Memory Corruption Attacks in the Context of Trusted Execution Environments

Lucas Davi
Secure Software Systems
University of Duisburg-Essen, Germany
Why Hardware-Assisted Application Security?

- Exploits: bugs in applications
- CPU Bugs: e.g., Spectre
- Malware
- Malware Infection: e.g., Zeus banking trojan
- Operating System
  - Kernel Exploits: e.g., Stagefright
  - DRAM Bugs: e.g., Rowhammer
- Hardware
  - CPU
  - Memory
Hardware-Assisted Security Enables Implementation of Trusted Execution Environments (TEEs)

- **Normal World**
  - App
  - App
  - Operating System

- **Secure World**
  - Trusted App
  - Trusted App
  - Trusted Operating System

Popular TEE Implementations:
- ARM TrustZone
- Intel Software Guard Extensions (SGX)
Principle of Remote Attestation

• **Goal**: Check if prover is **now** in a trustworthy state

Attestation Protocol

Verifier

Measure software state

Prover

Measurement Database

Verify Report

Challenge

Authentic Report

Attestation Protocol
History of Remote Attestation

TPM Attestation 2001

Pioneer [SOSP’05]...
SWATT [SP’04]

Software-based Attestation 2004

Dynamic Root of Trust 2005

PUF-based Attestation 2011

minimal Trust Anchors 2010

Property-based Attestation [NSPW’04]

Behavior-based Trust [NSPW’04]

Semantic Remote Attest. [VM’04]

AMD SVM

Intel TXT

Intel SGX

Flicker [Eurosys’08]...

POSE [ESORICS’10]

SMART [NDSS’12,DATE’14]

TrustLite [Eurosys’14]

SEDA [CCS’15]

SANA [CCS’16]

DARPA [WiSEC’16]

Lightweight PUF Attest. [WiSEC’11]
Key Limitation: current binary attestation schemes do not address run-time (memory corruption) attacks
CONTROL-FLOW ATTESTATION

Embedded System with ARM TrustZone

RUN-TIME ATTACKS AGAINST INTEL SGX

Intel SGX

TEE BUG FINDING
Problem Space of Run-time Attacks

Control-Flow Attack
[Shacham, ACM CCS 2007]
[Schuster et al., IEEE S&P 2015]

Non-Control-Data Attack
[Chen et al., USENIX Sec. 2005]
[Carlini et al., USENIX Sec. 2015]
Not suitable for control-flow attestation

- Integrity-based schemes usually target a specific runtime attack class
- These schemes only output whether an attack occurred but don’t attest the control-flow path
C-FLAT
[Abera et al., CCS 2016]

Verifier

Control-Flow Graph (CFG) Analysis

Path Measurement

Control-Flow Validation

Run-Time Path Measurement

App A

Path Measurement is performed inside a TEE (TrustZone)
How to attest the executed control flows without transmitting all executed branches?
C-FLAT Measurement Function

Cumulative Hash Value: \( H_i = H( H_{i-1}, N ) \)

- \( H_{i-1} \) - previous hash result
- \( N \) - instruction block (node) just executed

\[ H_1 = H(0, A) \]
\[ H_2 = H(H_1, B) \]
\[ H_3 = H(H_2, C) \]
\[ H_4 = H(H_2, D) \]
\[ H_5 = H(H_2, E) \]
\[ H_6 = H(H_5, F) \]
Loops are a challenge!

Different loop paths and loop iterations lead to many valid hash values
C-FLAT: Loop Handling

```
while (cond.) {...}
```

```
if (cond.) {...}
```
C-FLAT: Loop Handling

\[ H_6b = H(H_5, F) \]
\[ H_6a = H(H_4, F) \]
\[ H_3 = H(H_2a, C) \]
\[ H_4 = H(H_3, D) \]
\[ H_5 = H(H_3, E) \]
\[ H_1 = H(0, A) \]
\[ H_{2a} = H(H_1, B) \]
\[ H_{2b} = H(0, B) \]
\[ H_3 = H(H_2a, C) \]
\[ H_5 = H(H_3, E) \]

Loop Entry Hash
Loop Hash, Iteration

\[ H_7 = H(H_{2b}, G) \]
Prototype Architecture

- Implementation on Raspberry Pi 2

- Application Binary
- Trampolines

- Measurement Engine and Attestation

Hardware

Normal World

Secure World (ARM TrustZone)
Evaluation: Syringe Pump

- Original implementation targets Arduino boards
- We ported the code to Raspberry Pi
- 13,000 instructions with 332 CFG edges of which 20 are loops
- Main functions are set-quantity and move-syringe

Source: https://hackaday.io/project/1838-open-syringe-pump
Applying C-FLAT to Syringe Pump

**main()**

```c
while (1) {
    if (serialReady()) {
        cfa_init;
        processSerial();
        cfa_quote;
    }
}
```

**processSerial()**

```c
if (input == '+') {
    action(PUSH, bolus);
    updateScreen();
} else if (input == '-') {
    action(PULL, bolus);
    updateScreen();
}
```

**action(direction, bolus)**

```c
steps = bolus * steps_per_mL
if (direction == PUSH) {
    /* set stepper direction */
} else {
    /* PULL */
    /* set stepper direction */
}
for (steps) {
    /* move stepper */
}
```

**bolus** = dose of drug; volume of cylinder for a particular height

Please note that this slide shows a simplified view of the Syringe pump code and control-flow graph.
Final Hash Measurements

action(direction, bolus)

4 steps = bolus * steps_per_mL
5 if (direction = PUSH) {
6    /* set stepper direction */
7   } else /* PULL */
8   /* set stepper direction */
9 for (steps) {
10  /* move stepper */
Open Questions

- How to address performance overhead?
  - Tackled based on hardware assistance in a follow-up work, LO-FAT [DAC‘17]
- What can go wrong inside the TEE?
  - Next part of this talk with focus on SGX
Overview on Intel SGX

APP
- App Code
- App Data
- Enclave

Malware
- App Code
- App Data

APP
- App Code
- App Data
- Enclave

Operating System
- BUG

Hardware
- SGX
- CPU
Entry to Enclave code is only allowed at pre-defined entry points.
Academic Research on Side-Channel Attacks Against SGX

Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems

Weihong Cai, Microsoft Research weihong.cai@outlook.com
Marcus Peinado, Microsoft Research marcusp@microsoft.com

Boffins show Intel's SGX can leak crypto keys
Software Guard Extensions are supposed to hide data. But the 'Prime-Probe attack' fixes that

Software Grand Exposure: SGX Cache Attacks Are Practical
Ferdinand Brassier1, Ute Müller2, Alexandra Distinenko2, Kari Kostiainen2, Sriljan Cupkuk2, and
Aboud-Reza Saleghi1

CacheZoom: How SGX Amplifies The Power of Cache Attacks
Alunad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth

Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing
Sanjeev Lee, Ming-Wei Shih, Pranav Gera, Taeja Kim, and Hyeseon Kim,
Georgia Institute of Technology; Marcus Peinado, Microsoft Research

Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution
Jo Van Bulcke, imec-DistriNet, KU Leuven; Nico Weichert and Rüdiger Kapitza, IBB DS, TU Braunschweig; Frank Piessens and Raoul Strackx, imec-DistriNet, KU Leuven

OFEASHELD: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution
Jo Van Bulcke, imec-DistriNet, KU Leuven; Marina Minkin, Technion; Ofir Weisse,
Daniel Genn, and Baris Kaskici, University of Michigan; Frank Piessens, imec-DistriNet,
KU Leuven; Mark Silberstein, Technion; Thomas F. Wenisch, University of Michigan;
Yuvah Yarom, University of Adelaide and Data61; Raoul Strackx, imec-DistriNet, KU Leuven

SGXPECTRE Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution
Guoying Chen, Sandman Chen, Yuan Xiao, Yingtian Zhang, Zhiquan Lui, and H. Lu
Department of Computer Science and Engineering
What about Return-Oriented Programming Attacks?
Return-Oriented Programming
Return-Oriented Programming Attack

Program Stack

Corrupt Control Structures

Program Code

Sequence 1
- x86_ins
- ret

Sequence 2
- pop eax
- pop ebx
- ret

Sequence 3
- x86_ins
- ret

EAX: 0x80102030

EBX: 0xAABBCDD
First Run-Time Attacks and Defenses
Targeting Intel SGX
Related Work

**Dark ROP** [USENIX Sec. 2017]

- Analyzes the threat of memory corruption vulnerabilities in the context of SGX
- Presents ROP attack against (unknown) encrypted enclave binaries
- Based on probing attacks
- Requires kernel privileges and ability to repeatedly crash the enclave

**SGX-Shield** [NDSS 2017]

- Enforces fine-grained memory randomization of SGX enclave
- Software-based data execution prevention (DEP)
- Proposes control-flow integrity for return instructions
Can we bypass memory randomization in SGX?
Our main observation is that the Intel SGX SDK includes dangerous return-oriented programming gadgets which are essential for app-enclave communication
ECALL: Call into an enclave

App Code

App Data

ECALL

Untrusted Runtime System (uRTS)

Trusted Runtime System (tRTS)

Enclave Code

Function 0
Function 1
Function 2
Function 3

Enclave Stack

Enclave
OCALL: Enclave Call to the Host Application

APP

App Code

App Data

Untrusted Runtime System (uRTS)

Enclave Code

Function 0

Function 1

Function 2

Function 3

Enclave Stack

OCALL Frame

Register State

Trusted Runtime System (tRTS)

OCALL
AEX: Asynchronous Enclave Exit (Exception)

APP

- App Code
- App Data

Enclave

- Enclave Code
  - Function 0
  - Function 1
  - Function 2
  - Function 3

- Enclave Stack
  - Exception information structure
  - Register State

Trusted Runtime System (tRTS)

Untrusted Runtime System (uRTS)

Operating System
Restoring State is Critical

• When OCALL returns, the register state is restored by the tRTS function `asm_oreset()`

• If an attacker manages to inject a fake ocall frame, he controls the subsequent state

• After handling the exception, the register state is restored by the tRTS function `continue_execution()`

• If an attacker manages to inject a fake exception structure, he controls the subsequent state
Basic Attack Idea

APP
- App Code
- App Data

Untrusted Runtime System (uRTS)

Trusted Runtime System (tRTS)

Enclave Code
- Function 0
- Function 1
- Function 2
- Function 3

Enclave Stack
- Counterfeit State
- Mal. Register State

Counterfeit State Information
Two Attack Primitives

- Primitive to exploit OCALL mechanism
- It is based on injecting fake OCALL frames
- Prerequisites: stack control

- Primitive to exploit asynchronous exception handling in SGX
- Based on injecting fake exception structures
- Prerequisites: function pointer overwrite and control of rdi register
Attack Workflow for Stealing SGX-Protected Keys

Counterfeit State Information

Enclave Code
- Function 0
- get_key
- send_file
- Function 3

Enclave Stack
- Counterfeit State Fake OCALL Frames Except. Structures

APP
- App Code
- App Data

Untrusted Runtime System (uRTS)
- rip
- rsp
- rdi
- all_other_regs

Trusted Runtime System (tRTS)
- ORET Primitive
- CONT Primitive
Attack Workflow for Stealing SGX-Protected Keys

- **APP**
  - App Code
  - App Data

- **Enclave**
  - Enclave Code
    - Function 0
    - get_key
    - send_file
    - Function 3

- **Enclave Stack**
  - Counterfeit State
    - Fake OCALL Frames
    - Except. Structures

- **Trusted Runtime System (tRTS)**
  - ORET Primitive
  - CONT Primitive

- **Untrusted Runtime System (uRTS)**
  - rip
  - rsp
  - rdi
  - all_other_regs
Attack Workflow for Stealing SGX-Protected Keys

Counterfeit State Information

Enclave Code
- Function 0
  - get_key
  - send_file
  - Function 3

Enclave Stack
- Counterfeit State
- Fake OCALL Frames
- Except. Structures

App Code

Enclave Stack
- rip
- rsp
- rdi
- all_other_regs

Trusted Runtime System (tRTS)
- ORET Primitive
- CONT Primitive

Untrusted Runtime System (uRTS)
Attack Workflow for Stealing SGX-Protected Keys

- **Enclave Code**
  - Function 0
  - get_key
  - send_file
  - Function 3

- **Enclave Stack**
  - Counterfeit State
  - Fake OCALL Frames Except. Structures

- **Trust Runtime System (tRTS)**
  - ORET Primitive
  - CONT Primitive

- **Untrusted Runtime System (uRTS)**
  - rip
  - rsp
  - rdi
  - all_other_regs

- **APP**
  - App Code
  - App Data

- **Counterfeit State Information**
However, this attack doesn’t work if SGX-Shield randomizes the SGX address space
Revisited Attack to Bypass SGX-Shield

APP

- App Code
- App Data

Untrusted Runtime System (uRTS)

Trusted Runtime System (tRTS)

Enclave Code
- Function 0
- get_key
- send_file
- Function 3

Enclave Stack
- Shellcode
- Stealing Keys

Counterfeit State
Information

Except. Structures

Memory Write

CONT Primitive

ORET Primitive

Counterfeit State
Fake OCALL Frames

Counterfeit State
Information

Stealing Keys

Shellcode

get_key

send_file

Function 0

Function 3
Possible Defenses

• Removing SDK from enclave memory?
  • Not feasible as OCALL, ECALL, AEX require the tRTS

• Randomizing SDK code?
  • Challenging, the tRTS is accessed through fixed entry points

• Discovering vulnerabilities beforehand?
  • Last part of this talk: research on fuzzing and symbolic execution
Background: Bug Discovery Techniques

- **Symbolic Execution**
  - Emulate the program based on encoding the program state as symbolic variables
  - Utilize solver to find feasible crashing paths

- **Fuzzing**
  - Probabilistically explore program paths
  - Find new inputs with random mutation

---

Symbolic Input → Symbolically Execute Program → Program Path + State → Crash?

- No → SMT Solver
- Yes → Generate Crashing Input

Mutation → Run Program → Feedback (coverage) → Crash?

- No → Run Program
- Yes → Report Crash
TrustZone OS Fuzzing

Fuzzer

Initial Testcases

Feedback (Coverage, Crash Info)

Emulator

Normal World

Input

Agent

Linux Kernel

Emulated Hardware

Emulated Secure Monitor

Checkpoint & Restore

Trusted Kernel

“Trusted” Agent

Syscall

Record Coverage and Crash Information

Inputs

Inputs

TrustZone OS

Input

Fuzzing

Record Coverage and Crash Information

Inputs
Symbolic Execution of SGX Enclaves

- enclave.so
- State Checker
- Crash Report
- Unconstrained State
- Loader
- Explorer
- Symbolic Execution Engine
- Emulated Address Space
  - Enclave Code
  - Enclave Data
  - Enclave Stack

Our Tool: angr
Harware-assisted application security is vital to implement trustworthy systems and enhanced security services → **control-flow attestation**

However, we need to make sure that an attacker cannot exploit bugs inside the TEE → **return-oriented programming**

Hence, research on bug finding in TEE code is crucial → **fuzzing, symbolic execution**