untrusted cloud with Haven

Andrew Baumann Marcus Peinado Galen Hunt
Microsoft Research

Abstract

Today's cloud computing infrastruc
tial trust. Cloud users rely on both t
its globally-distributed software/har
expose any of their private data.
We introduce the notion of shiel
protects the confidentiality and inte
its data from the platform on which
operator's OS, VM and firmware). (
is the first system to achieve shiel
modified legacy applications, inclu
Apache, on a commodity OS (Wi
ity hardware. Haven leverages the h
Intel SGX to defend against privil
cal attacks such as memory probes,
dual challenges of executing unmo
and protecting them from a malici
motivated recent changes in the SG

The current best practice for protecting secrets in the

Introduction to Intel SGX Support

chiache edited this page Jul 20, 2016 · 10 revisions

What is Intel SGX?

SGX (Software Guard Extenstion) is a new feature of the latest Intel CPUs. According to https://github.com/ayeks/SGX-hardware, SGX is available in CPUs that are launched after October 1st, 2015.

Intel SGX is designed to protection critical applications against potentially malicious system stack, from the operating systems to hardware (CPU itself excluded). SGX creates a hardware encrypted memory region (so-called **enclaves**) from the protected applications, that neither compromised operating systems, nor hardware attack such as **cold-boot attack** can retrieve the application secrets.

Why use Graphene Library OS for Intel SGX?

Porting applications to Intel SGX platform can be cumbersome. To secure an application with SGX, developers must recompile the application executable with the Intel SDK (Linux SDK: https://github.com/01org/linux-sgx). Moreover, the secured applications have *no* access to any OS features, such as opening a file, creating a network connection, or cloning a thread. For any interaction with the host, developers must define untrusted interfaces that the secure applications can call to leave the enclaves.

Graphene Library OS provides the OS features needed by the applications, right inside the SGX enclaves. To secure any applications, developers can directly load native, unmodified binaries into enclaves, with minimal porting efforts. Graphene Library OS provides signing tool to sign all binaries that are loaded into the enclaves, just like the Intel SGX SDK.

Basics

- Introduction to Graphene
- Quick Start
- Run Applications in Graphene
- Manifest Syntax
- Implemented System Calls
- Building Linux Kernel Support

Intel SGX Support

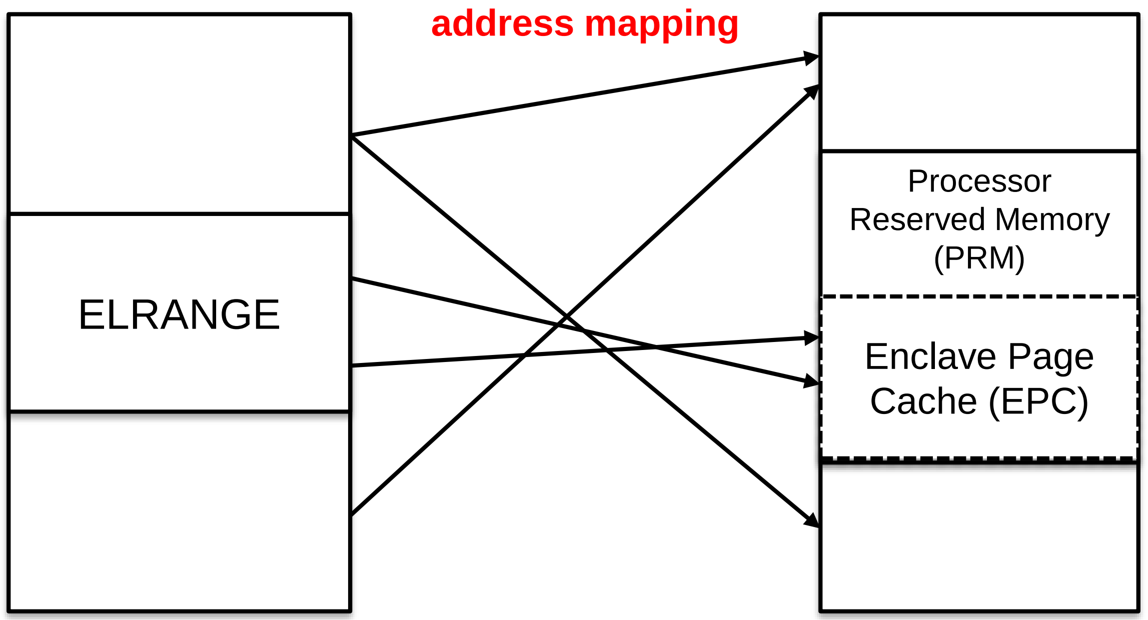
- Introduction to Intel SGX Support
- Quick Start
- Run Applications with SGX
- Manifest Syntax
- Debugging SGX Support

Developer's Guide

- Debugging Graphene
- PAL Host ABI
- Port Graphene PAL to Other Hosts

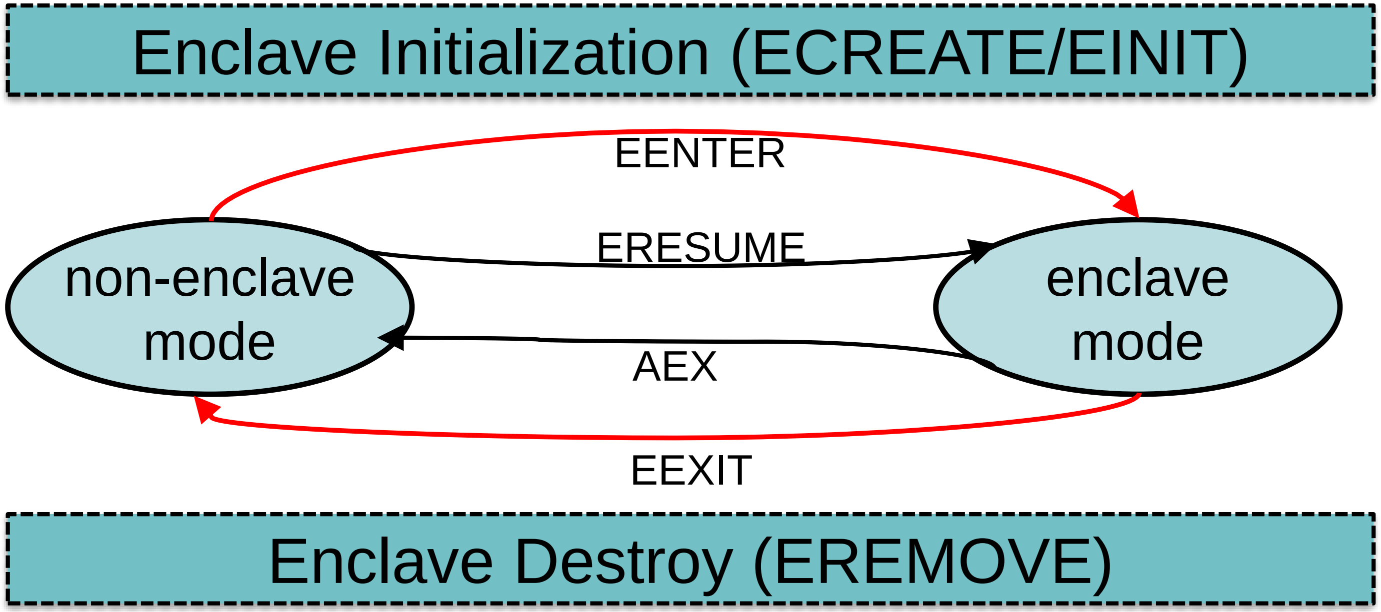
Background on SGX

- Two capabilities
  - change in enclave memory access semantics
  - protection of the address mappings of the application



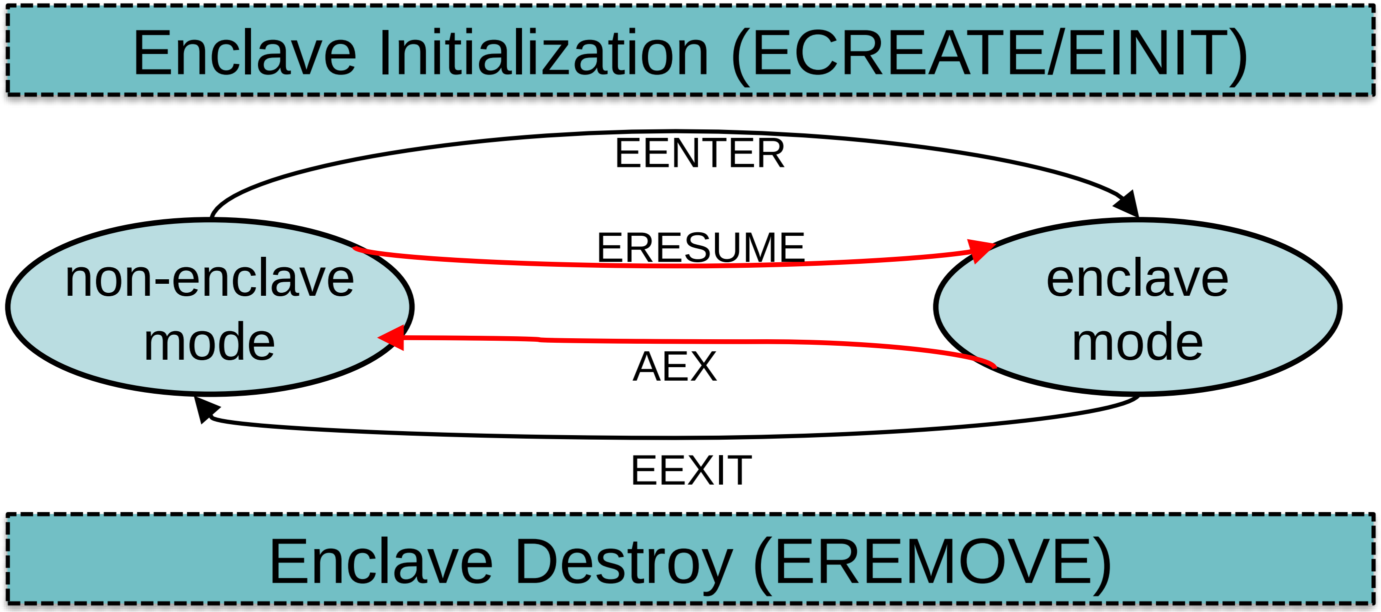
Background on SGX

- Life cycle



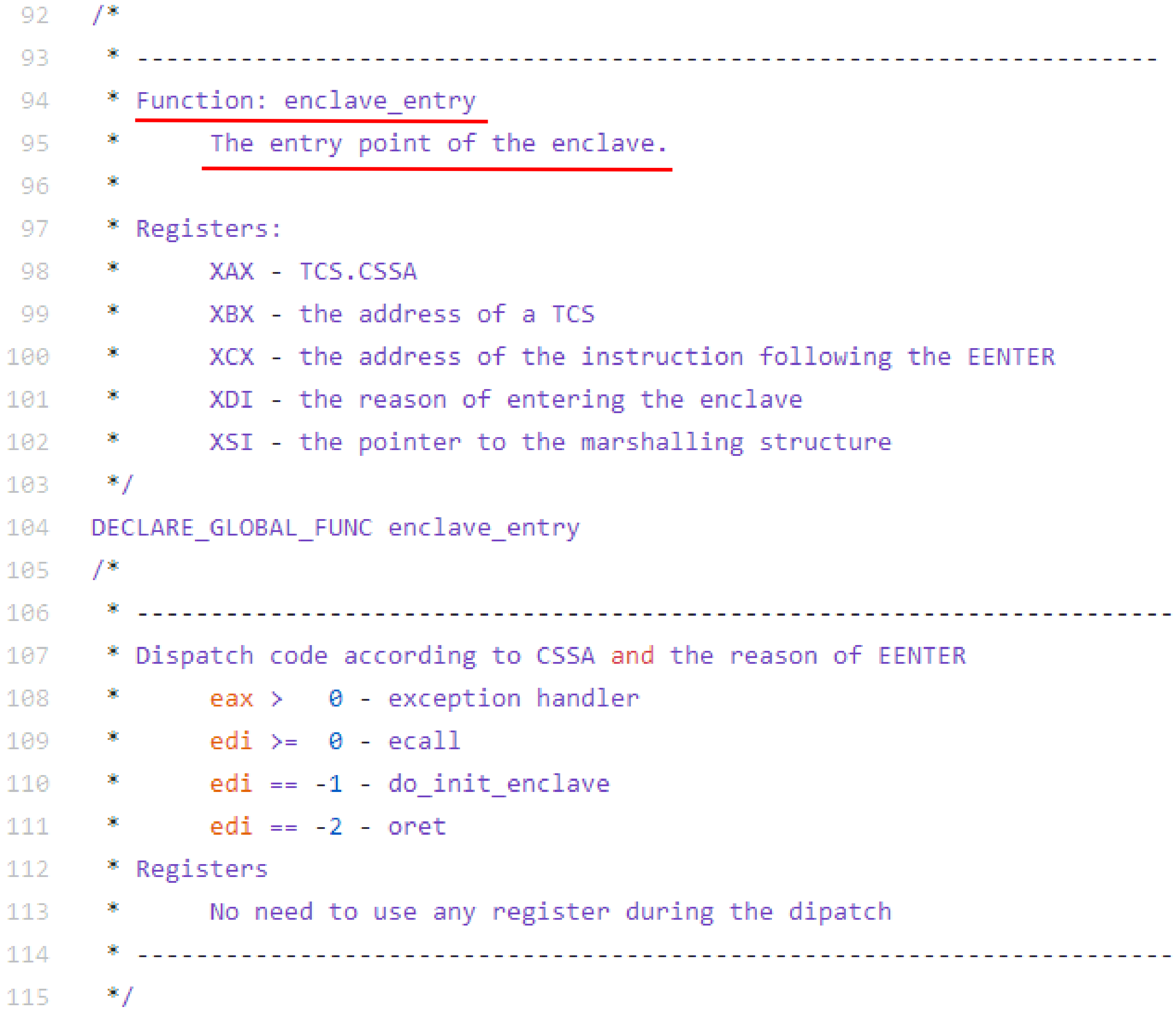
Background on SGX

- Life cycle



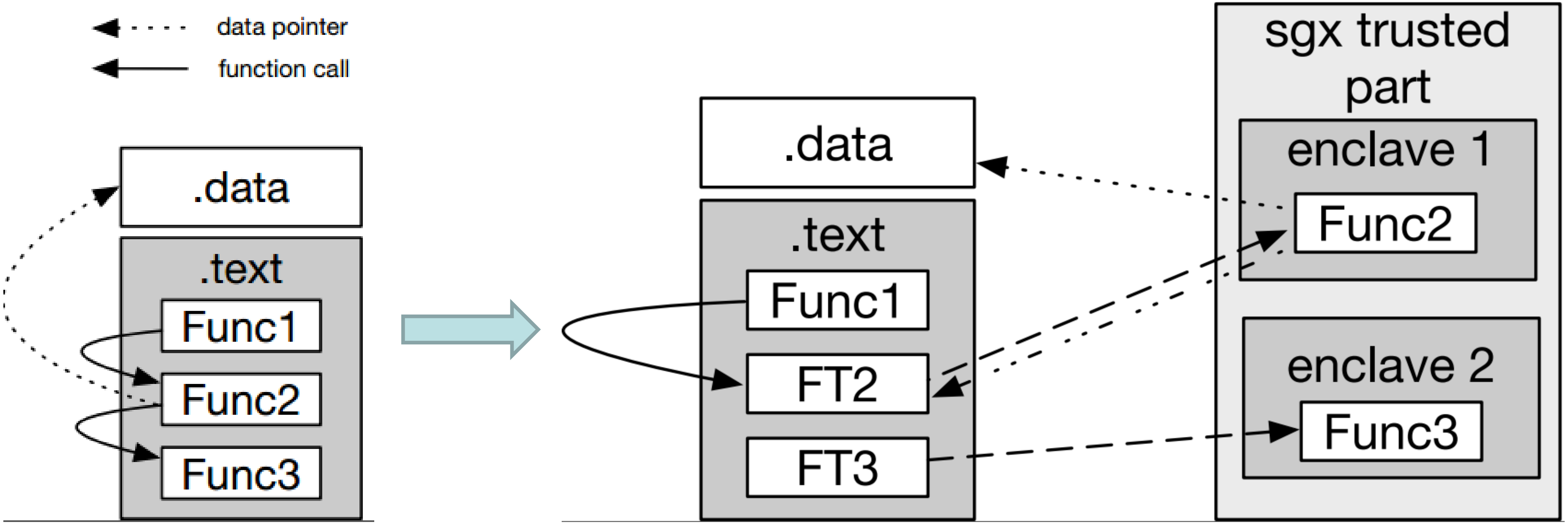
Background on SGX

- Controlled enclave entry
- Separated stack
- CPU state and registers are cleared if exceptions occur inside the enclaves.

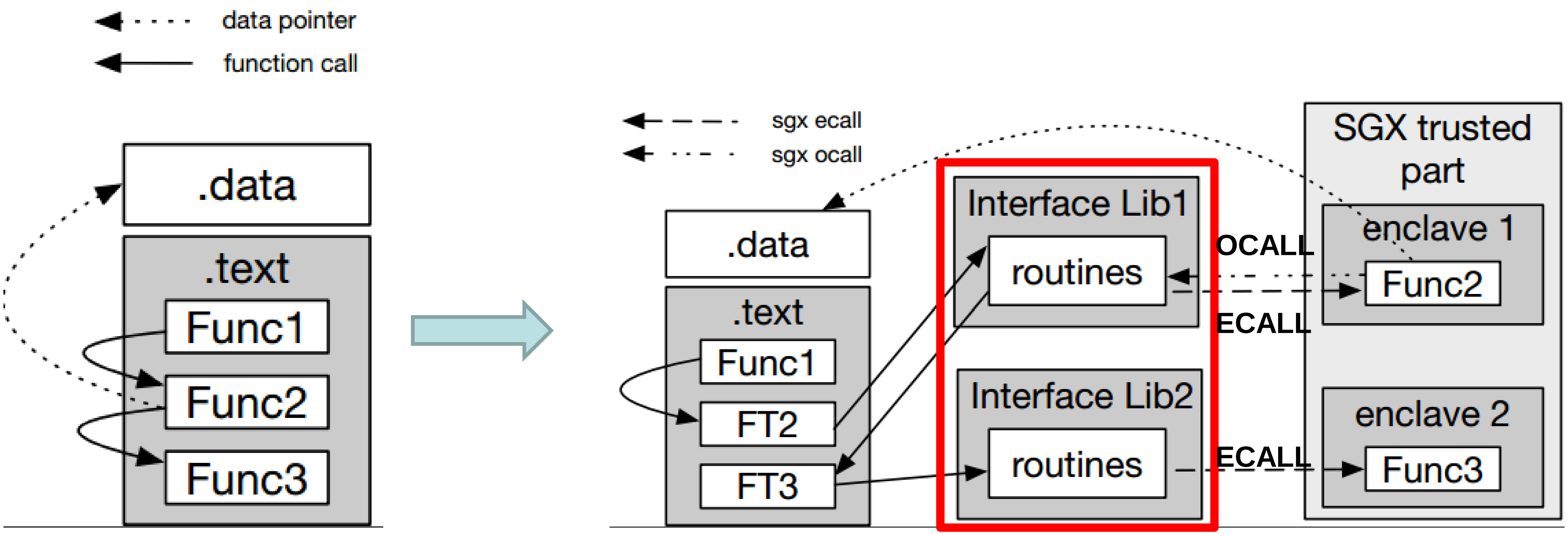


```
/*
 * ----------------------------------------------------------------------
 * Function: enclave_entry
 *      The entry point of the enclave.
 *
 * Registers:
 *      XAX - TCS.CSSA
 *      XBX - the address of a TCS
 *      XCX - the address of the instruction following the EENTER
 *      XDI - the reason of entering the enclave
 *      XSI - the pointer to the marshalling structure
 */
DECLARE_GLOBAL_FUNC enclave_entry
/*
 * ----------------------------------------------------------------------
 * Dispatch code according to CSSA and the reason of EENTER
 *      eax >   0 - exception handler
 *      edi >=  0 - ecall
 *      edi == -1 - do_init_enclave
 *      edi == -2 - oret
 * Registers
 *      No need to use any register during the dipatch
 * ----------------------------------------------------------------------
 */
```

Methodology

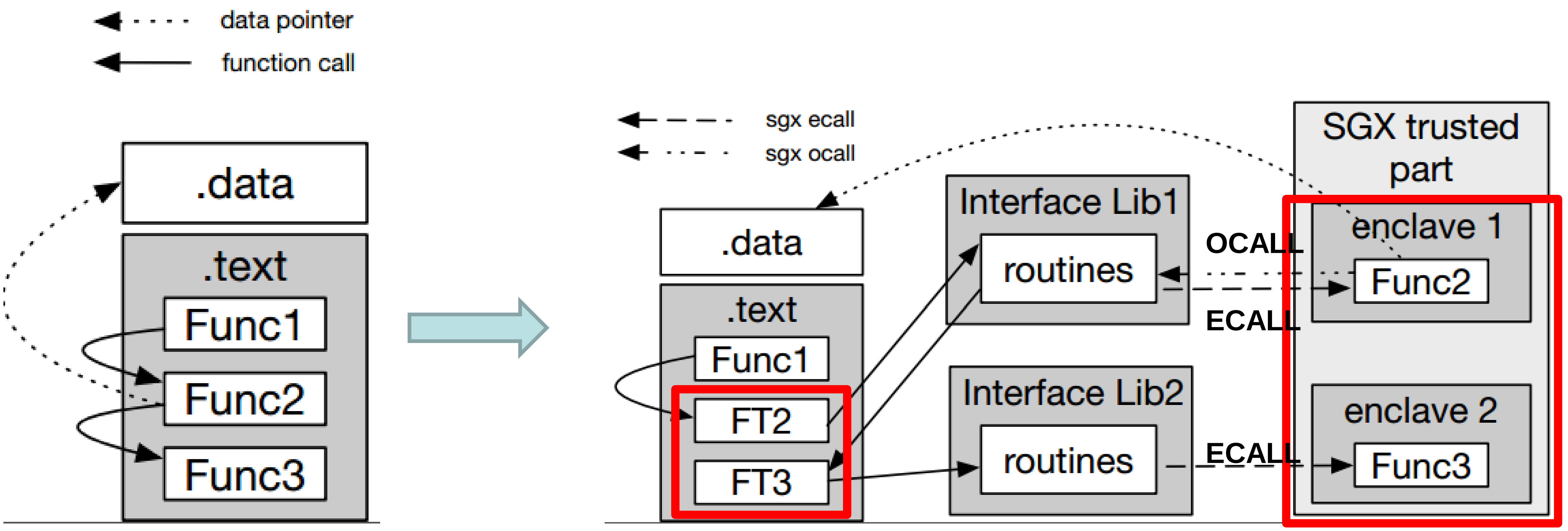


Methodology



Interface library: maintain routine code for **ecall** and **ocall**

Methodology



In-place binary editing: Trampoline code

Challenges

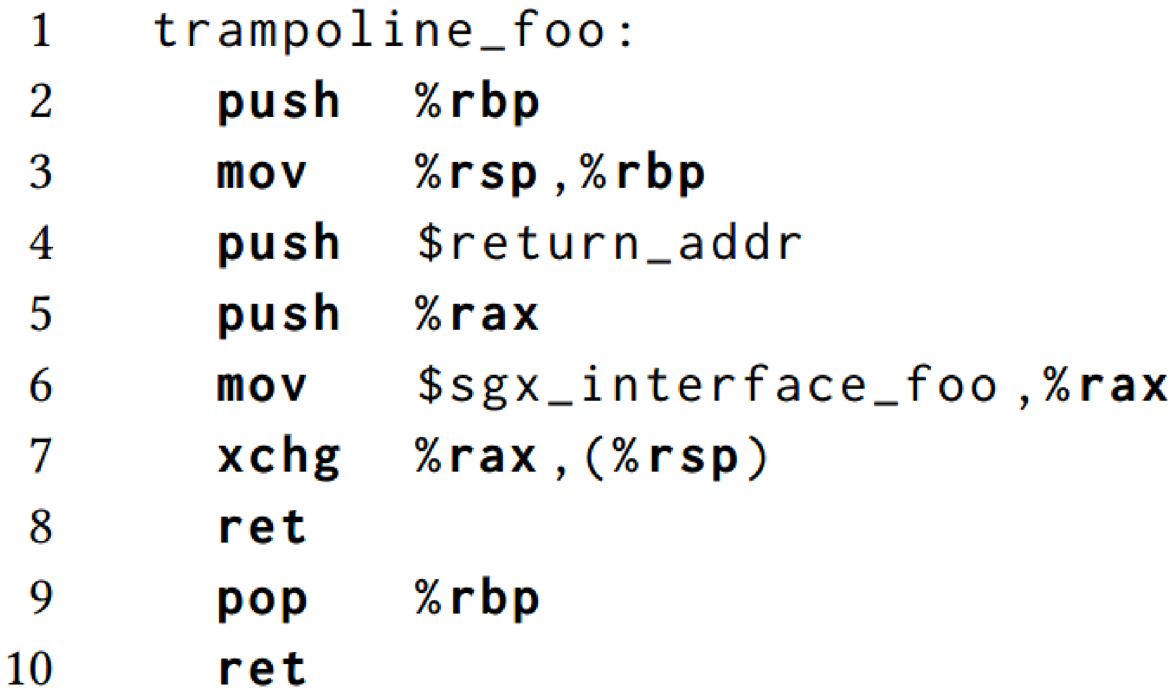
- Binary code reassembly disassembling
  - Uroboros
- How to generate enclave libraries
  - Intel SGX SDK
- Binary instrumentation to jump to the enclave entry
  - Trampoline code
- Exceptions
  - Customized exception handling inside the enclaves

Challenges

- Binary code reassembly disassembling
  - Uroboros
- How to generate enclave libraries
  - Intel SGX SDK
- Binary instrumentation to jump to the enclave entry
  - Trampoline code
- Exceptions
  - Customized exception handling inside the enclaves

Some technique details

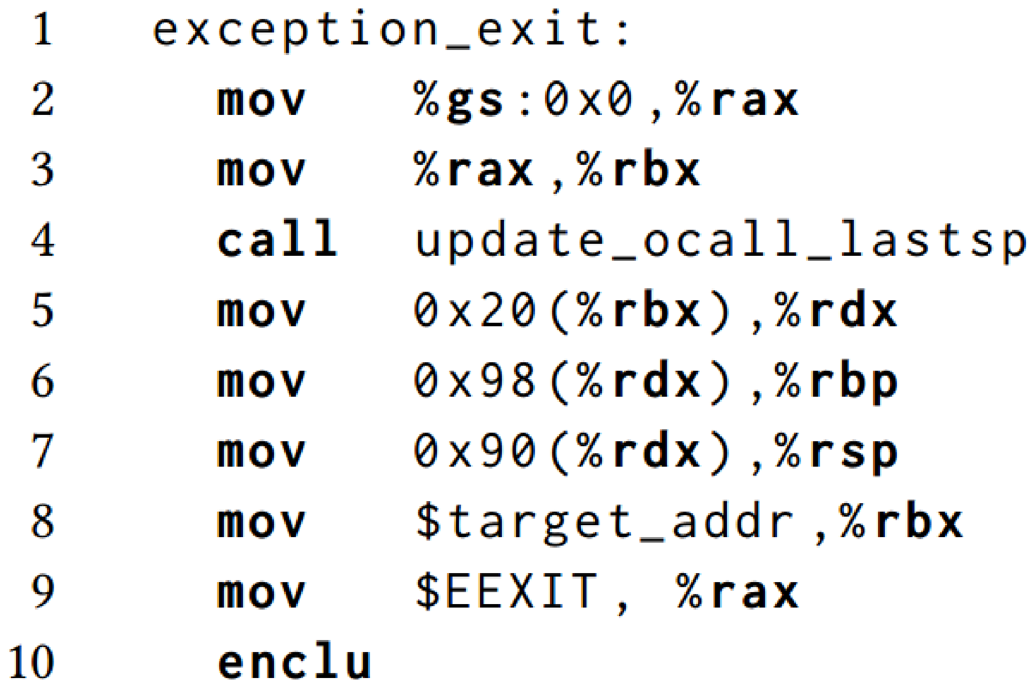
- In-place binary editing
  - Trampoline code



```
trampoline_foo:
  push   %rbp
  mov    %rsp,%rbp
  push   $return_addr
  push   %rax
  mov    $sgx_interface_foo,%rax
  xchg   %rax,(%rsp)
  ret
  pop    %rbp
  ret
```

Some technique details

- Exceptions
  - Customized exception handling inside the enclaves



```
exception_exit:
  mov    %gs:0x0,%rax
  mov    %rax,%rbx
  call   update_ocall_lastsp
  mov    0x20(%rbx),%rdx
  mov    0x98(%rdx),%rbp
  mov    0x90(%rdx),%rsp
  mov    $target_addr,%rbx
  mov    $EEXIT, %rax
  enclu
```

Proof-of-concept implementation

- Extend Uroboros with SGX instrumentation functionalities.
  - Employ the core functionality of Uroboros to identify program relocation symbols (e.g., code pointers).
  - Use industrial standard reverse engineering tool (IDA-Pro) to recover the function type information.
- Implement the instrumentation functionality in Scala, with over 1,700 LOC.
- The proof-of-concept implementation of the exception handling mechanism adds 56 lines of C code.

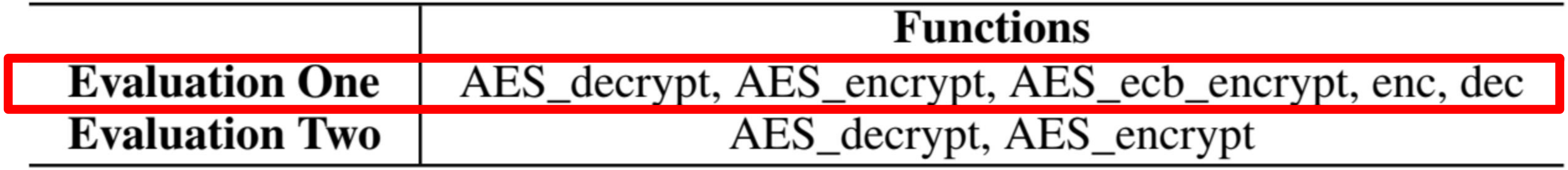
Evaluation

- Evaluations mainly focus on understanding the **feasibility** and **cost** of the instrumentation products.
- Two major factors would contribute to the performance penalty of the SGX protected code:
  - Execution slowdown of code components inside enclaves.
  - Cross-enclave control flow transfers, e.g., enclave ECALL.

Evaluation Setup

- Our preliminary evaluation instruments **sensitive procedures** provided by **cryptographic libraries**.
- AES implementation in OpenSSL (version 0.9.7)
  - Write sample code to trigger the **encryption** and **decryption** functions in the library.
  - key length is set as 256.
  - AES electronic codebook (ECB) mode.

Evaluation Setup

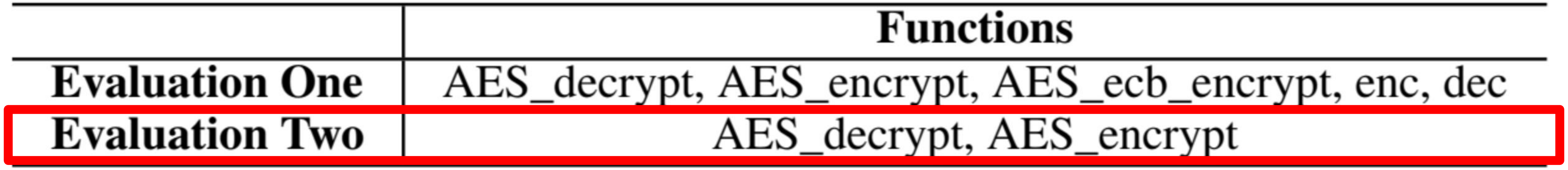


	Functions
Evaluation One	AES_decrypt, AES_encrypt, AES_ecb_encrypt, enc, dec
Evaluation Two	AES_decrypt, AES_encrypt

To measure the performance cost of code within enclave (first factor):

- All encryption/decryption computations are performed within one enclave.
- Pointers on key and data blocks are passed in through the interface.

Evaluation Setup

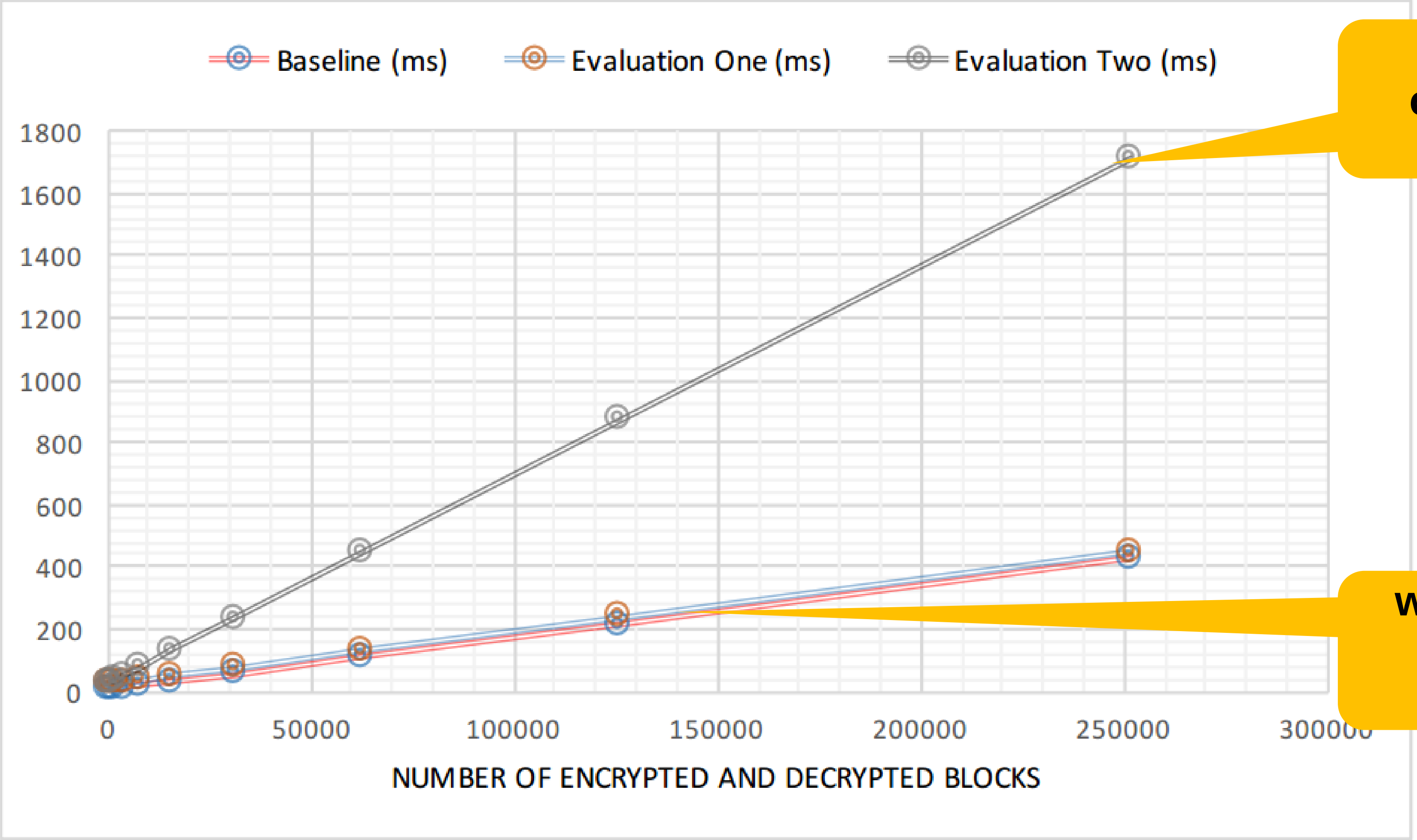


	Functions
Evaluation One	AES_decrypt, AES_encrypt, AES_ecb_encrypt, enc, dec
Evaluation Two	AES_decrypt, AES_encrypt

To measure the impact of inter-enclave control flow transfers (second factor):

- Put the block-level encryption/decryption functions into the enclave.
- Control the number of inter-enclave control transfers by changing the length of the input data.

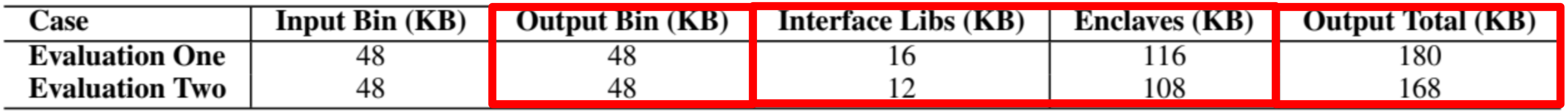
Evaluation Results



4× overhead over computation without SGX

when processing over 100k data blocks, overhead is 6.91%.

Evaluation Results



Case	Input Bin (KB)	Output Bin (KB)	Interface Libs (KB)	Enclaves (KB)	Output Total (KB)
Evaluation One	48	48	16	116	180
Evaluation Two	48	48	12	108	168

We measure the size increase in terms of multiple components:

- Size of output binary is identical with the input, since we perform in-place binary instrumentation.
- Both SDK routines and our routine code introduce size increase.
- The overall size increase is within a reasonable extent.
  - **Evaluation One** has three more functions than **Evaluation Two**.

Future works

Limitations

- How to reliably recover the function prototype?
- How to deal with the shared variables among several isolated enclaves?
- Some instructions/operations may not be supported inside the enclaves.
- …

Thanks!

Contact: ww31@indiana.edu