Our Research Overview in Software and System Security

Advanced Attacks
- Return-oriented programming without returns [ACM CCS 2010]
- JIT-ROP [IEEE S&P 2013]
  - Best Student Paper
- Stitching Gadgets [USENIX Sec. 2014]
- COOP [IEEE S&P 2015]

Advanced Defenses
- Mobile CFI [NDSS 2012]
- XIFER [AsiaCCS 2013]
- HAFIX [DAC 2015]
  - Best Paper
- ROP Filter [RAID 2014]
- Isomeron [NDSS 2015]
- Readactor(++) [IEEE S&P 2015, ACM CCS 2015]

Mobile Security
- Privilege Escalation on Android [ISC 2010]
- XManDroid [NDSS 2012]
- PSiOS [AsiaCCS 2013]
  - Distinguished Paper
- XiOS [AsiaCCS 2015]
  - Best Paper
- LR2 [NDSS 2016]
Overview of this Talk

- Background/Evolution of Code-Reuse Attacks
- Building Code Randomization Defenses
- Building Control-Flow Integrity Defenses
- Remote Attestation of Embedded Systems
Motivating Problem

- Software increasingly sophisticated and complex
- Various developers involved
- Native Code
- Many program bugs

Large attack surface for software exploits

100 Mio LOC in premium vehicles [DAC’15 Keynote]
Main Attack Techniques:
Code Injection and Code Reuse
General Principle of Code Injection Attacks

Control-Flow Graph (CFG):
- A
- B
- C

Basic Block (BBL) A:
- ENTRY asm_ins, ...
- EXIT

Basic Block (BBL) B:
- ENTRY asm_ins, ...
- EXIT

1. Buffer overflow
2. Code Injection
3. Control-flow deviation

Data flows
Program flows
General Principle of Code-Reuse Attacks

Control-Flow Graph (CFG)

ENTRY
asm_ins, ...
EXIT

Basic Block (BBL) A

ENTRY
asm_ins, ...
EXIT

1 Buffer overflow

2 Control-flow deviation

BBL B

ENTRY
asm_ins, ...
EXIT

Data flows
Program flows
Code Injection vs. Code Reuse

• Code Injection – *Adding a new node to the CFG*
  • Adversary can execute arbitrary malicious code
    • open a remote console (classical shellcode)
    • exploit further vulnerabilities in the OS kernel to install a virus or a backdoor

• Code Reuse – *Adding a new path to the CFG*
  • Adversary is limited to the code nodes that are available in the CFG
  • Requires reverse-engineering and static analysis of the code base of a program
Code injection is more powerful; so why are attacks today typically using code reuse?
Data Execution Prevention (DEP)

- Prevent execution from a writeable memory (data) area
Data Execution Prevention (DEP) cntd.

- Implementations
  - Modern OSes enable DEP by default (Windows, Linux, iOS, Android, Mac OSX)
  - Intel, AMD, and ARM feature a special No-Execute bit to facilitate deployment of DEP

- Side Note
  - There are other notions referring to the same principle
    - $W \oplus X$ – Writeable XOR eXecutable
    - Non-executable memory
Hybrid Exploits

- Today’s attacks combine code reuse with code injection

![Diagram showing code memory structures and functions.]
Hybrid Exploits

- Today’s attacks combine code reuse with code injection

![Diagram](image_url)

- CODE Memory I
  - Executable
  - `AllocateMemory()`
  - `CopyMemory()`
  - `ChangePermission()`

- DATA Memory
  - `Malicious Code`

- CODE Memory II (Libraries)
  - `readable and executable`

- DATA Memory
  - `readable and writeable`
  - `Malicious Code`
Hybrid Exploits

- Today’s attacks combine code reuse with code injection

[Diagram showing memory regions and functions related to code injection and execution]
Three Decades of Runtime Exploits

- **Morris Worm**
  - 1988

- **return-into-libc**
  - **Solar Designer**
  - 1997

- **Return-oriented programming**
  - **Shacham**
  - CCS 2007

- **Continuing Arms Race**

- **Code Injection**
  - **AlephOne**
  - 1996

- **Borrowed Code Chunk Exploitation**
  - **Krahmer**
  - 2005
Recent Attacks

**Stagefright** [Drake, BlackHat 2015]
These issues in Stagefright code critically expose 95% of Android devices, an estimated 950 million devices

**Cisco Router Exploit** [2016]
Million CISCO ASA Firewalls potentially vulnerable to attacks
Relevance and Impact

High Impact of Attacks

- Web browsers repeatedly exploited
- Zero-day issues exploited in Stuxnet/Duqu [Microsoft, BH 2012]
- iOS jailbreak

Industry Efforts on Defenses

- Microsoft Enhanced Mitigation Experience Toolkit (EMET)
- Microsoft Control Flow Guard (CFG) in Windows 10
- Google’s compiler extension VTV (virtual table verification)
- Intel Control-Flow Enforcement Technology (CET)

Hot Topic of Research

- A large body of recent literature on attacks and defenses
Overview on Return-Oriented Programming
The Big Picture

The New Yo

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous ProBlogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the bloggers who are improving their blogs. When asked about the success of his blog that relates to the

Return oriented Programming
Return-Oriented Programming (ROP)

[Shacham, CCS 2007]

1. Adversary can hijack control-flow (buffer overflow)
2. Adversary knows the memory layout (memory disclosure)
3. Adversary can construct gadgets
4. Adversary can write ROP payload in the data area (stack/heap)

Data Area
- ROP Payload
- Application
- Shared Libraries

Code Area
- Application Address Space
- MEMORY

Gadget Space
- (e.g., Shared Libraries)

MOV, ESP, LNOP, LOAD, ADD, CALL, XOR, STORE
ROP Attack Technique: Overview

Program Stack

- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2

Corrupt Control Structures

Program Code

Sequence 1
- `asm_ins`
- `POP {PC}

Sequence 2
- `POP REG1`
- `POP REG2`
- `POP {PC}

Sequence 3
- `asm_ins`
- `POP {PC}

REG1:

REG2:
ROP Attack Technique: Overview

Program Stack

- Return Address 1
- Return Address 2
- Return Address 3
- Value 1
- Value 2

Program Code

Sequence 1
- asm_ins
- POP {PC}

Sequence 2
- POP REG1
- POP REG2
- POP {PC}

Sequence 3
- asm_ins
- POP {PC}
ROP Attack Technique: Overview

Program Stack:
- Return Address 1
- Value 1
- Return Address 2
- Value 2
- Return Address 3

Program Code:
- Sequence 1
  - asm_ins
  - POP {PC}
- Sequence 2
  - POP REG1
  - POP REG2
  - POP {PC}
- Sequence 3
  - asm_ins
  - POP {PC}

SP

REG1:

REG2:

Value 1
ROP Attack Technique: Overview

Program Stack

Return Address 1
Value 1
Return Address 2
Value 2
Return Address 3

Program Code

Sequence 1
asm_ins
POP {PC}

Sequence 2
POP REG1
POP REG2
POP {PC}

Sequence 3
asm_ins
POP {PC}

REG1:
Value 1

REG2:
Value 2
ROP Attack Technique: Overview

Program Stack

Return Address 1
Value 1
Return Address 2
Value 2
Return Address 3

Program Code

Sequence 1
asm_ins
POP {PC}

Sequence 2
POP REG1
POP REG2
POP {PC}

Sequence 3
asm_ins
POP {PC}

REG1:
Value 1

REG2:
Value 2
Summary of Basic Idea

- Perform arbitrary computation with return-into-libc techniques

Approach

- Use **small instruction sequences** (e.g., of libc) instead of using whole functions
- Instruction sequences range from 2 to 5 instructions
- All sequences end with a `return (POP{PC})` instruction
- Instruction sequences are chained together to a **gadget**
- A gadget performs a particular task (e.g., load, store, xor, or branch)
- Afterwards, the adversary enforces his desired actions by combining the gadgets
Special Aspects of ROP
Code Base and Turing-Completeness

Application Code

Shared Libraries

MOV reg1, 0x1 → RET

MOV reg2, 0x2 → RET

ADD reg1, reg2 → RET

Static Analysis
Code Base and Turing-Completeness

Application Code

Shared Libraries

Static Analysis

GADGET SPACE

MOV
CALL
Uncond. JMP
LOGIC
Cond. JMP
STORE
LOAD
Arith.

Optional
Mandatory

Turing-complete language
Gadget Space on Different Architectures

Architectures with no memory alignment, e.g., Intel x86

Architectures with memory alignment, e.g., SPARC, ARM

Intended Code
mov $0x13,%eax
jmp 3aae9

Unintended Code
add %al,(%eax)
add %ch,%cl
ret
Stack Pivot
[Zovi, RSA Conference 2010]

- Stack pointer plays an important role
  - It operates as an instruction pointer in ROP attacks
- Challenge
  - In order to launch a ROP exploit based on a heap overflow, we need to set the stack pointer to point to the heap
  - This is achieved by a stack pivot
Stack Pivot in Detail

*REG1 is controlled by the adversary and holds beginning of ROP payload
Stack Pivot in Detail

*REG1 is controlled by the adversary and holds beginning of ROP payload
Stack Pivot in Detail

Stack

- TOP of Stack

Heap

- Return Address 3
- Return Address 2
- Return Address 1

Code

- label_pivot:
  - Stack Pivot
  - MOV SP, REG1*
  - POP {PC}

*REG1 is controlled by the adversary and holds beginning of ROP payload
ROP Variants

- **Motivation:** return address protection (shadow stack)
  - Validate every return (intended and unintended) against valid copies of return addresses
    [Davi et al., AsiaCCS 2011]
- **Exploit indirect jumps and calls**
  - ROP without returns
    [Checkoway et al., ACM CCS 2010]
1997

*ret2libc*
- Solar Designer

2001

*Advanced ret2libc*
- Nergal

2005

*Borrowed Code Chunk Exploitation*
- Krahmer

2007

*ROP on x86*
- Shacham (CCS)

2008

*ROP on SPARC*
- Buchanan et al (CCS)

*ROP on Atmel AVR*
- Francillon et al (CCS)

2009

*ROP Rootkits*
- Hund et al (USENIX)

*ROP on PowerPC*
- FX Lindner (BlackHat)

*ROP on ARM/iOS*
- Miller et al (BlackHat)

2010

*ROP without Returns*
- Checkoway et al (CCS)

*Practical ROP*
- Zovi (RSA Conference)

*Pwn2Own (iOS/IE)*
- Iozzo et al / Nils

2011/2012

*Real-World Exploits*

2013

*JIT-ROP*
- Snow et al (IEEE S&P)

*Blind ROP*
- Bittau et al (IEEE S&P)

2014

*Stitching Gadgets*
- Davi et al (USENIX)

*Out-Of-Control*
- Göktaş et al (IEEE S&P)

*Flushing Attacks*
- Schuster et al (RAID)

*ROP is Dangerous*
- Carlini et al (USENIX)
Overview of this Talk

- Background/Evolution of Code-Reuse Attacks
- Building Control-Flow Integrity Defenses
- Building Code Randomization Defenses
- Remote Attestation of Embedded Systems

Code-Reuse Attacks
ASLR – Address Space Layout Randomization
Basics of Code Randomization

- ASLR randomizes the base address of code/data segments

**Application Run 1**

Program Memory

- Library (e.g., libc)
- Executable
- Heap
- Stack

**Application Run 2**

Program Memory

- Library (e.g., libc)
- Executable
- Heap
- Stack

Brute-Force Attack [Shacham et al., ACM CCS 2004]

Guess Address of Library Function
Basics of Memory Randomization

- ASLR randomizes the base address of code/data segments

Application Run 1

Program Memory

Library (e.g., libc)

Executable

Heap

Stack

Application Run 2

Program Memory

Library (e.g., libc)

Executable

Heap

Stack

Disclosure Attack e.g., [Sotirov et al., Blackhat 2008]

1. Exploit disclosure vulnerability

2. Retrieve runtime $ADDR$ address

3. Revert all library addresses based on $ADDR$
Fine-Grained Code Randomization

- **ORP** [Pappas et al., IEEE S&P 2012]: Instruction reordering/substitution within a BBL
- **ILR** [Hiser et al., IEEE S&P 2012]: Randomizing each instruction’s location
- **STIR** [Wartell et al., ACM CCS 2012] &
  **XIFER** [Davi et al., AsiaCCS 2013]: Permutation of BBLs
Open Question:
Does fine-grained code randomization provide a viable defense in the long run?
Bad News:
Just-in-time code-reuse attacks (JIT-ROP) bypass fine-grained ASLR protection

High-Level Idea

Code Pointer

Code Page 1

INS_5
High-Level Idea

Code Pointer

Code Page 1

- INS_1
- INS_2
- INS_3
- JMP INS_10
- INS_4
- INS_5
- INS_6

Page Start

4KB

Page End

Disassemble

Scripting Engine
Code Randomization: Lessons Learned

1. Memory disclosure attacks are far more damaging than previously believed
   → A single address-instruction mapping leads to many leaks of code pages

2. Fine-grained ASLR can be bypassed with JIT-ROP
   → Enforce execute-only memory
     Software-based [Backes et al., CCS 2014]
     Hardware-based: Readactor(++) [with Crane et al., IEEE S&P 2015 & CCS 2015]
   → Combine code- and execution randomization
     Isomeron [with Liebchen et al., NDSS 2015]
   → Mitigating memory disclosure [Gawlik et al., DIMVA 2016]
Overview of this Talk

Background/Evolution of Code-Reuse Attacks

Building Control-Flow Integrity Defenses

Building Code Randomization Defenses

Remote Attestation of Embedded Systems
Basic Idea of Control-Flow Integrity
[Abadi et al., ACM CCS 2005 & TISSEC 2009]

Exit(C) == Label_5

A

Label_1

B

Label_2

C

Label_3

D

Label_4

E

Label_5

F

Label_6
Many CFI checks are required if unique labels are assigned per node.

Exit(B) == [Label_3, Label_4, Label_5]
Label Granularity: Trade-Offs (2/2)

- Optimization step: Merge labels to allow single CFI check
- However, this allows for unintended control-flow paths

```
Exit(B) == Label_3
```

```
Exit(C) == Label_3
```
Label Problem for Returns

- **Static CFI label checking** leads to coarse-grained protection for returns.

Program Code

Function A
- CALL C
- Code

Function B
- CALL C
- Code

Function C
- Code
- RETURN

Exit(C) == [Label_1, Label_2]

Diagram:

- Function A calls Function C
- Function B calls Function C
- Function C calls Function C
- Function C returns to Function A and Function B
- Exit(C) is labeled as [Label_1, Label_2]
Shadow Stack / Return Address Stack

- **Shadow stack** allows for fine-grained return address protection but incurs higher overhead.

\[
\text{Exit}(C) == \text{ShadowStack}[\text{TOS}]
\]

**Backup Storage for Return Addresses**

- **CALL** and **RET** stages.
- **Backup** and **Check** processes.
- **Return Addr** \(A'\).
CFI: Benefits and Limitations

Average overhead (without shadow stack): 16%; up to 45%

- Fine-grained protection
- Blackbox Vulcan (unpublished)
- Require side info (debug symbols, compiler support)
- Performance overhead
Hot Research Topic:
“Practical” (coarse-grained) Control Flow Integrity (CFI)

Recently, many solutions proposed

CCFIR
[IEEE S&P’13]

MS BlueHat Prize

kBouncer
[USENIX Sec’13]

ROPecker
[NDSS’14]

MS BlueHat Prize

CFI for COTS Binaries
[USENIX Sec’13]

ROPGuard
[Microsoft EMET]

Open Question:
Practical and secure mitigation of code reuse attacks
Turing-completeness of return-oriented programming
Negative Result:
All current (published) coarse-grained CFI solutions can be bypassed

Big Picture

Systematic Security Analysis of Coarse-Grained CFI

CFI Policies
Frequency of CFI Checks
Deriving a CFI policy that combines all schemes

Gadget Analysis
Turing-complete gadget set
Gadgets to bypass heuristics

Exploit Development
1. Systematic Security Analysis of Coarse-Grained CFI
Coarse-grained CFI leads to CFG imprecision

Reducing number of labels

Allowed paths: 1→2 and 2→1
Main Coarse-Grained CFI Policies

- **CFI Policy 1: Call-Preceded Sequences**
  - Returns need to target a call-preceded instruction
  - No shadow stack required

- **CFI Policy 2: Behavioral-Based Heuristics**
  - Prohibit a chain of $N$ short sequences each consisting of less than $S$ instructions

---

Application

<table>
<thead>
<tr>
<th>CALL A</th>
<th>INS_1</th>
<th>INS_2</th>
<th>CALL B</th>
<th>INS_3</th>
<th>CALL C</th>
<th>INS_4</th>
</tr>
</thead>
</table>

RET

1

2

... N

$< S$

$> S$

$< S$

$< S$

$< S$

$< S$

$< S$

$< S$

$< S$

$< S$
Main Coarse-Grained CFI Policies

- CFI Policy 1: Call-Preceded Sequences
  - Returns need to target a call-preceded instruction
  - No shadow stack required

- CFI Policy 2: Behavioral-Based Heuristics
  - Prohibit a chain of $N$ short sequences each consisting of less than $S$ instructions

Threshold Setting
- kBouncer: $(N=8; S<=20)$
- ROPecker: $(N=11; S<=6)$

Application

<table>
<thead>
<tr>
<th>CALL A</th>
<th>INS_1</th>
<th>INS_2</th>
<th>CALL B</th>
<th>INS_3</th>
<th>CALL C</th>
<th>INS_4</th>
</tr>
</thead>
</table>

- CALL B
- INS_3
- CALL C
- INS_4

RET

1 < S 2 < S N < S

1 > S

< S
## Deriving a Combined CFI Policy

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time of CFI Check</td>
<td>WinAPI</td>
<td>2 Page Sliding Window/Critical Functions</td>
<td>WinAPI/Critical Functions</td>
<td>Indirect Branch</td>
<td>Any Time</td>
</tr>
</tbody>
</table>

Here only the core policies shown. However, we consider all other deployed policies in our analysis.
2. Gadget Analysis
Methodology

Common Library
*kernel32.dll*

Sequence Finder (IDA Pro)

List of Call-Preceded Sequences

Sequence Filter (D Program)

Sequence Subset 1

Sequence Subset \( n \)

Search for Gadgets

Provide filters on Reg, Ins, Opnd, Length

Gadget Generation (manual)

MOV
ADD
CALL
XOR
STORE
(Excerpt of) Turing-Complete Gadget Set in CFI-Protected *kernel32.dll*

<table>
<thead>
<tr>
<th>Gadget Type</th>
<th>CALL-Preceded Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOAD Register</strong></td>
<td><strong>CALL-Preceded Sequence</strong> ending in a RET instruction</td>
</tr>
<tr>
<td>EBP  := pop ebp</td>
<td>ESI := pop esi; pop ebp</td>
</tr>
<tr>
<td>ESI  := pop esi; pop ebp</td>
<td>EDI := pop edi; leave</td>
</tr>
<tr>
<td>EDI  := pop edi; leave</td>
<td>ECX := pop ecx; leave</td>
</tr>
<tr>
<td>ECX  := pop ecx; leave</td>
<td>EBX := pop edi; pop esi; pop ebx; pop ebp</td>
</tr>
<tr>
<td>EBX  := pop edi; pop esi; pop ebx; pop ebp</td>
<td>EAX := mov eax,edi; pop edi; leave</td>
</tr>
<tr>
<td>EAX  := mov eax,edi; pop edi; leave</td>
<td>EDX := mov eax,[ebp-8]; mov edx,[ebp-4]; pop edi; leave</td>
</tr>
<tr>
<td><strong>LOAD/STORE Memory</strong></td>
<td><strong>LOAD/STORE Memory</strong></td>
</tr>
<tr>
<td>LD(EAX) := mov eax,[ebp+8]; pop ebp</td>
<td>ST(EAX) := mov [esi],eax; xor eax,eax; pop esi; pop ebp</td>
</tr>
<tr>
<td>ST(EAX) := mov [esi],eax; xor eax,eax; pop esi; pop ebp</td>
<td>ST(ESI) := mov [ebp-20h],esi</td>
</tr>
<tr>
<td>ST(ESI) := mov [ebp-20h],esi</td>
<td>ST(EDI) := mov [ebp-20h],edi</td>
</tr>
<tr>
<td><strong>Arithmetic/Logical</strong></td>
<td><strong>Arithmetic/Logical</strong></td>
</tr>
<tr>
<td>ADD/SUB := sub eax,esi; pop esi; pop ebp</td>
<td>XOR := xor eax,edi; pop edi; pop esi; pop ebp</td>
</tr>
<tr>
<td><strong>Branches</strong></td>
<td><strong>Branches</strong></td>
</tr>
<tr>
<td>unconditional branch 1 := leave</td>
<td>unconditional branch 2 := add esp,0Ch; pop ebp</td>
</tr>
<tr>
<td>unconditional branch 2 := add esp,0Ch; pop ebp</td>
<td>conditional LD(EAX) := neg eax; sbb eax,eax; and eax,[ebp-4]; leave</td>
</tr>
<tr>
<td>conditional LD(EAX) := neg eax; sbb eax,eax; and eax,[ebp-4]; leave</td>
<td></td>
</tr>
</tbody>
</table>
Long-NOP Gadget

ROP Gadget 1 → Store Registers → Prepare Long NOP → Long NOP → Reset Registers

ESI
EDI
EBX

Stack

Static Constants
Arbitrary Data Area (36 Bytes)

ROP Gadget 2 → …
3. Exploit Development

Adobe Reader 9.1
CVE-2010-0188

MPlayer Lite r33064 m3u
Buffer Overflow Exploit

Original exploits detected by coarse-grained CFI

Our instrumented exploits bypass coarse-grained CFI
Coarse-Grained CFI: Lessons Learned

1. Too many call sites available
   → Restrict returns to their actual caller (shadow stack)

2. Heuristics are ad-hoc and ineffective
   → Adjusted sequence length leads to high false positive

3. Too many indirect jump and call targets
   • Resolving indirect jumps and calls is non-trivial
   → Compromise: Compiler support
HAFIX: Hardware Flow Integrity Extensions

Design Decisions: Why CFI Processor Support?

CFI Processor Support based on Instruction set architecture (ISA) extensions

- High efficiency
- Dedicated CFI instructions
- System-wide protection
- CFI control state
- Binding of CFI data to CFI state and instructions
Big Picture

State 0
Normal Execution
- Function Calls
- Indirect Jumps
- Function Returns

CFI State
Only CFI instructions allowed
- CFI Check Call
- CFI Check Jump
- CFI Check Return
Return to the Original Caller

**State 0**
Normal Execution

**Function A**
- CFIBR label_A1
- CALL B
- CFIRET label_A1
- Code

**Function B**
- Code
- RET

**CFI State**
Only CFI instructions allowed

**Function A**
- CALL B
- Code

**Function B**
- Code
- RET

**Label State Stack (LSS)**
- label_A1
- ...

**Label State Stack (LSS)**
Evaluation

On average 1%-2% performance overhead

On average 13.5% increase in code size
Overview of this Talk

Background/Evolution of Code-Reuse Attacks

Building Control-Flow Integrity Defenses

Building Code Randomization Defenses

Remote Attestation of Embedded Systems

Code-Reuse Attacks
Remote attestation checks trustworthiness of a remote (embedded) device
Background: Principle of Remote Attestation

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

Attestation Protocol

Verifier

- Verify Report
- Measurement Database

Challenge

Prover

- Measure software state

Authentic Report
History of Remote Attestation

TPM Attestation
2001

Software-based Attestation
2004

PUF-based Attestation
2011

SWARM Attestation
2015

Property-based attestation
2004

Dynamic Root of Trust
2005

Minimal Trust Anchors
2010

Property-based Attest. [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]

Pioneer [SOSP’05]...
SWATT [SP’04]

Lightweight PUF Attest. [WiSEC’11]

SEDA [CCS’15]
SANA [CCS’16]
DARPA [WiSEC’16]
Key Limitation: current binary attestation schemes do not address runtime attacks
Problem Space of Runtime 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]

Adversary

Control-Flow Attack

Non-Control-Data Attack

DEP

X

inject malicious code

corrupt data pointer/variable

Memory write
Program flow

corrupt code pointer

switch (opmode)
case recovery: C
  case op1: D
  case op2: E, F

Adversary
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
Design and implementation of control-flow attestation (C-FLAT)

Detection of control-flow and non-control-data attacks

Two case studies based on embedded system software
Big Picture

Verifier

Control-Flow Graph (CFG) Analysis

Path Measurement

App A

Control-Flow Validation

Run-Time Path Measurement

Prover

LP_1

P_1, P_2

P_1, #LP_1

P_2

P^*_x

P^*_2

P^*_x, P^*_2
How to attest the executed control flows without transmitting all executed branches?
C-FLAT Measurement Function

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

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

\[
\begin{align*}
H_1 &= \mathcal{H}(0, A) \\
H_2 &= \mathcal{H}(H_1, B) \\
H_3 &= \mathcal{H}(H_2, C) \\
H_4 &= \mathcal{H}(H_2, D) \\
H_5 &= \mathcal{H}(H_2, E) \\
H_6 &= \mathcal{H}(H_5, F)
\end{align*}
\]
Loops are a challenge!

Different loop paths and loop iterations lead to many valid hash values.
C-FLAT Approach:

Treat loops as sub-graphs and report their hash values and # of iterations separately.
C-FLAT: Loop Handling

while (cond.) {...}

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

\[
\begin{align*}
\mathcal{H}_{6b} &= \mathcal{H}(\mathcal{H}_5, F) \\
\mathcal{H}_{6a} &= \mathcal{H}(\mathcal{H}_4, F) \\
\mathcal{H}_3 &= \mathcal{H}(\mathcal{H}_2, C) \\
\mathcal{H}_4 &= \mathcal{H}(\mathcal{H}_3, D) \\
\mathcal{H}_2 &= \mathcal{H}(\mathcal{H}_1, B) \\
\mathcal{H}_2a &= \mathcal{H}(0, B) \\
\mathcal{H}_3a &= \mathcal{H}(0, A) \\
\mathcal{H}_5 &= \mathcal{H}(\mathcal{H}_3, E) \\
\mathcal{H}_6 &= \mathcal{H}(\mathcal{H}_5, F) \\
\mathcal{H}_6b &= \mathcal{H}(\mathcal{H}_6, F) \\
\mathcal{H}_7 &= \mathcal{H}(\mathcal{H}_{2b}, F)
\end{align*}
\]

Loop Entry Hash: \( \mathcal{H}_1 \)

Loop Hash, Iteration: \( \mathcal{H}_{6a}, \#\mathcal{H}_{6a}, \mathcal{H}_{6b}, \#\mathcal{H}_{6b} \)
C-FLAT loop handling includes support of nested loops, break statements, and recursive functions.
Implementation
Prototype Architecture

- Implementation on Raspberry Pi 2

Application Binary

Trampolines

Measurement Engine and Attestation

Hardware

Normal World

Secure World (ARM TrustZone)
Binary Instrumentation

Function A
- *asm_ins*
- BL Function B
- *asm_ins*

Function B
- PUSH {...,LR}
- BLX R3
- POP {...,PC}

Function C
- *asm_ins*
- *asm_ins*
- BX LR

Function A
- *asm_ins*
- BL Direct-Call
- *asm_ins*

Function B
- PUSH {...,LR}
- BL Reg-Call
- BL Stack-Return

Function C
- *asm_ins*
- *asm_ins*
- *asm_ins*

Normal World

Secure World

Branch Table

Trampoline per branch type
- Store Context
- Save LR
- Lookup Target
- Secure Monitor Call
- Restore LR
- Restore Context
- Branch dest

C-FLAT Measurement Engine

Saved LR
LR

R1
R2
src,dest
Evaluation: Case Studies

Syringe Pump
Soldering Iron Temperature Controller
Syringe Pump

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

main()

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

processSerial()

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

action(direction,bolus)

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

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

action(direction, bolus)

Final Measurements for PUSH, PULL operations:

Loop Measurement:

97 78 fb fc 93 09 4e d7
ac 32 5d 65 eb 29 08 0c (#iterations)
Only the number of loop iterations is different per bolus 682 (0.1 ml), 1365 (0.2 ml)
Attacking the Syringe Pump

- Constructed several exploits to validate the effectiveness of C-FLAT
  - Control-flow attack uses ROP to dispense liquid at unexpected time
    → C-FLAT detects attack due to unexpected measurement
  - Non-control-data attack that dispenses more liquid than requested
    → C-FLAT detects attack due to an unexpectedly high number of loop iterations
Discussion

- C-FLAT attests control flow
  - Pure data attacks that don’t affect control flow are **not** covered
- Scalability depends on program size and complexity
  - We target typical (simple) embedded software, e.g., Syringe Pump that scales well for C-FLAT
- Reducing context switch overhead
  - ARMv8 Cortex-A53 needs ~3700 cycles at 800MHz; TrustZone-M only requires a few cycles
Conclusion

- **Code-reuse attacks are prevalent**
  - Google, Microsoft, Intel take these attacks seriously
  - Many real-world exploits
  - Existing solutions can be bypassed

- **Control-flow integrity (CFI) and fine-grained randomization are promising defense techniques**
  - Highly efficient hardware-assisted CFI enforcement
  - Fine-grained ASLR coupled with execute-only memory defends against JIT-ROP attacks