Silhouette: Efficient Protected Shadow Stacks for Embedded Systems

Jie Zhou, Yufei Du, and Zhuojia Shen, *University of Rochester;* Lele Ma, *University of Rochester and College of William and Mary;* John Criswell, *University of Rochester;* Robert J. Walls, *Worcester Polytechnic Institute*

https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Silhouette: Efficient Protected Shadow Stacks for Embedded Systems

Jie Zhou[1], Yufei Du[1], Zhuojia Shen[1], Lele Ma[1,2*], John Criswell[1], and Robert J. Walls[3]

[1]University of Rochester
[2]College of William & Mary
[3]Worcester Polytechnic Institute

Abstract

Microcontroller-based embedded systems are increasingly used for applications that can have serious and immediate consequences if compromised—including automobile control systems, smart locks, drones, and implantable medical devices. Due to resource and execution-time constraints, C is the primary language used for programming these devices. Unfortunately, C is neither type-safe nor memory-safe, and control-flow hijacking remains a prevalent threat.

This paper presents *Silhouette*: a compiler-based defense that efficiently guarantees the integrity of return addresses, significantly reducing the attack surface for control-flow hijacking. Silhouette combines an incorruptible shadow stack for return addresses with checks on forward control flow and memory protection to ensure that all functions return to the correct dynamic caller. To protect its shadow stack, Silhouette uses *store hardening*, an efficient intra-address space isolation technique targeting various ARM architectures that leverages special store instructions found on ARM processors.

We implemented Silhouette for the ARMv7-M architecture, but our techniques are applicable to other common embedded ARM architectures. Our evaluation shows that Silhouette incurs a geometric mean of 1.3% and 3.4% performance overhead on two benchmark suites. Furthermore, we prototyped *Silhouette-Invert*, an alternative implementation of Silhouette, which incurs just 0.3% and 1.9% performance overhead, at the cost of a minor hardware change.

1 Introduction

Microcontroller-based embedded systems are typically developed in C, meaning they suffer from the same memory errors that have plagued general-purpose systems [4, 59, 67]. Indeed, hundreds of vulnerabilities in embedded software have been reported since 2017.[1] Exploitation of such systems can directly lead to physical consequences in the real world. For example, the control system of a car is crucial to passenger safety; the security of programs running on a smart lock is essential to the safety of people's homes. As these systems grow in importance,[2] their vulnerabilities become increasingly dangerous [40, 54, 61].

Past work on control-flow hijacking attacks highlights the need to protect return addresses, even when the software employs other techniques such as forward-edge control-flow integrity (CFI) [19, 20, 25, 29, 37]. Saving return addresses on a separate shadow stack [18] is a promising approach, but shadow stacks themselves reside in the same address space as the exploitable program and must be protected from corruption [18, 25]. Traditional memory isolation that utilizes hardware privilege levels can be adapted to protect the shadow stack [70], but it incurs high overhead as there are frequent crossings between protection domains (e.g., once for every function call). Sometimes *information hiding* is used to approximate intra-address space isolation as it does not require an expensive context switch. In information hiding, security-critical data structures are placed at a random location in memory to make it difficult for adversaries to guess the exact location [43]. Unfortunately, information hiding is poorly suited to embedded systems as most devices have a limited amount of memory that is directly mapped into the address space—e.g., the board used in this work has just 384 KB of SRAM and 16 MB of SDRAM [66].

This paper presents *Silhouette*: an efficient write-protected shadow stack [28] system that guarantees that a return instruction will always return to its dynamic legal destination. To provide this guarantee, Silhouette combines a shadow stack, an efficient intra-address space isolation mechanism that we call *store hardening*, a Control-Flow Integrity [1] implementation to protect forward-edge control flow, and a corresponding Memory Protection Unit (MPU) configuration to enforce memory access rules. Utilizing the unprivileged store instructions on modern embedded ARM architectures,

*Work done when the author was visiting the University of Rochester.

[1]Examples include CVE-2017-8410, CVE-2017-8412, CVE-2018-3898, CVE-2018-16525, CVE-2018-16526, and CVE-2018-19417.

[2]Both Amazon and Microsoft have recently touted operating systems targeting microcontroller-based embedded devices [16, 52]

store hardening[3] creates a logical separation between the code and memory used for the shadow stack and that used by application code. Unlike hardware privilege levels, store hardening does not require expensive switches between protection domains. Also, unlike the probabilistic protections of information hiding, protections based on store hardening are hardware-enforced. Further, the forward-edge control-flow protection prevents unexpected instructions from being executed to corrupt the shadow stack or load return addresses from illegal locations. Finally, the MPU configuration enforces memory access rules required by Silhouette.

We focus on the ARMv7-M architecture [12] given the architecture's popularity and wide deployment; however, our techniques are also applicable to a wide range of ARM architectures, including ARMv7-A [11] and the new ARMv8-M Main Extension [13]. We also explore an alternative, inverted version of Silhouette that promises significant performance improvements at the cost of minor hardware changes; we call this version *Silhouette-Invert*. We summarize our contributions as follows:

- We built a compiler and runtime system, Silhouette, that leverages store hardening and coarse-grained CFI to provide embedded applications with efficient intra-address space isolation and a protected shadow stack.
- We have evaluated Silhouette's performance and code size overhead and found that Silhouette incurs a geometric mean of 1.3% and 3.4% performance overhead, and a geometric mean of 8.9% and 2.3% code size overhead on the CoreMark-Pro and the BEEBS benchmark suites, respectively. We also compare Silhouette to two highly-related defenses: RECFISH [70] and μRAI [5].
- We prototyped and evaluated the Silhouette-Invert variant and saw additional improvements with an average performance overhead measured at 0.3% and 1.9% by geometric mean and code size overhead measured at 2.2% and 0.5%, again, on CoreMark-Pro and BEEBS.

In addition to the above contributions, we observe that store hardening could be extended to protect other security-critical data, making Silhouette more flexible than other approaches. For example, Silhouette could be extended to isolate the sensitive pointer store for Code-Pointer Integrity (CPI) [43]. Similarly, it could be used to protect kernel data structures within an embedded operating system (OS) such as Amazon FreeRTOS [16].

2 ARMv7-M Architecture

Our work targets the ARMv7-M architecture [12]. We briefly summarize the privilege and execution modes, address space layout, and memory protection features of the ARMv7-M.

[3] *u*XOM [44] independently developed a similar technique for implementing execute-only memory. We compare the implementation differences between store hardening and that of *u*XOM in Section 6.2.

Embedded Application Privilege Modes ARMv7-M supports the execution of both *privileged* and *unprivileged* code. Traps, interrupts, and the execution of a *supervisor call* (`SVC`) instruction switches the processor from unprivileged mode to privileged mode. Unlike server systems, embedded applications often run in privileged mode. Such applications also frequently use a *Hardware Abstraction Layer* (HAL) to provide a software interface to device-specific hardware. HAL code is often generated by a manufacturer-provided tool (e.g., HALCOGEN [68]), is linked directly into an application, and runs within its address space.

Address Space ARMv7-M is a memory-mapped architecture, lacking support for virtual memory and using a 32-bit address space. While the exact layout varies between hardware, the address space is generally divided into 8 sections. The `Code` section holds code and read-only data; it usually maps to an internal ROM or flash memory. An `SRAM` section along with two `RAM` sections are used to store runtime mutable data, e.g., the stack, heap, and globals. The `Peripheral` and two `Device` regions map hardware timers and I/O device registers. The `System` region maps system control registers into the processor's physical address space.

A security-critical subsection of `System` is the *System Control Space*, which is used for important tasks such as system exception management. It also contains the address space for the Memory Protection Unit (MPU) [12]. Since ARMv7-M is a memory-mapped architecture, all of the security-critical registers, such as MPU configuration registers, are also mapped to the `System` region.

Memory Protection Unit An ARMv7-M-based device can optionally have a Memory Protection Unit. The MPU is a programmable memory protection component that enforces memory access permissions [9, 12]. The MPU allows privileged software to create a set of memory regions which cover the physical address space; the permission bits on each region dictate whether unprivileged and privileged memory accesses can read or write the region. The number of configurable MPU regions is implementation specific, e.g., the target device in this paper supports 8 regions [65]. The memory regions configured by the MPU do not need to exactly match the default memory regions described in the **Address Space** paragraph. The size of each MPU-configured region varies from 32 bytes to 4 GB.

Currently, the MPU design makes several assumptions about how memory access permissions are to be configured. First, it assumes that privileged software should have as many or more access rights to memory than unprivileged code. Consequently, the MPU cannot be configured to give unprivileged code more access to a memory region than privileged code.

Second, the MPU assumes that certain memory regions—e.g., the `System` region—should not be executable, and it prevents instruction fetches from these regions regardless of the MPU configuration. Third, the MPU design assumes that unprivileged code should not be able to reconfigure security-critical registers on the processor. Therefore, the MPU will prevent unprivileged code from writing to memory regions that include memory-mapped device registers, such as those that configure the MPU.

3 Threat Model and System Assumptions

While embedded code can be conceptually divided into application code, libraries, kernel code, and the hardware abstraction layer, there is often little distinction *at runtime* between these logical units. Due to performance, complexity, and real-time considerations, it is quite common for all of this code to run in the same address space, without isolation, and with the same privilege level [24, 42, 44]. For example, under the default configuration of Amazon FreeRTOS (v1.4.7), *all* code runs as privileged in ARMv7-M [16]. These embedded characteristics heavily inform our threat model and the design decisions for Silhouette.

Our threat model assumes a strong adversary that can exploit a memory error in the *application code* to create a *write-what-where* style of vulnerability. That is, the adversary can attempt to write to any location in memory at any time. The adversary's goal is to manipulate the control flow of a program by exploiting the aforementioned memory error to overwrite memory (e.g., a return address). Non-control data attacks [21, 39] are out of scope of this work. Further, we assume the adversary has full knowledge of the memory contents and layout; we do not rely on information hiding for protection. Our threat model is consistent with past work on defenses against control-flow hijacking.

We assume the target system runs a single bare-metal application statically linked with all the library code and the hardware abstraction layer (HAL)—the latter provides a device-specific interface to the hardware. We assume the HAL is part of the Trusted Computing Base (TCB) and is either compiled separately from the application code or annotated, allowing Silhouette to forgo transformations on the HAL that might preclude privileged hardware operations. Similarly, we assume that exception handlers are part of the TCB. Further, we assume the whole binary runs in privileged mode for the reasons mentioned previously.

Finally, we assume the target device includes a memory protection unit (or similar hardware mechanism) for configuring coarse-grained memory permissions, i.e., Silhouette is able to configure read, write, and execute permissions for five regions (summarized in Section 6.4) of the address space.

4 Intra-Address Space Isolation

Many security enforcement mechanisms rely on intra-address space isolation to protect security-critical data; in other words, the defenses are built on the assumption that application code, under the influence of an attacker, cannot modify security-critical regions of the address space. For example, defenses with shadow stacks [18] need a safe region to store copies of return addresses, and CPI [43] needs a protected region of the address space to place its safe stack and sensitive pointer store. Complicating matters, defenses often intersperse accesses to the protected region with regular application code; the former should be able to access the protected region while the latter should not. Consequently, existing mechanisms to switch between protection domains—e.g., system calls between unprivileged and privileged mode—are often too inefficient for implementing these security mechanisms for microcontroller-based embedded systems. Rather than incur the performance penalty of true memory isolation, some defenses hide the security-critical data structures at random locations in the address space [24, 43]. Embedded systems have limited entropy sources for generating random numbers and only kilobytes or megabytes of usable address space; we do not believe hiding the shadow stack will be effective on such systems.

We devise a protection method, *store hardening*, for embedded ARM systems utilizing unique features of a subset of ARM architectures [11–13], including ARMv7-M. These architectures provide special unprivileged store instructions for storing 32-bit values (`STRT`), 16-bit values (`STRHT`), and 8-bit values (`STRBT`). When a program is running in the processor's privileged mode, these store instructions are treated as though they are executed in unprivileged mode, i.e., the processor always checks the unprivileged-mode permission bits configured in the MPU when executing an `STRT`, `STRHT`, or `STRBT` instruction regardless of whether the processor is executing in privileged or unprivileged mode. We leverage this feature to create two protection domains. One *unprivileged domain* contains regular application code and only uses the unprivileged `STRT`, `STRHT`, and `STRBT` instructions for writing to memory. The second *privileged domain* uses regular (i.e., privileged) store instructions. As code from both domains runs in the same, privileged, processor mode, this method allows us to enforce memory isolation without costly context switching.

To completely isolate the data memory used by the unprivileged and privileged domains, two additional features are needed. First, there needs to be a mechanism to prevent unprivileged code from jumping into the middle of privileged code; doing so could allow unprivileged code to execute a privileged store instruction with arbitrary inputs. We can use forward-edge CFI checks to efficiently prevent such attacks. Second, a trusted code scanner must ensure that the code contains no *system instructions* that could be used to modify important program state without the use of a store instruction.

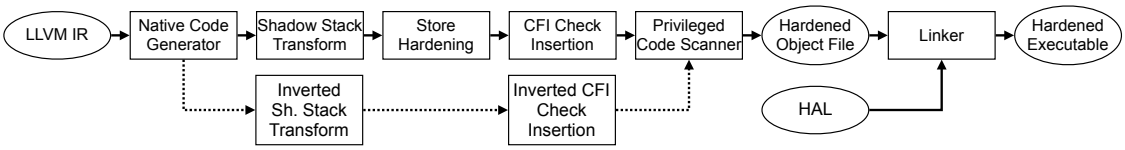

Figure 1: Architecture of Silhouette and the Silhouette-Invert Variant

For example, an adversary could use the `MSR` instruction [12] to change the value of the main or process stack pointer registers (`MSP` and `PSP`, respectively), effectively changing the location of the shadow stack and potentially moving it to an unprotected memory region. We discuss a defense that leverages these techniques in the next section.

5 Silhouette Design

Silhouette is a compiler and run-time system that leverages our memory isolation scheme to efficiently protect embedded systems from control-flow hijacking attacks. As Figure 1 shows, Silhouette transforms application code with four new compiler passes placed after native code generation but before linking the hardened object code with the hardware abstraction layer (HAL). We also explore an alternative, inverted version of these passes that promises significant performance improvements at the cost of minor hardware changes; we call this version *Silhouette-Invert* (see Section 5.5). Silhouette's new compiler passes are as follows:

1. **Shadow Stack Transformation**: The shadow stack transformation modifies the native code to save return values on a shadow stack and to use the return value stored in the shadow stack in return instructions.

2. **Store Hardening**: The store hardening pass modifies all store instructions, except those used in the shadow stack instrumentation and Store-Exclusive instructions [12] (see Section 5.2 for the reasons), to use variants that check the unprivileged-mode permission bits.

3. **CFI Transformation**: The CFI transformation instruments indirect function calls and other computed branches (aside from returns) to ensure that program execution follows a pre-computed control-flow graph. Consequently, this instrumentation prevents the execution of gadgets that could, for example, be used to manipulate protected memory regions.

4. **Privileged Code Scanner**: The privileged code scanner analyzes the native code prior to emitting the final executable to ensure that application code is free of privileged instructions that an adversary might seek to use to disable Silhouette's protections.

In addition to the above transformations, Silhouette employs mechanisms to prevent memory safety errors from disabling the hardware features that Silhouette uses to provide its security guarantees. In the context of ARMv7-M, it means that the MPU cannot be reconfigured to allow unprivileged accesses to restricted memory regions. Also note that the HAL library is not transformed with Silhouette as it may contain I/O functions that need to write to memory-mapped I/O registers that are only accessible to privileged store instructions. We also forbid inlining HAL functions into application code.

Moreover, Silhouette specially handles variable-length arrays on the stack and `alloca()` calls with argument values that cannot be statically determined by the compiler. For these two types of memory allocation, Silhouette adopts the method from SAFECode [31] and SVA [27] that promotes the allocated data from stack to heap. As Section 7.1 explains, such stack allocations (while rare in C code) can cause stack register spills, endangering the integrity of the shadow stack.

5.1 Shadow Stack

In unprotected embedded systems, programs store return addresses on the stack, leaving return addresses open to corruption by an adversary. To mitigate such attacks, some compilers transform code to use shadow stacks. A *shadow stack* [18] is a second stack, stored in an isolated region of memory, on which a program saves the return address. Only the code that saves the return address should be able to write to the shadow stack; it should be otherwise inaccessible to other store instructions in the program. If the shadow stack cannot be corrupted by memory safety errors, then return addresses are not corrupted. Furthermore, if the function epilogue uses the correct return address stored on the shadow stack, then the function always returns to the correct dynamic call site.

Silhouette's shadow stack transformation pass modifies each function's prologue to save the return address on a shadow stack and each function's epilogue to use the shadow stack return address on function return. A special case to handle is `setjmp`/`longjmp`. `setjmp` saves the current execution context to a memory location specified by its argument, and `longjmp` recovers the saved context from the specified memory location as if the execution was just returned from a previous call to `setjmp`. Using `setjmp`/`longjmp`, a program is able to perform non-local indirect jumps that are challenging to track by a shadow stack. As few programs use `setjmp`/`longjmp`, we refer interested readers to Appendix A

which discusses how Silhouette supports these two functions. Once the transformation is complete, the program uses a shadow stack, but the shadow stack is not protected. For that, Silhouette employs the store hardening pass and the CFI pass.

5.2 Protection via Store Hardening

Silhouette leverages the MPU and the intra-address space isolation mechanism described in Section 4 to efficiently protect the shadow stack. This protection is comprised of two parts. First, during compilation, Silhouette's *store hardening* pass transforms all store instructions in application code from privileged instructions to equivalent unprivileged store instructions (`STRT`, `STRHT`, and `STRBT`). As discussed previously, these unprivileged variants always check the MPU's unprivileged-mode permission bits. Second, when loading the program, Silhouette instrumentation configures the MPU so that the shadow stack is readable and writeable in privileged mode but only readable in unprivileged mode. This ensures that store instructions executed in unprivileged mode and unprivileged stores (`STRT`, `STRHT`, and `STRBT`) executed in privileged mode cannot modify values on the shadow stack. Together, these mechanisms ensure shadow stack isolation, *even if the entire program is executed in privileged mode*.

Store hardening transforms all stores within the application code except for two cases. First, store hardening does not transform stores used as part of Silhouette's shadow stack instrumentation as they must execute as privileged instructions so that they can write to the shadow stack. The shadow stack pass marks all stores to the shadow stack with a special flag, making them easily identifiable. Second, store hardening cannot transform atomic stores (Store-Exclusive [12]) because they do not have unprivileged counterparts. Silhouette utilizes Software Fault Isolation (SFI) [69] to prevent those stores from writing to the shadow stack region.

As discussed in Section 3, Silhouette does not transform the HAL code; thus, the stores in the HAL code are left unmodified. This is because the HAL contains hardware I/O and configuration code that must be able to read and write the `System`, `Device`, and `Peripheral` memory regions. To prevent attackers from using privileged stores within the HAL code, Silhouette employs CFI as Section 5.3 explains.

5.3 Forward Branch Control-Flow Integrity

Shadow stacks protect the integrity of function returns, but memory safety attacks can still corrupt data used for forward-edge control flow branches, e.g., function pointers. If left unchecked, these manipulations would allow an attacker to redirect control flow to anywhere in the program, making it trivial for the attacker to corrupt the shadow stack with an arbitrary value or to load a return address from an arbitrary location. Consequently, Silhouette must restrict the possible targets of forward-edges to ensure return address integrity.

There are two types of forward branches: indirect function calls and forward indirect jumps. For the former, Silhouette uses label-based CFI checks [1,17] to restrict the set of branch targets and ensure that the remaining privileged store instructions cannot be leveraged by an attacker to corrupt the shadow stack. Silhouette-protected systems use privileged store instructions only in the HAL library and in function prologues to write the return address to the shadow stack. The HAL library is compiled separately and has no CFI labels in its code; even coarse-grained CFI ensures that no store instructions within the HAL library can be exploited via an indirect call (direct calls to HAL library functions are permitted as they do not require CFI label checks). For a function call, ARM processors automatically put the return address in the `lr` register. Silhouette's shadow stack transformation pass modifies function prologues to store `lr` to the shadow stack. Label-based CFI guarantees an indirect function call can only jump to the beginning of a function, ensuring that attackers cannot use the function prologue to write arbitrary values to the shadow stack.

There are three constructs in C that may cause a compiler to generate forward indirect jumps: indirect tail function calls, large `switch` statements, and computed `goto` statements ("Label as Values" in GNU's nomenclature [36]). Silhouette's CFI forces indirect tail function calls to jump to the beginning of a function. Restricting large `switch` statements and computed `goto` statements is implementation-dependent. We explain how Silhouette handles them in Section 6.3.

5.4 Privileged Code Scanner

As Silhouette executes all code within the processor's privileged mode, Silhouette uses a code scanner to ensure the application code is free of privileged instructions that could be used by an attacker to disable Silhouette's protections. If the scanner detects such instructions, it presents a message to the application developer warning that the security guarantees of Silhouette could be violated by the use of such instructions. It is the application developer's decision whether to accept the risk or modify the source code to avoid the use of privileged instructions.

On ARMv7-M [12], there is only one privileged instruction that must be removed: `MSR` (Move to Special register from Register). One other, `CPS` (Change Processor State), must be rendered safe through hardware configuration. Specifically, the `MSR` instruction can change special register values in ways that can subvert Silhouette. For example, MPU protections on the shadow stack could be bypassed by changing the stack pointer registers (`MSP` or `PSP` on ARMv7-M) to move the shadow stack to a memory region writeable by unprivileged code. The `CPS` instruction can change the execution priority, and the MPU will elide protection checks if the current execution priority is less than 0 and the `HFNMIENA` bit in the MPU Control Register (`MPU_CTRL`) is set to 0 [12]. However,

Silhouette disables this feature by setting the HFNMIENA bit to 1, rendering the CPS instruction safe. A third instruction, MRS (Move to Register from Special register), can read special registers [12] but cannot be used to compromise the integrity of Silhouette.

Finally, as Silhouette provides control-flow integrity, an attacker cannot use misaligned instruction sequences to execute unintended instructions [1]. Therefore, a linear scan of the assembly is sufficient for ensuring that the application code is free of dangerous privileged instructions.

5.5 Improvements with Silhouette-Invert

Swapping a privileged store with a single equivalent unprivileged store introduces no overhead. However, as Section 6.2 explains, Silhouette must add additional instructions when converting some privileged stores to unprivileged stores. For example, transforming floating-point stores and stores with a large offset operand adds time and space overhead.

However, we can minimize store hardening overhead by *inverting* the roles of hardware privilege modes. Specifically, if we can invert the permissions of the shadow stack region to disallow writes from privileged stores but allow writes from unprivileged stores, then we can leave the majority of store instructions unmodified. In other words, this design would allow all stores (except shadow stack writes) to remain unmodified, thereby incurring negligible space and time overhead for most programs. We refer to this variant as *Silhouette-Invert*.

Silhouette-Invert is similar in design to ILDI [22] which uses the Privileged Access Never (PAN) feature on ARMv8-A [8, 14] to prevent privileged stores from writing to user-space memory. Unfortunately, the ARMv7-M architecture lacks PAN support and provides no way of configuring memory to be writeable by unprivileged stores but inaccessible to privileged stores [12]. We therefore reason about the potential performance benefits using a prototype that mimics the overhead of a real Silhouette-Invert implementation. Section 6.5 discusses two potential hardware extensions to ARMv7-M to enable development of Silhouette-Invert.

5.6 Hardware Configuration Protection

As all code on our target system resides within a single address space and, further, as Silhouette executes application code in privileged mode to avoid costly context switching, we must use both the code transformations described above and load-time hardware configurations to ensure that memory safety errors cannot be used to reconfigure privileged hardware state. For example, such state would include the interrupt vector table and memory-mapped MPU configuration registers; on ARMv7-M, most of this privileged hardware state is mapped into the physical address space and can be modified using store instructions [12]. If application code can write to these physical memory locations, an adversary can reconfigure the MPU to make the shadow stack writable or can violate CFI by changing the address of an interrupt handler and then waiting for an interrupt to occur. Therefore, Silhouette makes sure that the MPU prevents these memory-mapped registers from being writable by unprivileged store instructions. As Section 2 explains, the ARMv7-M MPU is automatically configured this way.

```
mov.w ip, #0xe00000 // ip is the intra-procedure
                    // call scratch register
str.w lr, [sp, ip]  // Save lr to mem[sp + ip]
```

Listing 1: Instructions to Update the Shadow Stack

6 Implementation

We implemented Silhouette by adding three new MachineFunction passes to the LLVM 9.0 compiler [45]: one that transforms the prologue and epilogue code to use a shadow stack, one that inserts CFI checks on all computed branches (except those used for returns), and one that transforms stores into STRT, STRHT, or STRBT instruction sequences. Silhouette runs our new passes after instruction selection and register allocation so that subsequent code generator passes do not modify our instrumentation. Finally, we implemented the privileged code scanner using a Bourne Shell script which disassembles the final executable binary and searches for privileged instructions. Writing a Bourne shell script made it easier to analyze code within inline assembly statements; such statements are translated into strings within special instructions in the LLVM code generator. We measured the size of the Silhouette passes and code scanner using SLOCCount 2.26. Silhouette adds 2,416 source lines of C++ code to the code generator; the code scanner is 95 source lines of Bourne shell code.

6.1 Shadow Stack Transformation

Our prototype implements a parallel shadow stack [28] which mirrors the size and layout of the normal stack. By using parallel shadow stacks, the top of the shadow stack is always a constant offset from the regular stack pointer. Listing 1 shows the two instructions inserted by Silhouette in a function's prologue for our STM32F469 Discovery board [64, 66]. The constant moved to the ip register may vary across different devices based on the available address space. Note that the transformed prologue writes the return address into both the regular stack and the shadow stack.

Silhouette transforms the function epilogue to load the saved return address to either pc (program counter) or lr, depending on the instructions used in the original epilogue code. The instructions added by the shadow stack transformation are marked with a special flag so that a later pass (namely, the

store hardening pass) knows that these instructions implement the shadow stack functionality.

Silhouette also handles epilogue code within `IT` blocks [12]. An `IT` (short for If-Then) instruction begins a block of up to 4 instructions called an `IT` *block*. An `IT` block has a condition code and a mask to control the conditional execution of the instructions contained within the block. A compiler might generate an `IT` block for epilogue code if a function contains a conditional branch and one of the branch targets contains a `return` statement. For each such epilogue `IT` block, Silhouette removes the `IT` instruction, applies the epilogue transformation, and inserts new `IT` instruction(s) with the correct condition code and mask to cover the new epilogue code.

6.2 Store Hardening

Silhouette transforms all possible variations of regular stores to one of the three unprivileged store instructions: `STRT` (store word), `STRHT` (store halfword), and `STRBT` (store byte) [12]. When possible, Silhouette swaps the normal store with the equivalent unprivileged store. However, some store instructions are not amenable to a direct one-to-one translation. For example, some store instructions use an offset operand larger than the offset operand supported by the unprivileged store instructions; Silhouette will insert additional instructions to compute the target address in a register so that the unprivileged store instructions can be used. ARMv7-M also supports instructions that store multiple values to memory [12]; Silhouette converts such instructions to multiple unprivileged store instructions. For Store-Exclusive instructions [12], Silhouette adds two `BIC` (bitmasking) instructions before the atomic store to force the address operand to point into the global, heap, or regular stack regions.

Silhouette handles store instructions within `IT` [12] blocks in a similar way to how it handles epilogue code within `IT` blocks. If an `IT` block has at least one store instruction, Silhouette removes the `IT` instruction, applies store hardening for each store instruction within the `IT` block, and adds new `IT` instruction(s) to cover newly inserted instructions as well as original non-store instructions within the old `IT` block. This guarantees store hardening generates semantically equivalent instructions for every store in an `IT` block.

Silhouette sometimes adds code that must use a scratch register. For example, when transforming floating-point store instructions, Silhouette must create code that moves the value from a floating-point register to one or two integer registers because unprivileged store instructions cannot access floating-point registers. Our prototype uses LLVM's `LivePhysRegs` class [51] to find free registers to avoid adding register spill code. This optimization significantly reduces store hardening's performance overhead on certain programs; for example, we observed a reduction from 39% to 4.9% for a loop benchmark. Section 8.3 presents detailed data of our experiments.

Comparison with *u*XOM's Store Transformation There are two major differences between Silhouette's implementation of store hardening and the corresponding store transformation of *u*XOM [44]. First, Silhouette performs store hardening near the end of LLVM's backend pass pipeline (after register allocation and right before the `ARMConstantIslandPass` [48]). We made this choice to avoid situations wherein later compiler passes (potentially added by other developers) either generate new privileged stores or transform instructions inserted by Silhouette's shadow stack, store hardening, and CFI passes. As mentioned above, Silhouette avoids register spilling by utilizing LLVM's `LivePhysRegs` class to find free registers. In contrast, *u*XOM transforms store instructions prior to register allocation to avoid searching for scratch registers. As a consequence, subsequent passes, such as prologue/epilogue insertion or passes added by future developers, must ensure that they do not add any new privileged store instructions. Second, our store hardening pass transforms all privileged stores (sans Store-Exclusives) while *u*XOM optimizes its transformation by eliding transformation of certain stores (such as those whose base register is `sp`) when it is safe to do so. The *u*XOM optimization is safe when used with *u*XOM's security policy but may not be safe if store hardening is used to enforce a new security policy that does not protect the integrity of the stack pointer register. Implementing store hardening and optimization in a single pass makes the compiler efficient. However, by adhering to the Separation of Concerns principle in compiler implementation [15], our code is more easily reused: to use store hardening for a new security policy, one simply changes the compiler to run our store hardening pass and then implements any optimization passes that are specific to that security policy.

6.3 Forward Branch Control-Flow Integrity

Indirect Function Calls With link-time optimization enabled, Silhouette inserts a CFI label at the beginning of every address-taken function. Silhouette also inserts a check before each indirect call to ensure that the control flow transfers to a target with a valid label.

Our prototype uses coarse-grained CFI checks, i.e., the prototype uses a single label for all address-taken functions. We picked `0x4600` for the CFI label as it encodes the Thumb instruction `mov r0, r0` and therefore has no side effect when executed. With the addition of static call graph analysis [46], it is possible to extend the Silhouette prototype to use multiple labels with no increase in runtime overhead.

Forward Indirect Jumps Table 1 summarizes the three types of constructs of C that may cause a compiler to generate a forward indirect jump and how they are handled by Silhouette. The compiler may insert indirect jumps to implement large `switch` statements. LLVM lowers large `switch`

Code Pattern	How Silhouette Handles Them
Large `switch` statement	Compiled to bounds-checked `TBB` or `TBH`
Indirect tail function call	Restricted by CFI
Computed `goto` statement	Transformed to `switch` statement

Table 1: C Code That May Be Compiled to Indirect Jumps

statements into PC-relative jump-table jumps using `TBB` or `TBH` instructions [12]; for each such instruction, LLVM places the jump table immediately after the instruction and inserts a bounds check on the register holding the jump-table index to ensure that it is within the bounds of the jump table. As jump-table entries are immutable and point to basic blocks that are valid targets, such indirect jumps are safe. Tail-call optimization transforms a function call preceding a return into a jump to the target function. Silhouette's CFI checks ensure that tail-call optimized indirect calls jump only to the beginning of a function. The last construct that can generate indirect jumps is the computed `goto` statement. Fortunately, LLVM compiles computed `goto` statements into `indirectbr` IR instructions [50]. Silhouette uses LLVM's existing `IndirectBrExpandPass` [49] to turn `indirectbr` instructions into `switch` instructions. We can then rely upon LLVM's existing checks on `switch` instructions, described above, to ensure that indirect jumps generated from `switch` instructions are safe. In summary, Silhouette guarantees that no indirect jumps can jump to the middle of another function.

6.4 MPU Configuration

Our prototype also includes code that configures the MPU before an application starts. Figure 2 shows the address space and the MPU configuration for each memory region of a Silhouette-protected system on our STM32F469 Discovery board [64,66]. Silhouette uses five MPU regions to prevent unprivileged stores from corrupting the shadow stack, program code, and hardware configuration. First, Silhouette sets the code region to be readable, executable, and non-writable for both privileged and unprivileged accesses. No other regions are configured executable; this effectively enforces W⊕X. Second, Silhouette configures the shadow stack region to be writable only by privileged code. All other regions of RAM are set to be readable and writable by both privileged and unprivileged instructions. Our prototype restricts the stack size to 2 MB; this should suffice for programs on embedded devices.[4] Note that Silhouette swaps the normal positions of the stack and the heap to detect shadow stack overflow: a stack overflow will decrement the stack pointer to point to the inaccessible region near the top of the address space; a trap will occur when the prologue attempts to save the return address there. An alternative to preventing the overflow is to put an inaccessible guard region between the stack and the heap; however, it costs extra memory and an extra MPU configuration region. Finally, Silhouette enables the default background region which disallows any unprivileged reads and writes to address ranges not covered by the above MPU regions, preventing unprivileged stores from writing the MPU configuration registers and the `Peripheral`, `Device`, and `System` regions.

[4] The default stack size of Android applications, including both Java code and native code, is only around 1 MB [6].

6.5 Silhouette-Invert

Our Silhouette-Invert prototype assumes that the hardware supports the hypothetical inverted-design described in Section 5.5, i.e., the MPU can be configured so that the shadow stack is only writable in unprivileged mode. We briefly propose two designs to change the hardware to support the memory access permissions required by Silhouette-Invert.

One option is to use a reserved bit in the Application Program Status Register (`APSR`) [12] to support the PAN state mentioned in Section 5.5. In ARMv8-A processors, PAN is controlled by the `PAN` bit in the Current Program Status Register (`CPSR`) [14]. Currently, 24 bits of `APSR` are reserved [12] and could be used for PAN on ARMv7-M.

The second option is to add support to the MPU. In ARMv7-M, the permission configuration of each MPU region is defined using three Access Permission (AP) bits in the MPU Region Attribute and Size Register (`MPU_RASR`) [12]. Currently, binary value `0b100` is reserved, so one could map this reserved value to read and write in unprivileged mode and no access in privileged mode, providing support to the permissions required by Silhouette-Invert without changing the size of AP or the structure of `MPU_RASR`.

In the Silhouette-Invert prototype, the function prologue writes the return address to the shadow stack using an unprivileged store instruction, and CFI uses regular store instructions to save registers to the stack during label checks; all other store instructions remain unchanged. The MPU is also configured so that the shadow stack memory region is writable in unprivileged mode, and other regions of RAM are accessible only in privileged mode. As configuring memory regions to be writable in unprivileged mode only would require a hardware change, the Silhouette-Invert prototype instead configures the shadow stack region to be writable by both unprivileged and privileged stores. We believe both of the potential hardware changes proposed above would add negligible performance overhead. Section 8 shows that Silhouette-Invert reduces overhead considerably.

6.6 Implementation Limitations

Our Silhouette and Silhouette-Invert prototypes share a few limitations. First, they currently do not transform inline assembly code. The LLVM code generator represents inline

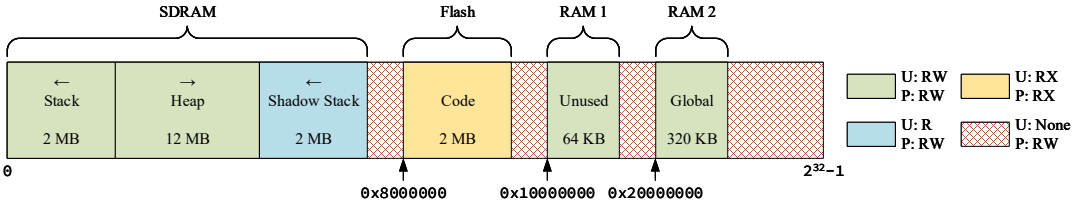

Figure 2: Address Space and MPU Configurations of Silhouette on STM32F469 Discovery Board

assembly code within a C source file as a special "inline asm" instruction with a string containing the assembly code. Consequently, inline assembly code is fed directly into the assembler without being transformed by `MachineFunction` passes. Fortunately, hand-written inline assembly code in applications is rare; our benchmarks contain no inline assembly code. Future implementations could implement store hardening within the assembler which would harden stores in both compiler-generated and hand-written assembly code. Second, our current prototypes do not instrument the startup code or the `newlib` library [56]. These libraries are provided with our development board as pre-compiled native code. In principle, a developer can recompile the startup files and `newlib` from source code to add Silhouette and Silhouette-Invert protections. Third, we have not implemented the "stack-to-heap" promotion (discussed in Section 5) for dynamically-sized stack data. Only one of our benchmarks allocates a variable-length local array; we manually rewrote the code to allocate the variable on the heap. Lastly, we opted not to implement Silhouette's `setjmp`/`longjmp` support, described in Appendix A, as none of our benchmarks use `setjmp` and `longjmp`.

7 Security Analysis

This section explains how Silhouette hinders control-flow hijacking attacks. We first discuss how Silhouette's protected shadow stack, combined with the defenses on forward control-flow, ensure that each return instruction transfers control back to its dynamic caller. We then explain why these security mechanisms provide strong protection against control-flow hijacking attacks.

7.1 Integrity of Return Addresses

Silhouette ensures that functions return control flow to their dynamic callers when executing a return instruction by enforcing three invariants at run-time:

Invariant 1 (I1). A function stores the caller's return address on the shadow stack, or never spills the return address in register `lr` to memory.

Invariant 2 (I2). Return addresses stored on the shadow stack cannot be corrupted.

Invariant 3 (I3). If a function stores the return address on the shadow stack, its epilogue will always retrieve the return address from the correct memory location in the shadow stack, i.e., the location into which its prologue stored the return address.

As the prologue and epilogue code use the stack pointer to compute the shadow stack pointer, maintaining all the invariants requires maintaining the integrity of the stack pointer. Invariants **I1** and **I3** require the function prologue and epilogue to keep the stack pointer within the stack region. Additionally, for **I3**, Silhouette must ensure that the stack pointer is restored to the *correct* location on the stack to ensure that the shadow stack pointer is pointing to the correct return address. For **I2**, besides being inside the stack region, any function call's stack pointer must be guaranteed to stay lower than its frame pointer; otherwise, the valid return addresses on the shadow stack may be corrupted.

To maintain the invariants, Silhouette prevents programs from *loading corrupted values into the stack pointer* by ensuring that application code never spills and reloads the stack pointer to/from memory. In particular, functions that have dynamically-sized stack allocations or that allocate stack memory within a loop may trigger the code generator to spill and reload the stack pointer. As Section 5 explains, Silhouette promotes such problematic `alloca` instructions into heap allocations, ensuring that all functions have constant-sized stack frames and therefore have no need to spill the stack pointer.

The next issue is ensuring that the remaining fixed-size stack memory allocations and deallocations cannot be used to violate the invariants. To prevent *stack overflow*, Silhouette positions the regular stack at the bottom of the address space as Figure 2 shows. If a stack overflow occurs, the stack pointer will point to a location near the top of the address space; if any function prologue subsequently executes, it will attempt to write the return address into an inaccessible location, causing a trap that will allow the TCB to respond to the overflow.

To ensure that stack deallocation does not cause *stack underflow*, Silhouette ensures that deallocation frees the same amount of stack memory that was allocated in the function prologue. Several Silhouette features ensure this. First, the checks on forward control flow ensure that control is never transferred into the middle of a function (as Section 6.3 describes). Second, if **I1**, **I2**, and **I3** hold prior to the underflow, then the shadow stack ensures that a function returns

to the correct caller, preventing mismatched prologues and epilogues. Finally, since the function prologue dominates all code in the function, and since the function epilogue post-dominates all code in the function, the epilogue will always deallocate the memory allocated in the prologue.

In summary, Silhouette maintains **I1** and **I3** by ensuring that the stack pointer stays within the stack region during the function prologue and epilogue and that the epilogue will always deallocate stack memory correctly. Silhouette also ensures that the stack pointer will always be lower than the frame pointer, maintaining **I2**.

7.2 Reduced Attack Surface

Recent work has shown the importance of protecting return addresses to increase the precision, and thus strength, of CFI-based defenses [19, 20, 25, 29, 37]. In particular, without a protected shadow stack or other mechanisms to ensure the integrity of return addresses, CFI with static labels cannot ensure that a function returns to the correct caller at runtime; instead, a function is typically allowed to return to a *set* of possible callers. Attacks against CFI exploit this imprecision.

Most attacks against CFI target programs running on general-purpose systems. Some attacks exploit features specific to certain platforms, and it is not clear if they can be ported to attack embedded devices. For example, Conti et al. [25] showed how to corrupt return addresses saved by unprotected context switches on Windows on 32-bit x86 processors. However, many attacks involve generic code patterns that can likely be adapted to attack CFI-protected programs on embedded systems. We now discuss generic control-flow hijacking code patterns discovered by recent work [19, 20, 29, 37]. As we discuss below, Silhouette is robust against these attacks.

Göktas et al. [37] evaluated the effectiveness of coarse-grained CFI that allows two types of gadgets: *Call-site* (CS) gadgets that start after a function call and end with a return, and *Entry-point* (EP) gadgets that start at the beginning of a function and end with any indirect control transfer. CS gadgets are a result of corrupted return addresses, and EP gadgets stem from corrupted function pointers or indirect jumps if the CFI policy does not distinguish indirect calls and jumps. The authors proposed four methods of chaining the gadgets: CS to CS (i.e., return-oriented programming), EP to EP (call-oriented programming), EP to CS, and CS to EP. Three of these four methods require a corrupted return address. Their proof-of-concept exploit uses both types of the gadgets. Similarly, Carlini et al. [20] and Davi et al. [29] showed how to chain *call-preceded* gadgets (instruction sequences starting right after a call instruction) to launch code-reuse attacks against CFI. As Silhouette prevents return address corruption, only attacks that chain EP gadgets are possible.

Carlini et al. [19] also demonstrated the weaknesses of CFI and emphasized the importance of a shadow stack. They proposed a *Basic Exploitation Test* (BET)—i.e., a minimal program for demonstrating vulnerabilities—to quickly test the effectiveness of a CFI policy. Their work identifies five dangerous gadgets that allow arbitrary reads, writes, and function calls in the BET under a coarse-grained CFI policy. However, all of these are call-preceded gadgets, and Silhouette's protected shadow stack stymies call-preceded gadgets.

Additionally, Carlini et al. [19] demonstrated a fundamental limitation of CFI defenses when used *without* another mechanism to provide return address integrity. Specifically, they showed that even fully-precise static CFI cannot completely prevent control-flow hijacking attacks, concluding that, regardless of the precision of the computed call graph, protection for return addresses is needed.

In summary, with the protection of Silhouette, control-flow hijacking attacks are restricted to only call-oriented programming. Although there are still potential dangers [35], Silhouette significantly reduces the control-flow hijacking attack surface for embedded programs.

8 Experimental Results

Below, we evaluate the performance and code size overhead of our Silhouette and Silhouette-Invert prototypes. We also compare Silhouette to an orthogonal approach, *SSFI*, which uses Software Fault Isolation (SFI), instead of store hardening, to isolate the shadow stack from application code. In summary, we find that Silhouette and Silhouette-Invert incur low runtime overhead (1.3% and 0.3% on average for CoreMark-Pro, respectively) and small increases in code size (8.9% and 2.2%, respectively). In addition, we compare Silhouette with the two most closely related defenses, RECFISH [70] and μRAI [5]; they both protect return addresses of programs running on microcontroller-based embedded devices but leverage different mechanisms than Silhouette.

8.1 Methodology

We evaluated Silhouette on an STM32F469 Discovery board [64, 66] that can run at speeds up to 180 MHz. The board encapsulates an ARM Cortex-M4 processor [9] and has 384 KB of SRAM (a 320 KB main SRAM region and a 64 KB CCM RAM region), 16 MB of SDRAM, and 2 MB of flash memory. As some of our benchmarks allocate megabytes of memory, we use the SDRAM as the main memory for all programs; global data remains in the main SRAM region.

We used unmodified Clang 9.0 to compile all benchmark programs as the baseline, and we compare this baseline with programs compiled by Silhouette, Silhouette-Invert, and SSFI for performance and code size overhead. We also measured the overhead incurred for each benchmark program when transformed with only the shadow stack (SS) pass, only the store hardening (SH) pass, and only the CFI pass. For all experiments, we used the standard `-O3` optimizations, and

	Baseline (ms)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
cjpeg-rose7-...	12,765	1.002	1.004	1.001	1.006	1.003	1.041
core	137,385	1.013	1.002	1.000	1.017	1.015	1.024
linear_alg-...	18,278	1.000	1.010	1.000	1.010	1.000	1.015
loops-all-...	35,241	1.000	1.049	1.000	1.049	1.000	1.016
nnet_test	222,461	1.000	1.013	1.000	1.013	1.000	1.023
parser-125k	9,985	1.004	1.001	1.001	1.005	1.004	1.009
radix2-big-64k	17,270	1.000	1.007	1.000	1.007	1.000	1.019
sha-test	40,725	1.002	1.005	0.999	1.007	1.005	1.046
zip-test	19,955	1.000	1.000	1.000	1.001	1.000	1.006
Min	9,985	1.000	1.000	0.999	1.001	1.000	1.006
Max	222,461	1.013	1.049	1.001	1.049	1.015	1.046
Geomean	—	1.002	1.010	1.000	1.013	1.003	1.022

Table 2: Performance Overhead on CoreMark-Pro

we used LLVM's `lld` linker with the `-flto` option to do link-time optimization.

As Silhouette-Invert requires a hardware enhancement for a fully-functional implementation, the numbers we present here are an estimate of Silhouette-Invert's performance. However, as Sections 5.5 and 6.5 discuss, the hardware changes needed by Silhouette-Invert should have minor impact on execution time and no impact on code size. Therefore, our evaluation of the Silhouette-Invert prototype should provide an accurate estimate of its performance and memory overhead.

We discuss the implementation of SSFI and compare it with Silhouette and Silhouette-Invert in Section 8.5.

8.2 Benchmarks

We chose two benchmark suites for our evaluation: CoreMark-Pro [34] and BEEBS [58]. The former is the de facto industry standard benchmark for embedded processors; the latter has been used in the evaluation of other embedded defenses [24, 44, 70].

CoreMark-Pro The CoreMark-Pro [34] benchmark suite is designed for both low-end microcontrollers and high-end multicore processors. It includes five integer workloads (including JPEG compression and SHA-256) and four floating-point workloads such as fast Fourier transform (FFT) and a neural network benchmark. One of the workloads is a more memory-intense version of the original CoreMark benchmark [33]; note, ARM recommends the use of the original CoreMark benchmark to test Cortex-M processors [10]. We used commit `d15927b` of the CoreMark-Pro repository on GitHub.

The execution time of CoreMark-Pro is reported by benchmarks themselves, which is by calling `HAL_GetTick()` [63] to mark the start and the end of benchmark workload execution and printing out the time difference in milliseconds. We added code before the main function starts to initialize the HAL, set up the clock speed, configure the MPU, and establish a serial output. We run each CoreMark-Pro benchmark in different number of iterations so that the baseline execution time is between 5 to 500 seconds.

	Baseline (ms)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
bubblesort	2,755	1.001	1.247	1.000	1.248	1.000	1.510
ctl-string	1,393	1.015	1.011	0.999	1.027	1.016	1.035
cubic	28,657	1.002	1.002	1.000	1.002	1.001	1.005
dijkstra	40,580	1.002	1.001	1.000	1.003	1.002	1.117
edn	2,677	1.000	1.004	1.000	1.004	1.000	1.058
fasta	16,274	1.000	1.000	1.000	1.000	1.000	1.001
fir	16,418	1.000	1.000	1.000	1.000	1.000	1.021
frac	8,846	1.000	1.003	1.000	1.000	1.000	1.009
huffbench	46,129	1.000	1.005	1.000	1.005	1.000	1.017
levenshtein	7,835	1.005	1.019	1.000	1.207	1.186	1.248
matmult-int	5,901	1.000	1.011	1.000	1.012	1.000	1.048
nbody	124,578	1.000	0.997	1.000	0.997	1.000	1.003
ndes	1,938	1.010	1.008	1.000	1.016	1.011	1.039
nettle-aes	7,030	1.000	1.003	1.000	1.003	1.000	1.111
picojpeg	43,010	1.037	1.057	0.997	1.098	1.037	1.380
qrduino	43,564	1.000	1.036	1.000	1.036	1.000	1.072
rijndael	78,849	1.001	1.008	1.000	1.008	1.005	1.146
sglib-dllist	1,327	1.001	1.006	1.000	1.007	1.001	1.268
sglib-listins...	1,359	1.001	1.000	1.000	1.001	1.001	1.054
sglib-listsort	1,058	1.001	0.999	1.000	1.000	1.001	1.233
sglib-queue	2,135	1.000	1.029	1.000	1.030	1.000	1.122
sglib-rbtree	7,802	1.092	1.017	1.000	1.110	1.093	1.157
slre	4,163	1.031	1.013	1.000	1.045	1.035	1.112
sqrt	55,894	1.000	1.002	1.000	1.006	1.002	1.002
st	20,036	1.002	1.002	1.002	1.002	1.002	1.008
stb_perlin	3,168	1.073	1.052	1.000	1.049	1.073	1.045
trio-sscanf	1,335	1.037	1.006	1.022	1.073	1.063	1.115
whetstone	97,960	1.000	1.001	1.000	1.001	1.000	1.002
wikisort	160,307	1.011	1.013	1.016	1.039	1.029	1.180
Min	1,058	1.000	0.997	0.997	0.997	1.000	1.001
Max	160,307	1.092	1.247	1.022	1.248	1.186	1.510
Geomean	—	1.011	1.018	1.001	1.034	1.019	1.102

Table 3: Performance Overhead on BEEBS

BEEBS The BEEBS benchmark suite [58] is designed for measuring the energy consumption of embedded devices. However, it is also useful for evaluating performance and code size overhead because it includes a wide range of programs, including a benchmark based on the Advanced Encryption Standard (AES), integer and floating-point matrix multiplications, and an advanced sorting algorithm.

The major drawback of BEEBS is that many of its programs either are too small or process too small inputs, resulting in insufficient execution time. For example, `fibcall` is intended to compute the 30th Fibonacci number, but Clang computes the result during compilation and returns a constant directly. To account for this issue, we exclude programs with a baseline execution time of less than one second with 10,240 iterations. We also exclude `mergesort` because it failed the `verify_benchmark()` check when compiled with unmodified Clang. For all the other programs, all of our transformed versions passed this function, if it was implemented. We used commit `049ded9` of the BEEBS repository on GitHub.

To record the execution time of an individual BEEBS benchmark, we wrapped 10,240 iterations of benchmark workload execution with calls to `HAL_GetTick()` [63] and added code to print out the time difference in milliseconds. We also did the same initialization sequence for each BEEBS benchmark as we did for CoreMark-Pro.

8.3 Runtime Overhead

Tables 2 and 3 show the performance overhead that Silhouette and Silhouette-Invert induce on CoreMark-Pro and BEEBS, respectively; overhead is expressed as execution time normalized to the baseline. The **SS** column shows the overhead of just the shadow stack transformation, **SH** shows the overhead induced when only store hardening is performed, and **CFI** shows the overhead of the CFI checks on forward branches. The **Silhouette** and **Invert** columns show the overhead of the complete Silhouette and Silhouette-Invert prototypes, respectively. The **SSFI** column denotes overhead incurred by a version of Silhouette that uses Software Fault Isolation (SFI) in place of store hardening; Section 8.5 describes that experiment in more detail.

Silhouette Performance As Tables 2 and 3 show, Silhouette incurs a geometric mean overhead of only *1.3%* on CoreMark-Pro and *3.4%* on BEEBS. The highest overhead is 4.9% from CoreMark-Pro's loops benchmark and 24.8% from BEEBS's `bubblesort` benchmark. The `bubblesort` benchmark exhibits high overhead because it spends most of its execution in a small loop with frequent stores; to promote these stores, Silhouette adds instructions to the loop that compute the target address. Another BEEBS program with high overhead is `levenshtein`. The reason is that one of its functions has a variable-length array on the stack and that function is called in a loop; Silhouette promotes the stack allocation to the heap with `malloc()` and `free()`. Without this promotion, Silhouette incurs 2.2% overhead on `levenshtein`. Nearly all (8 of 9) of the CoreMark-Pro benchmarks slow down by less than 2%, and 5 programs have less than 1% overhead. For BEEBS, 24 of the 29 programs slow down by less than 5%; 16 programs have overhead less than 1%. Tables 2 and 3 also show that the primary source of the overhead is typically store hardening, though for some programs e.g., `core` and `sglib-rbtree`, the shadow stack induces more overhead due to extensive function calls. CFI overhead is usually negligible because our benchmarks seldom use indirect function calls.

Silhouette-Invert Performance Silhouette-Invert greatly decreases the overhead because it only needs to convert the single privileged store instruction in the prologue of a function to a unprivileged one and leave all other stores unchanged. It incurs only *0.3%* geomean overhead on CoreMark-Pro. Seven of the 9 programs show overhead less than 0.5%. For BEEBS, the geometric mean overhead is 1.9%. When excluding the special case of `levenshtein`, the average overhead is 1.3%. Twenty of the 29 programs slow down by less than 1%. Only three programs, `sglib-rbtree`, `stb_perlin`, and `trio-sscanf`, again, except `levenshtein`, slow down by over 5%, and all of them have very frequent function calls.

	Baseline (bytes)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
Min	51,516	1.005	1.028	1.002	1.036	1.008	1.071
Max	99,156	1.017	1.111	1.094	1.193	1.113	1.315
Geomean	—	1.008	1.068	1.012	1.089	1.022	1.172

Table 4: Code Size Overhead on CoreMark-Pro

	Baseline (bytes)	SS (×)	SH (×)	CFI (×)	Silhouette (×)	Invert (×)	SSFI (×)
Min	30,144	1.003	1.005	1.000	1.009	1.000	1.009
Max	46,108	1.006	1.061	1.013	1.068	1.019	1.201
Geomean	—	1.004	1.018	1.001	1.023	1.005	1.044

Table 5: Code Size Overhead on BEEBS

8.4 Code Size Overhead

Small code size is critical for embedded systems with limited memory. We therefore measured the code size overhead incurred by Silhouette by measuring the code size of the CoreMark-Pro and BEEBS benchmarks. Due to space limitations, we only show the highest, lowest, and average code size increases in Tables 4 and 5. In summary, Silhouette incurs a geometric mean of 8.9% and 2.3% code size overhead on CoreMark-Pro and BEEBS, respectively.

For Silhouette, most of the code size overhead comes from store hardening. As Section 6.2 explains, Silhouette transforms some regular store instructions into a sequence of multiple instructions. Floating-point stores and stores that write multiple registers to contiguous memory locations bloat the code size most. In BEEBS, `picojpeg` incurs the highest code size overhead because an unrolled loop contains many such store instructions, and the function that contains the loop is inlined multiple times. For Silhouette-Invert, because it leaves nearly all stores unchanged, its code size overhead is only 2.2% on CoreMark-Pro and 0.5% on BEEBS.

8.5 Store Hardening vs. SFI

An alternative to using store hardening to protect the shadow stack is to use Software Fault Isolation (SFI) [69]. To compare the performance and code size overhead of store hardening against SFI, we built a system that provides the same protections as Silhouette but that uses SFI in place of store hardening. We dub this system *Silhouette-SFI* (*SSFI*). Our SFI pass instruments all store instructions within a program other than those introduced by the shadow stack pass and those in the HAL. Specifically, our SSFI prototype adds the same `BIC` [12] (bitmasking) instructions as what Silhouette does for Store-Exclusives (discussed in Section 6.2) before each store to restrict them from writing to the shadow stack.

SSFI incurs much higher performance and code size overhead compared to Silhouette. On CoreMark-Pro, SSFI incurs a geometric mean of 2.2% performance overhead, nearly doubling Silhouette's average overhead of 1.3%; on BEEBS,

SSFI slows down programs by 10.2%, three times of Silhouette's 3.4%. Only on one program, the loops benchmark in CoreMark-Pro, SSFI performs better than Silhouette. For code size, SSFI incurs an average of 17.2% overhead on CoreMark-Pro and 4.4% on BEEBS; the highest overhead is 31.5% and 20.1%, respectively, while on Silhouette it is 19.3% and 6.8%. The specific implementation of SFI will vary on different devices due to different address space mappings, so it is possible to get different overhead on different boards for the same program. In contrast, Silhouette's performance overhead on the same program should be more predictable across different boards because the instructions added and replaced by Silhouette remain the same.

8.6 Comparison with RECFISH and μRAI

RECFISH [70] and μRAI [5] are both recently published defenses that provide security guarantees similar to Silhouette but via significantly different techniques. Like Silhouette, they provide return address integrity coupled with coarse-grained CFI protections for ARM embedded architectures. As each defense has distinct strengths and weaknesses, the choice of defense depends on the specific application to be protected. To compare Silhouette with RECFISH and μRAI more directly and fairly, we also evaluated Silhouette with BEEBS and the original CoreMark benchmark using only SRAM and present their performance numbers.

RECFISH [70], which is designed for real-time systems, runs code in unprivileged mode and uses supervisor calls to privileged code to update the shadow stack. Due to frequent context switching between privilege levels, RECFISH incurs higher overhead than Silhouette or μRAI. For the 24 BEEBS benchmarks that RECFISH and Silhouette have in common,[5] RECFISH incurs a geometric mean of 21% performance overhead, and approximately 30% on CoreMark whereas Silhouette incurs just 3.6% and 6.7%, respectively. Unlike the other two defenses, RECFISH patches binaries; no application source code or changes to the compiler are needed.

μRAI [5] protects return addresses, in part, by encoding them into a single reserved register and guaranteeing this register is never corrupted. This approach is more complicated but requires no protected shadow stack. Consequently, μRAI is very efficient for most function calls, incurring three to five cycles for each call-return. However, there are cases, such as calling a function from an uninstrumented library, when μRAI needs to switch hardware privilege levels to save/load the reserved register to/from a safe region, which is expensive.

The μRAI paper [5] reports an average of 0.1% performance overhead on CoreMark and five IoT applications. The μRAI authors observed that one IoT program, `FatFs_RAM`, saw a 8.5% speedup because their transformation triggered the compiler to do a special optimization that was not performed on the baseline code. When accounting for this optimization, μRAI incurred an overhead of 6.9% on `FatFs_RAM` and 2.6% on average for all benchmarks. We measured the performance of CoreMark using Silhouette; the result is 6.7% overhead compared to μRAI's reported 8.1% [5].

Finally, we observe that Silhouette's store hardening is a general technique for intra-address space isolation. Thus, Silhouette can be extended to protect other security-critical data in memory, which Section 9 discusses. In contrast, μRAI only protects a small amount of data by storing it within a reserved register; its approach cannot be as easily extended to protect arbitrary amounts of data. μRAI does rely on SFI-based instrumentation in exception handlers for memory isolation, but our results in Section 8.5 show that store hardening is more efficient than SFI and could therefore be used to replace SFI in μRAI.

9 Extensibility

Although Silhouette focuses on providing control-flow and return address integrity for bare-metal applications, it can also be extended to other use cases. For example, with minimal modification, Silhouette can be used to protect other security-critical data in memory, such as CPI's sensitive pointer store [43] or the kernel data structures within an embedded OS like Amazon FreeRTOS [16].

With moderate modification, Silhouette can also emulate the behavior of running application code in unprivileged mode on an embedded OS. First, the kernel of the embedded OS would need to configure the MPU to disable unprivileged write access to all kernel data. Second, the embedded OS kernel's scheduler would need to disable unprivileged write access to memory of background applications. Third, in addition to store hardening, Silhouette would need to transform loads in the application code into unprivileged loads in order to protect the confidentiality of OS kernel data structures. It would also need to ensure that the embedded OS kernel code contains no CFI labels used by user-space applications. Fourth, the privileged code scanner must be adjusted to forbid all privileged instructions (as opposed to only those that can be used to bypass Silhouette's protections) in application code, forbid direct function calls to internal functions of the kernel, and allow privileged instructions in the embedded OS kernel. Fifth, since the stack pointer of background applications needs to be spilled to memory during context switch, the embedded OS kernel must protect the stack pointer of applications from corruption in order to enforce Silhouette's security guarantee of return address integrity. One simple solution would be storing application stack pointers to a kernel data structure not writable by application code. Finally, system calls require no changes. In ARMv7-M [12], application code calls a system call using the `SVC` instruction, which generates a supervisor call exception. A pointer to the exception

[5] We obtained RECFISH's detailed performance data on BEEBS via direct correspondence with the RECFISH authors.

handler table (which stores the address of exception handler functions) is stored in a privileged register within the `System` region; Silhouette can protect both the `System` region and the exception handler table to ensure that the `SVC` instruction always transfers control to a valid system call entry point. Also, regardless of current privilege mode, exception handlers in ARMv7-M, including the supervisor call handler, will execute in privileged mode and switch the stack pointer to use the kernel stack [12]. Therefore, system calls require no change for Silhouette to work as intended.

10 Related Work

Control-Flow Hijacking Defenses for Embedded Systems Besides RECFISH [70] and μRAI [5], which Section 8.6 discusses, there are several other control-flow hijacking defenses for embedded devices. CFI CaRE [57] uses supervisor calls and TrustZone-M technology, available on the ARMv8-M [13] architecture but not on ARMv7-M, to provide coarse-grained CFI and a protected shadow stack. CFI CaRE's performance overhead on CoreMark is 513%. SCFP [71] provides fine-grained CFI by extending the RISC-V architecture. Unlike Silhouette, SCFP is a pure CFI defense and does not provide a shadow stack. Therefore, it cannot mitigate attacks such as control-flow bending [19] while Silhouette can, as Section 7.2 shows.

Use of Unprivileged Loads/Stores Others [22, 44] have explored the use of ARM's unprivileged loads and stores to provide security guarantees; however, these works differ from Silhouette's store hardening in both implementation and application. *u*XOM [44] transforms regular load instructions to unprivileged ones to implement execute-only memory on embedded systems. Aside from differences in the provided security guarantees—i.e., execute-only memory versus control-flow and return address integrity—these systems differ in how they handle dangerous instructions that could be manipulated to bypass protections. In particular, *u*XOM inserts verification routines before unconverted load/store instructions to ensure that they will not access security-critical memory regions while Silhouette leverages CFI and other forward branch protections to prevent unexpected instructions from being executed. ILDI [22] combines unprivileged loads and stores on the ARMv8-A architecture along with the PAN state and `hyp mode` to isolate data within the Linux kernel—the latter two features are not available on the ARMv7-M systems targeted by Silhouette.

Intra-Address Space Isolation Silhouette protects the shadow stack by leveraging store hardening. Previous work has explored other methods of intra-address space isolation which could be used to protect the shadow stack. Our evaluation in Section 8.5 compares Silhouette to Software Fault Isolation (SFI) [69], so we focus on other approaches here.

ARM Mbed μVisor [7], MINION [42], and ACES [23] enforce memory compartmentalization on embedded systems using the MPU. They all dynamically reconfigure the MPU at runtime but target different scenarios; Mbed μVisor and MINION isolate processes from each other at context switches, and ACES dissects a bare-metal application at function boundaries for intra-application isolation. As discussed previously, isolation that requires protection domain switching is poorly-suited to security instrumentation that requires frequent crossing of the isolation boundaries—such as Silhouette's shadow stack accesses.

ARMlock [72] uses ARM domains to place pages into different protection domains; a privileged register controls access to pages belonging to different domains. ARM domains are only available for CPUs with MMUs [11, 12] and therefore cannot be used in ARMv7-M systems. Additionally, access to ARM domains can only be modified in privileged mode; software running in user-space must context switch to privileged mode to make changes.

Information Hiding Given the traditionally high cost of intra-address space isolation, many defenses hide security-critical data by placing it at a randomly chosen address. This class of techniques is generally referred to as information hiding. For example, EPOXY [24] includes a backward-edge control-flow hijacking defense that draws inspiration from CPI [43]—relying on information hiding to protect security-critical data stored in memory. Consequently, an adversary with a write-what-where vulnerability (as assumed in our threat model) can bypass EPOXY protections.

Fundamentally, information hiding is unlikely to be a strong defense on embedded systems as such systems tend to use only a fraction of the address space (and the memory is directly mapped) which limits the entropy attainable. For example, our evaluation board only has 2 MB of memory for code; if each instruction occupies two bytes, randomizing the code segment provides at most 20 bits of entropy. In contrast, Silhouette's defenses are effective even if the adversary has full knowledge of the memory layout and contents.

Memory Safety Memory safety provides strong protection but incurs high overhead. Solutions using shadow memory [2, 3, 32, 47, 62] may consume too much memory for embedded systems. Other solutions [30, 31, 41, 55, 60] incur too much performance overhead. nesCheck [53] is a memory safety compiler for TinyOS [38] applications which induces 6.3% performance overhead on average. However, nesCheck cannot support binary code libraries as it adds additional arguments to functions. Furthermore, nesCheck's performance relies heavily on static analysis. We believe that, due to their simplicity, the benchmarks used in the nesCheck evaluation are more amenable to static analysis than applications for slightly more powerful embedded systems (such as ours). In

contrast, Silhouette's performance does not depend on static analysis's precision.

11 Conclusions and Future Work

In conclusion, we presented Silhouette: a software control-flow hijacking defense that guarantees the integrity of return addresses for embedded systems. To minimize overhead, we proposed Silhouette-Invert, a system which provides the same protections as Silhouette with significantly lower overhead at the cost of a minor hardware change. We implemented our prototypes for an ARMv7-M development board. Our evaluation shows that Silhouette incurs low performance overhead: a geometric mean of 1.3% and 3.4% on two benchmark suites, and Silhouette-Invert reduces the overhead to 0.3% and 1.9%. We are in the process of opening source the Silhouette compiler and related development tools. They should be available at `https://github.com/URSec/Silhouette`.

We see two primary directions for future work. First, we can optimize Silhouette's performance. For example, Section 7.1 shows that Silhouette ensures that the stack pointer stays within the stack region. Consequently, store instructions using the `sp` register and an immediate to compute target addresses are unexploitable; Silhouette could elide store hardening on such stores. Second, we can use Silhouette to protect other memory structures, such as the safe region used in CPI [43] and the process state saved on interrupts and context switches (like previous work [26] does).

Acknowledgements

The authors thank the anonymous reviewers for their insightful comments and Trent Jaeger, our shepherd, for helping us improve our paper. This work was funded by NSF awards CNS-1618213 and CNS-1629770 and ONR Award N00014-17-1-2996.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Oakland, CA, 2008. IEEE Computer Society.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Security '09, pages 51–66, Montreal, QC, Canada, 2009. USENIX Association.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), November 1996.

[5] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. μRAI: Securing embedded systems with return address integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium*, NDSS '20, San Diego, CA, 2020. Internet Society.

[6] Verifying app behavior on the Android runtime (ART). `https://developer.android.com/guide/practices/verifying-apps-art`.

[7] Mbed μVisor. `https://www.mbed.com/en/technologies/security/uvisor`.

[8] The ARMv8-A architecture and its ongoing development, 2014. `https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/the-armv8-a-architecture-and-its-ongoing-development`.

[9] Arm Holdings. *Cortex-M4 Technical Reference Manual*, March 2010. DDI 0439B.

[10] Arm Holdings. *CoreMark Benchmarking for ARM Cortex Processors: Application Note 350*, July 2013. DAI 0350A.

[11] Arm Holdings. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, May 2014. DDI 0406C.c.

[12] Arm Holdings. *ARMv7-M Architecture Reference Manual*, December 2014. DDI 0403E.b.

[13] Arm Holdings. *ARMv8-M Architecture Reference Manual*, October 2019. DDI 0553B.i.

[14] Arm Holdings. *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile*, March 2020. DDI 0487F.b.

[15] Marc Auslander and Martin Hopkins. An overview of the PL.8 compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, CC '82, pages 22–31, Boston, MA, 1982. ACM.

[16] Amazon FreeRTOS. `https://aws.amazon.com/freertos`.

[17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Survey*, 50(1):16:1–16:33, April 2017.

[18] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, SP '19, pages 985–999, San Francisco, CA, 2019. IEEE Computer Society.

[19] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium*, Security '15, pages 161–176, Washington, DC, 2015. USENIX Association.

[20] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 385–399, San Diego, CA, 2014. USENIX Association.

[21] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, Security '05, pages 177–191, Baltimore, MD, 2005. USENIX Association.

[22] Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. Instruction-level data isolation for the kernel on ARM. In *Proceedings of the 54th Annual Design Automation Conference*, DAC '17, Austin, TX, 2017. ACM.

[23] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. ACES: Automatic compartments for embedded systems. In *Proceedings of the 27th USENIX Security Symposium*, Security '18, pages 65–82, Baltimore, MD, 2018. USENIX Association.

[24] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, SP '17, pages 289–303, San Jose, CA, 2017. IEEE Computer Society.

[25] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, Denver, CO, 2015. ACM.

[26] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, San Jose, CA, 2014. IEEE Computer Society.

[27] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A safe execution environment for commodity operating systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, Stevenson, WA, 2007. ACM.

[28] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '15, pages 555–566, Singapore, Republic of Singapore, 2015. ACM.

[29] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, Security '14, pages 401–416, San Diego, CA, 2014. USENIX Association.

[30] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, Shanghai, China, 2006. ACM.

[31] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, Ottawa, ON, Canada, 2006. ACM.

[32] Baozeng Ding, Yeping He, Yanjun Wu, Alex Miller, and John Criswell. Baggy bounds with accurate checking. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering Workshops*, ISSREW '12, pages 195–200, Dallas, TX, 2012. IEEE Computer Society.

[33] CoreMark: An EEMBC benchmark. https://www.eembc.org/coremark.

[34] CoreMark-Pro: An EEMBC benchmark. https://www.eembc.org/coremark-pro.

[35] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, Denver, CO, 2015. ACM.

[36] GNU Project. Label as values. https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html.

[37] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, San Jose, CA, 2014. IEEE Computer Society.

[38] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '00, pages 93–104, Cambridge, MA, 2000. ACM.

[39] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '15, pages 969–986, San Jose, CA, 2016. IEEE Computer Society.

[40] Yier Jin, Grant Hernandez, and Daniel Buentello. Smart Nest Thermostat: A smart spy in your home. In *Black Hat USA*. 2014.

[41] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, AADEBUG '97, pages 13–26, Linköping, Sweden, 1997. Linköping University Electronic Press; Linköpings universitet.

[42] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Proceedings of the 2018 Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, 2018. Internet Society.

[43] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 147–163, Broomfield, CO, 2014. USENIX Association.

[44] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. *u*XOM: Efficient execute-only memory on ARM Cortex-M. In *Proceedings of the 28th USENIX Security Symposium*, Security '19, pages 231–247, Santa Clara, CA, 2019. USENIX Association.

[45] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization*, CGO '04, Palo Alto, CA, 2004. IEEE Computer Society.

[46] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, San Diego, CA, 2007. ACM.

[47] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*, ISMM '17, pages 36–46, Barcelona, Spain, 2017. ACM.

[48] ARMConstantIslandPass file reference. https://llvm.org/doxygen/ARMConstantIslandPass_8cpp.html.

[49] IndirectBrExpandPass.cpp file reference. https://llvm.org/doxygen/IndirectBrExpandPass_8cpp.html.

[50] LLVM language reference manual. https://llvm.org/docs/LangRef.html.

[51] llvm::LivePhysRegs class reference. https://llvm.org/doxygen/classllvm_1_1LivePhysRegs.html.

[52] Azure Sphere. https://azure.microsoft.com/en-us/services/azure-sphere.

[53] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory safety for embedded devices with nesCheck. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, ASIACCS '17, pages 127–139, Abu Dhabi, United Arab Emirates, 2017. ACM.

[54] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. In *Black Hat USA*. 2014.

[55] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, Portland, OR, 2002. ACM.

[56] Newlib. https://sourceware.org/newlib.

[57] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '17, pages 259–284, Atlanta, GA, 2017. Springer-Verlag.

[58] James Pallister, Simon Hollis, and Jeremy Bennett. BEEBS: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, August 2013.

[59] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.

[60] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Network and Distributed System Security Symposium*, NDSS '04, San Diego, CA, 2004. Internet Society.

[61] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. Security and privacy challenges in industrial Internet of Things. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC '15, pages 54:1–54:6, San Francisco, CA, 2015. ACM.

[62] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, ATC '12, pages 309–318, Boston, MA, 2012. USENIX Association.

[63] STMicroelectronics. *UM1725 User Manual: Description of STM32F4 HAL and LL Drivers*, February 2017. DocID025834 Rev 5.

[64] STMicroelectronics. *RM0386 Reference Manual: STM32F469xx and STM32F479xx Advanced Arm®-Based 32-Bit MCUs*, June 2018. RM0386 Rev 5.

[65] STMicroelectronics. *PM0214 Programming Manual: STM32 Cortex®-M4 MCUs and MPUs Programming Manual*, March 2020. PM0214 Rev 10.

[66] STMicroelectronics. *UM1932 User Manual: Discovery Kit with STM32F469NI MCU*, April 2020. UM1932 Rev 3.

[67] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, San Francisco, CA, 2013. IEEE Computer Society.

[68] Texas Instruments. Hardware abstraction layer code generator for Hercules MCUs, 2019. https://www.ti.com/tool/HALCOGEN.

[69] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, Asheville, NC, 1993. ACM.

[70] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. Control-flow integrity for real-time embedded systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, ECRTS '19, pages 2:1–2:24, Stuttgart, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum füer Informatik.

[71] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. Sponge-based control-flow protection for IoT devices. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy*, EuroSP '18, pages 214–226, London, United Kingdom, 2018. IEEE Computer Society.

[72] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 558–569, Scottsdale, AZ, 2014. ACM.

A Design to Support `setjmp`/`longjmp`

Calls to `setjmp` and `longjmp` can undermine Silhouette's return addresses integrity guarantees because `longjmp` uses

Algorithm 1: Silhouette `setjmp`

```
   Input: A jmp_buf buf
1  foreach entry e in map do
2  |  if e.buf == &buf then
3  |  |  e.{sp, lr, ...} ← {sp, lr, ...};
4  |  |  return 0;
5  |  end
6  end
7  if map.size < map.capacity then
8  |  Insert a new entry {&buf, sp, lr, ...} into map;
9  |  map.size ← map.size +1;
10 |  return 0;
11 else
12 |  Error("Map reached its capacity");
13 end
```

Algorithm 2: Silhouette `longjmp`

```
   Input: A jmp_buf buf
   Input: An integer val
1  buf_entry ← null;
2  foreach entry e in map do
3  |  if e.buf == &buf then
4  |  |  buf_entry ← e;
5  |  |  break;
6  |  end
7  end
8  if buf_entry == null then
9  |  Error("Invalid jmp_buf");
10 end
11 foreach entry e in map do
12 |  if e.sp < buf_entry.sp then
13 |  |  Invalidate e;
14 |  |  map.size ← map.size −1;
15 |  end
16 end
17 {sp, lr, ...} ← buf_entry.{sp, lr, ...};
18 if val == 0 then
19 |  return 1;
20 else
21 |  return val;
22 end
```

a return address from its `jmp_buf` argument which could be located in corruptible global, heap, or stack memory. Applications might also misuse `setjmp` and `longjmp`, such as calling `longjmp` after the function that called `setjmp` with the corresponding `jmp_buf` returns, leading to undefined behaviors exploitable by attackers. Silhouette modifies the implementation of `setjmp` and `longjmp` to support them while maintaining its return address integrity guarantees.

Specifically, Silhouette reserves part of the protected shadow stack region to store a map of active `jmp_buf` records in use by the program. Figure 3 shows the format of a map entry; the address of a `jmp_buf` passed to `setjmp`/`longjmp` serves as a key, and all callee-saved registers plus `sp` and `lr` are values. Algorithms 1 and 2 depict the design of our custom `setjmp` and `longjmp`, respectively. When the application calls `setjmp`, instead of saving the execution context to the application-specified `jmp_buf`, Silhouette's `setjmp` saves it to the map by inserting a new entry or overriding an existing entry, based on the address of `jmp_buf`. If we are inserting a new entry and the number of active `jmp_buf` records reaches the map's capacity, Silhouette's `setjmp` reports an error and aborts the program; this is not a practical problem as we expect the program to have only a few `jmp_buf`s. We can also provide an option for the application developer to specify a desired size of the map. Our store hardening pass will recognize this safe version of `setjmp` and generate regular stores (instead of unprivileged stores) for it to access the map. Saving the execution context in the protected region ensures the integrity of saved stack pointer values and return addresses.

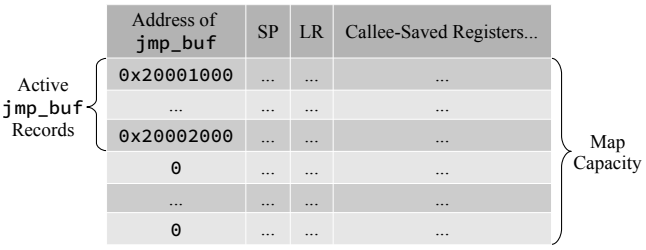

Figure 3: Format of `jmp_buf` Records

Silhouette's `longjmp` checks if the address of the supplied `jmp_buf` matches an entry in the map. If no matched entry is found, either the supplied `jmp_buf` is invalid or the supplied `jmp_buf` has expired due to function returns or a call to `longjmp` on an outer-defined `jmp_buf` (both explained below). In both cases, execution is aborted. If a matched entry is found, Silhouette's `longjmp` first invalidates all entries in the map that have a smaller `sp` value than that of the matched entry; these `jmp_buf`s become expired when the control flow is unwound to an outer call site of `setjmp`. The execution context stored in the matched entry is then recovered.

The remaining case is that, when a function that calls `setjmp` returns, the `jmp_buf`s used in the function and in its callees become obsolete. Silhouette handles this case by inserting code in the epilogue of such functions to invalidate all the map entries whose `sp` value is smaller than or equal to the current `sp` value. This ensures that future calls to `longjmp` do not use obsolete `sp` and `lr` values.