uXOM: Efficient eXecute-Only Memory on Cortex-M

Donghyun Kwon\textsuperscript{1,4}, Jangseop Shin\textsuperscript{1}, Giyeol Kim\textsuperscript{1}, Byoungyoung Lee\textsuperscript{1,2}, Yeongpil Cho\textsuperscript{3}, Yunheung Paek\textsuperscript{1}

\textsuperscript{1}Seoul National University, \textsuperscript{2}Purdue University, \textsuperscript{3}Soongsil University, \textsuperscript{4}Electronics and Telecommunications Research Institute
A memory which has only a execute permission
- No read and write permission

Purpose
- Protect intellectual properties (IPs)
- Prohibit obtaining CRA (Code Reuse Attack) gadgets at runtime
  - [Stephen et al. S&P’15]

High-end CPU architectures support XOM
- X86 – EPT, MPK
- AArch64 - MMU
Motivation

- ARMv7-M architecture
  - Used in Cortex-M3/4/7 processors
    - prominent processor in embedded systems
  - No MMU
  - No execute-only permission in MPU (Memory Protection Unit)
    - Available permissions: NA, RO, RX, RW, RWX

- We propose uXOM
  - New software technique to implement XOM on Cortex-M processors.
Threat model & Assumption

- Consider software attacks at runtime
  - Assume that target firmware has memory vulnerabilities.
  - Attacker can perform arbitrary memory read and write
  - Attacker can subvert control-flow
    - Manipulate function pointer or return address

- Not consider offline attacks on firmware
- Not consider hardware attacks
  - Bus probing, memory tampering, etc.

- Any software components of the firmware are not trusted
  - include the exception handlers
- All software components are executed in privileged mode
  - [Abraham el al. S&P’17], [Chung Hwan et al. NDSS’18]
Basic Design

LDR R0, [R1]

...
Basic Design

1. Execute the code in Privileged mode

2. P: RX, U: NA

3. LDRT R0, [R1]

Code memory

uXOM: Efficient eXecute Only Memory on Cortex-M
**Basic Design**

1. Execute the code in Privileged mode

   \[ \text{LDRT R0, [R1]} \]

2. \[ \text{P: RX, U: NA} \]

3. \[ \text{STRT R2, [R3]} \]

4. Code memory

   Private Peripheral Bus (PPB)

uXOM: Efficient eXecute Only Memory on Cortex-M
C1. Unconvertible memory instructions
- Exclusive memory instructions (LDREX, STREX)
- PPB access memory instructions
Challenges

- **C1. Unconvertible memory instructions**
  - Exclusive memory instructions (LDREX, STREX)
  - PPB access memory instructions

- **C2. Malicious indirect branches**
  - Jump to unconverted memory instructions
    - By manipulating target address register

- **C3. Malicious exception returns**
  - Return to unconverted memory instructions
    - By manipulating exception context (PC) in the stack
Challenges

- C1. Unconvertible memory instructions
  - Exclusive memory instructions (LDREX, STREX)
  - PPB access memory instructions
- C2. Malicious indirect branches
  - Jump to unconverted memory instructions
    - By manipulating target address register
- C3. Malicious exception returns
  - Return to unconverted memory instructions
    - By manipulating exception context (PC) in the stack
- C4. Malicious data manipulation
Challenges

- **C1. Unconvertible memory instructions**
  - Exclusive memory instructions (LDREX, STREX)
  - PPB access memory instructions

- **C2. Malicious indirect branches**
  - Jump to unconverted memory instructions
    - By manipulating target address register

- **C3. Malicious exception returns**
  - Return to unconverted memory instructions
    - By manipulating exception context (PC) in the stack

- **C4. Malicious data manipulation**

- **C5. Unintended instructions**
  - Unaligned execution
  - Execution of embedded data in the code memory
Solving Challenges

- Finding Unconvertible Memory Instructions ➔ C1
  - Exclusive Memory Instructions
    • Identified by opcode in the instruction encoding
  - PPB access instructions
    • Check if the accessed memory address is belonging to PPB region
    • Intra-procedure analysis
Solving Challenges

- Atomic Verification Technique ➔ C4
  - Add the verification routine before the unconverted instruction
  - Disable exception during the instruction sequence
    • Protection against an attacker generates an exception after the verification code

```assembly
1: update_register:
2: str r1, [r0]
3:
4:
5:
6:
7: str r1, [r0]
8:
9:
10:
11:
12:
```

```assembly
1: update_register:
2: cpsid i
3: [verification routine]
4: str r1, [r0]
5: cpsie i
6:
7:
8:
9:
10:
11:
12:
```

Atomic instruction Sequence
Solving Challenges

- Atomic Verification Technique (cont’d) ➔ C2, C3
  - 1) Use a dedicated register as memory address register of unconverted instructions
  - 2) Enforce following two invariant properties
    - IP1) When atomic instruction seq. is executed, the dedicated register holds sensitive address
    - IP2) When atomic instruction seq. is not executed, the dedicated register holds non-harmful value
      - ➔ instrumentation for IP2 requires tremendous overhead
      - ➔ The dedicated register cannot be used in the code except for the atomic verification sequences

- Drawback
  - Increase register spills ➔ Performance Drop
Solving Challenges

- Atomic Verification Technique (cont’d) ➔ C2, C3
  - 1) Use a SP register as memory address register of unconverted instructions
  - 2) Enforce following two invariant properties
    - IP1) When atomic instruction seq. is executed, SP register holds sensitive address
    - IP2) When atomic instruction seq. is not executed, SP register points non-harmful value
  - ➔ instrumentation for IP2 could be implemented in a efficient way
  - ➔ SP register can be used in the code including the atomic verification sequences
Solving Challenges

- Atomic Verification Technique (cont’d)

```assembly
update_register:
    str r1, [r0]

update_register:
    cpsid i       // disable interrupt
    mov r10, sp   // backup the value of sp
    mov sp, r0    // set sp to a target address (IP1)
[verification routine]  // verify the subsequent unconverted inst.
    str r1, [sp]  // perform an unconverted inst.
    mov sp, r10   // restore the value of sp
    [check sp]    // check the value of sp (IP2)
    cpsie i      // enable interrupt
```
Solving Challenges

- Handling Unintended Instructions ➔ C5
  - Replace the exploitable instruction with safe instruction sequence
    - Serves the same functionality
  - Use static binary analysis to find out all exploitable instructions.
Evaluation

- Implementation
  - Code Instrumentation: LLVM 5.0
  - Binary analysis: Radare2

- Experiment setup
  - Arduino-due
    - Cortex-M3 processor
  - RIOT-OS
  - BEEBS benchmark suite
Evaluation

![Bar chart showing evaluation results for SFI-XOM, uXOM, uXOM-UI, and uXOM-CRA for code size, execution time, and energy.]
Evaluation

![Bar chart showing comparison of Code Size, Execution Time, and Energy between SFI-XOM, uXOM, uXOM-UI, and uXOM-CRA.](chart.png)
Evaluation

![Graph showing evaluation results for different memory models](image-url)
Evaluation

![Evaluation Diagram]

- Code Size
- Execution Time
- Energy

Legend:
- SFI-XOM
- uXOM
- uXOM-UI
- uXOM-CRA
Conclusion

▪ Software technique to implement execute-only memory on Cortex-M processors
  – MPU, unprivileged memory instructions

▪ Strong threat model
  – Assuming attacker is able to read/modify the memory and subvert control-flow
  – Do not assume any software TCB in the system

▪ Evaluation
  – Better than SFI-based XOM in terms of performance and security
  – uXOM is compatible with existing XOM-based solutions (Key protection, CRA defense)
Q & A

Thank you for listening