SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX

Yuan Chen[1], Jiaqi Li[1], Guorui Xu[1], Yajin Zhou[1], Zhi Wang[2], Cong Wang[3], Kui Ren[1]

[1]Zhejiang University, [2]Florida State University, [3]City University of Hong Kong

USENIX Security 2022

Motivation

- SGX: a popular TEE solution

Blockchain

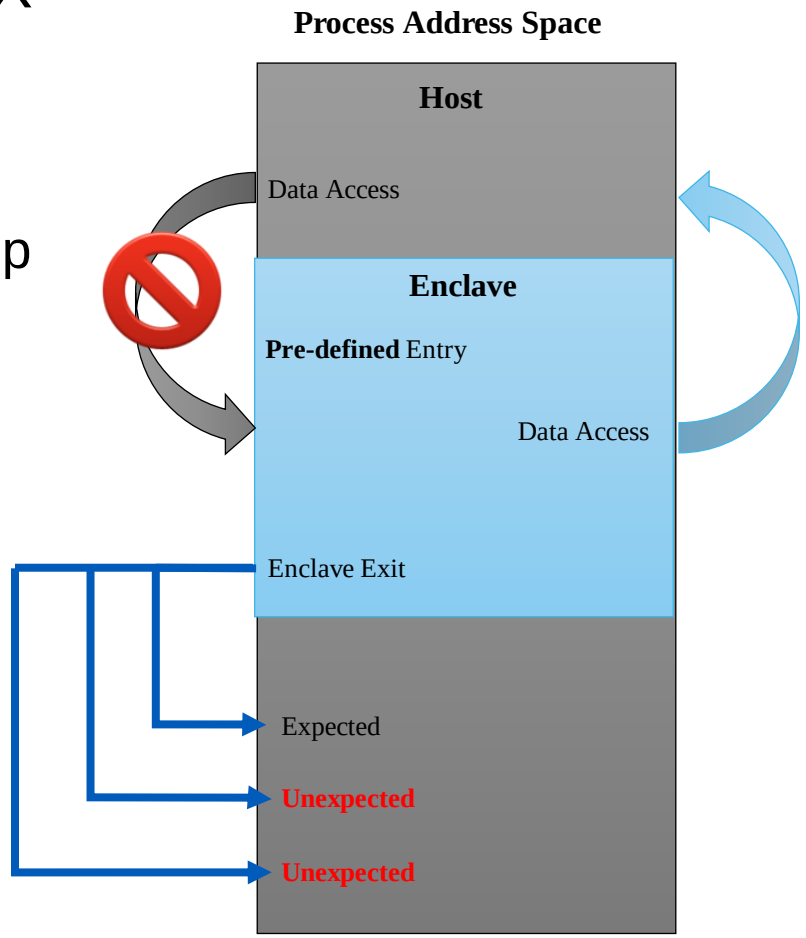
Finance

Health Care

- Application Components with SGX
  - Enclave: Sensitive code/data
  - Host application： Main application logic
- Problematic Assumption of SGX
  - Enclave is considered as trusted, while the host app is untrusted
  - In reality, enclave and host app are **mutual** untrusted
    - e.g., third-party enclaves, enclaves with flaw

Enclave-Host Asymmetry

- Introduced by the problematic assumption of SGX
  - Blind trust of the host app to the enclave
- Control Flow Asymmetry
  - Enclave can jump to **arbitrary** locations of the host app
  - The host app enters enclave via **pre-defined** entry
- Data Access Asymmetry
  - Enclave can access host memory
  - Not vice verse



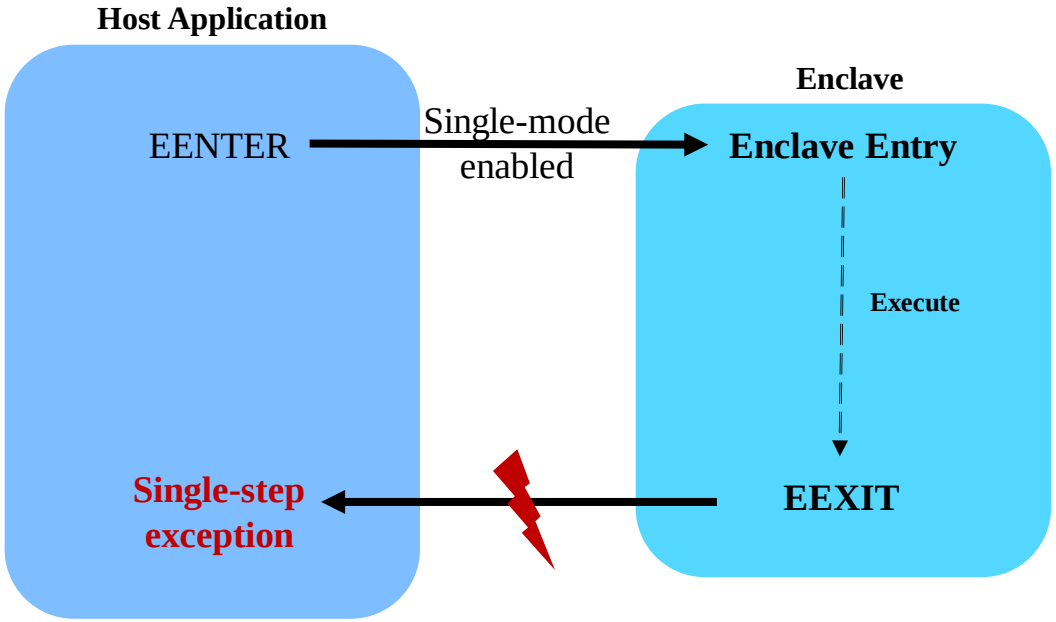
Our solution: SGXLock

- **Goal**: Eliminate enclave-host asymmetry and establish mutual distrust

- Control flow asymmetry elimination
  - Leverage single-step mode
- Data access asymmetry elimination
  - Leverage Intel MPK

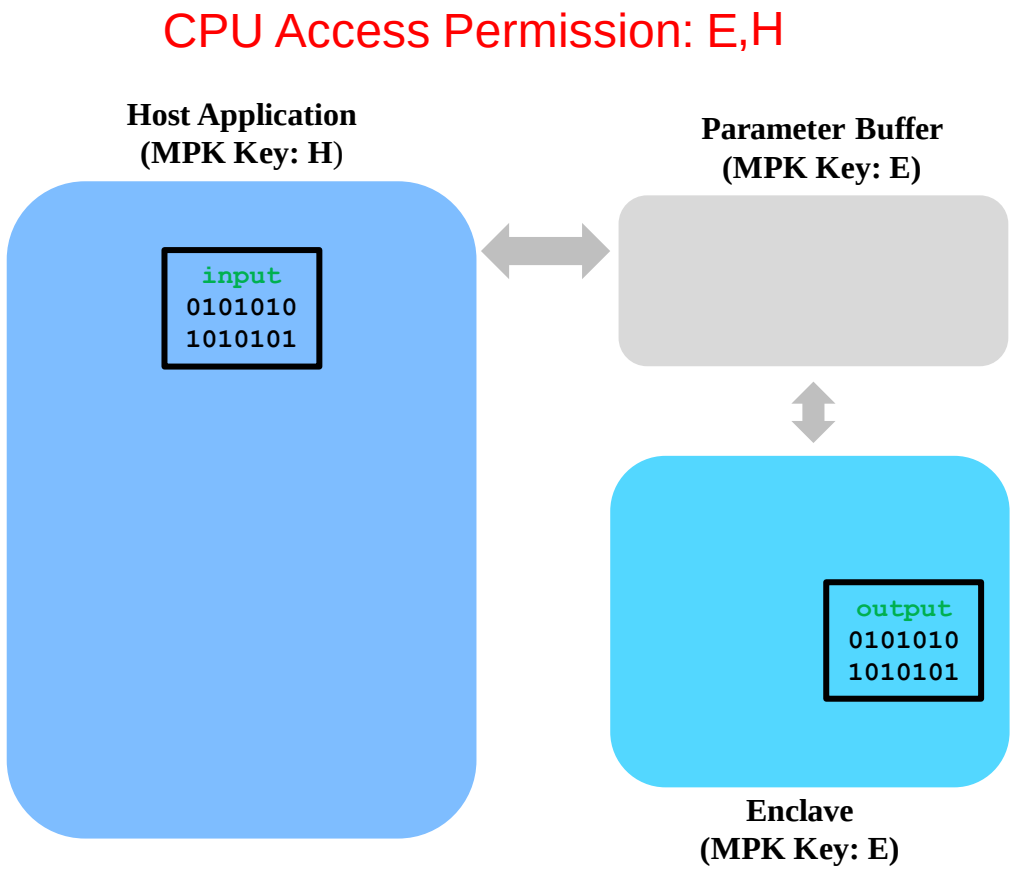
Control Flow Asymmetry Elimination

- Based on single-step mode
  - The execution inside enclave is treated as a single instruction for single-step mode
- Workflow
  - **Enter** enclave with single-step mode enabled
  - Execute inside enclave
  - When **EEXIT**, single-step exception triggered



Data Access Asymmetry Elimination

- Based on Intel MPK
- Initialize
  - Allocate different MPK keys (i.e., E, H) for the enclave and the host app
  - Allocate the parameter buffer for data interaction, assigned with MPK key E
- Before entering enclave
  - Copy input into parameter buffer
  - CPU access permission: {E,H} -> {E}
- After exiting enclave
  - CPU access permission: {E} -> {E,H}
  - Copy output from parameter buffer



Challenges

- C1. PKRU register update inside the enclave
  - PKRU represents the CPU’s access permission to MPK keys
  - Two ways to update PKRU inside the enclave
    - XSTORE instruction
    - WRPKRU instruction
- C2. Host stack pointer manipulation
  - Enclave can manipulate host stack pointer

Solution of Challenges

- For C1.
  - XRSTORE: PKRU is restored from mem as processor's extended state
    - Solution: disable the bit 9 of enclave's XFRM field
  - WRPKRU: Update PKRU directly
    - Solution: binary inspection to avoid the occurrence of WRPKRU inside the enclave

	What to inspect	**When to inspect**	**Who to inspect**
Static inspection	Plain enclave code	At enclave creation	Inspection code outside enclave
Dynamic inspection	Dynamic enclave code	At enclave runtime, triggered by W $\oplus$ X violation	Embedded inspection code

- For C2.
  - Host stack integrity check based on a secret key

Experimental Setup

- Implementation based on Intel SGX SDK v2.9.1 for Linux
- Ubuntu 18.04.4 (Kernel v5.4.28) with SGX driver v2.6 installed
- Intel i7-10700F CPU (2.90GHz), which supports SGX and MPK

Micro-Benchmarks

- Raw ECALL/OCALL latency
  - Raw means no workload for ECALL/OCALL

	Original (cycle)	SGXLock (cycle)
ECALL	7636	11662 (52.7%)
OCALL	5908	9588 (62.3%)

SGXLock introduces relatively high latency overhead for host-enclave interaction

Macro-Benchmarks

- Three representative scenarios
  - ML inference service, Database operation, HTTP web server

	Overhead	OCALL
ML inference service	0.84%	
Database operation	1.26%	~ 13k OCALLs/s
HTTP web server	3.98%	~ 30k OCALLs/s

SGXLock is efficient in the above real-world scenarios, even with high-frequency OCALLs

Conclusion & Takeaway

- Blind trust of the host app to the enclave introduces enclave-host asymmetry
- SGXLock: a defense solution to confine an untrusted enclave’s behavior
- Evaluation from real-world scenarios shows the efficiency of SGXLock

Thank you for listening!

Questions?

Yuan Chen

yuanchen96@zju.edu.cn