Silhouette: Efficient Protected Shadow Stacks for Embedded Systems

Jie Zhou
University of Rochester

Yufei Du
University of Rochester

Zhuojia Shen
University of Rochester

Lele Ma
College of William & Mary

John Criswell
University of Rochester

Robert J. Walls
Worcester Polytechnic Institute

Presented at USENIX Security 2020
Microcontroller-based Systems are Almost Everywhere

ECU

Bluetooth module

Wi-Fi module
Microcontroller-based Embedded Devices

- Limited CPU speed
- Limited memory
- Real-time constraints
- Frequent direct operations on hardware

THE C PROGRAMMING LANGUAGE
Control-flow Hijacking: corrupting control-data to divert control flow to attacker-selected destinations

```c
void foo(...) {
    ...
    bar();
    ...
}

void bar(...) {
    ...
    ...
    return;
}
```
Control-flow Hijacking: corrupting control-data to divert control flow to attacker-selected destinations
Control-Flow Integrity (CFI)

Common weakness of practical CFI*: allowing a return instruction to return back to multiple places

*Exploited by Out of Control @Oakland’14, ROP is Still Dangerous @USENIX Security’14, Control-flow Bending @USENIX Security’15, etc.
Silhouette: a compiler-based defense that

- guarantees the integrity of return addresses
- coarse-grained forward-edge CFI
- low performance overhead (1.3% and 3.4% overhead on two benchmark suites)

- Developed for ARMv7-M due to its popularity
- Also working on other ARM embedded processors
Outline

• Silhouette Design

• Evaluation

• Summary
Outline

• Silhouette Design

• Evaluation

• Summary
Shadow Stack

Protecting *return addresses*

Shadow stack itself also needs protection!

![Diagram of Shadow Stack]

- SSP: 0x40000000
- SP: 0x36000000
- Regular stack:
  - top: 0x36000000
  - bottom: 0x30000000
- Shadow stack:
  - top: 0x40000000
  - bottom: 0x30000000

- ret. addr. of foo
- ret. addr. of bar

**Figure:** Shadow Stack Diagram

10
From a Shadow Stack’s Point of View

All Store Instructions

- stores writing to the shadow stack
- all other stores

Legal

Illegal

Can we make the shadow stack writable only by its legal stores?
Background on ARMv7-M

• Execution Mode: Privileged and Unprivileged.
  (Embedded devices usually run everything in privileged mode.)
• Memory access permissions are configurable.

Can we make the shadow stack only writable by its legal stores?

Is it possible to make
  • only the shadow-stack-legal stores privileged
  • all other stores unprivileged?

Yes
Act as if running in unprivileged mode when running in privileged mode.

// running in privileged mode
strt r1, [r0, #12]  \[writable only by privileged stores\]

This instruction would fail!
Use Unprivileged Store to Protect Shadow Stack

- Configure the memory region for shadow stack to be writable only by privileged stores.

- Transform all stores to be unprivileged stores except
  - shadow-stack-legal stores
  - those that require to run as privileged such as some I/O-related operations.

**Effect**: even if memory is corrupted and control flow is diverted, illegal store instructions do not have write access to corrupt the shadow stack.
Store Instructions of ARMv7-M

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Number of Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Store Instructions</td>
<td>over 40</td>
</tr>
<tr>
<td>source register, base register, offset register, immediate, left shift, write back, store multiple, floating-point stores</td>
<td></td>
</tr>
<tr>
<td>Unprivileged Store Instructions</td>
<td>3</td>
</tr>
<tr>
<td>source register, base register, immediate</td>
<td></td>
</tr>
</tbody>
</table>

Comparison of Normal and Unprivileged Store Instructions
Store Hardening Examples

// example 1
str r0, [r1, #4]
no performance overhead
no code size overhead
str r0, [r1, #4]

// example 2
str r0, [sp, #-12]
performance and code size overhead
sub sp, #12
str r0, [sp, #0]
add sp, #12
Forward-edge Control-flow Issues

- Transform all stores to be unprivileged stores except
  - shadow-stack-legal stores
  - those that require to run as privileged

<table>
<thead>
<tr>
<th>Forward-edge Control Flow</th>
<th>How Silhouette Handles Them</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect Function Calls</td>
<td>Restricted by Label-based Forward-edge CFI</td>
</tr>
<tr>
<td>Large switch Statements</td>
<td>Compiled to Bounds-checked TBB or TBH instructions</td>
</tr>
<tr>
<td>Computed goto Statements</td>
<td>Transformed to switch statements</td>
</tr>
</tbody>
</table>
Security guarantee:
• Return instruction always returns to its legal destination
• Forward-edge control flows are restricted to selected destinations
Outline

• Silhouette Design

• Evaluation

• Summary
Experiment Setup

Evaluated both performance and code size overhead

Development board: **STM32F469**
- Cortex-M4 processor, run at 180 MHz
- 384 KB SRAM
- 16 MB SDRAM
- 2 MB Flash Memory

Benchmarks: all 9 programs in **CoreMark-Pro**
- 29 programs in **BEEBS**

Base compiler: Clang/LLVM 9.0

Optimization level: -O3
Performance on CoreMark-Pro Benchmarks

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Shadow Stack</th>
<th>Store Hardening</th>
<th>CFI</th>
<th>Silhouette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>-0.1%</td>
<td>0.1%</td>
</tr>
<tr>
<td>Max</td>
<td>1.3%</td>
<td>4.9%</td>
<td>0.1%</td>
<td>4.9%</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>0.2%</td>
<td>1%</td>
<td>0</td>
<td>1.3%</td>
</tr>
</tbody>
</table>
Code Size on CoreMark-Pro Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Shadow Stack</th>
<th>Store Hardening</th>
<th>CFI</th>
<th>Silhouette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0.5%</td>
<td>2.8%</td>
<td>0.2%</td>
<td>3.6%</td>
</tr>
<tr>
<td>Max</td>
<td>1.7%</td>
<td>11.1%</td>
<td>9.4%</td>
<td>19.3%</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>0.8%</td>
<td>6.8%</td>
<td>1.2%</td>
<td>8.9%</td>
</tr>
</tbody>
</table>
Performance Overhead on BEEBS Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Shadow Stack</th>
<th>Store Hardening</th>
<th>CFI</th>
<th>Silhouette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0</td>
<td>-0.3%</td>
<td>-0.3%</td>
<td>-0.3%</td>
</tr>
<tr>
<td>Max</td>
<td>9.2%</td>
<td>24.7%</td>
<td>2.2%</td>
<td>24.8%</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>1.1%</td>
<td>1.8%</td>
<td>0.1%</td>
<td>3.4%</td>
</tr>
</tbody>
</table>
Code Size on BEEBS Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Shadow Stack</th>
<th>Store Hardening</th>
<th>CFI</th>
<th>Silhouette</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0.3%</td>
<td>0.5%</td>
<td>0</td>
<td>0.9%</td>
</tr>
<tr>
<td>Max</td>
<td>0.6%</td>
<td>6.1%</td>
<td>1.3%</td>
<td>6.8%</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>0.4%</td>
<td>1.8%</td>
<td>0.1%</td>
<td>2.3%</td>
</tr>
</tbody>
</table>
Silhouette-Invert

Writable only by unprivileged stores but not by privileged stores?

Silhouette-Invert
• Configure shadow stack to be unprivileged-write-only
• Transform shadow-stack-legal stores to be unprivileged
• Leave all other stores unchanged

Not supported on ARMv7-M

Proposed two solutions with minor hardware changes. See the paper for details.
Silhouette v.s. Silhouette-Invert on CoreMark-Pro

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Silhouette</th>
<th>Silhouette-Invert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0.1%</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>4.9%</td>
<td>1.5%</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>1.3%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>
Silhouette v.s. Silhouette-Invert on BEEBS

<table>
<thead>
<tr>
<th></th>
<th>Silhouette</th>
<th>Silhouette-Invert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>-0.3%</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>24.8%</td>
<td>18.6%</td>
</tr>
<tr>
<td>Geo. Mean</td>
<td>3.4%</td>
<td>1.9%</td>
</tr>
</tbody>
</table>
Summary

• Silhouette: an efficient defense to protect return addresses for ARM embedded systems

• Low performance and code size overhead

• Silhouette-Invert:
  • Further decreases performance and code size penalty
  • Minor hardware change

• Open-Source: https://github.com/URSec/Silhouette

Contacts:
  Jie Zhou (jzhou41@cs.rochester.edu)
  Zhuojia Shen (zshen10@cs.rochester.edu)
  John Criswell (criswell@cs.rochester.edu)