Silhouette: Efficient Intra-Address Space Isolation for Protected Shadow Stacks on Embedded Systems

Jie Zhou\textsuperscript{1}, Yufei Du\textsuperscript{1}, Lele Ma\textsuperscript{1,2}, Zhuojia Shen\textsuperscript{1}, John Criswell\textsuperscript{1}, and Robert J. Walls\textsuperscript{3}

\textsuperscript{1}University of Rochester
\textsuperscript{2}College of William & Mary
\textsuperscript{3}Worcester Polytechnic Institute

Abstract

Embedded systems are increasingly deployed in devices that can have physical consequences if compromised by an attacker—including automobile control systems, smart locks, drones, and implantable medical devices. Due to resource and execution-time constraints, C is the primary programming language for writing both operating systems and bare-metal applications on these devices. Unfortunately, C is neither memory safe nor type safe.

In this paper, we present an efficient intra-address space isolation technique for embedded ARM processors that leverages unprivileged store instructions. Using a compiler transformation, dubbed \textit{store promotion}, which transforms regular stores into unprivileged equivalents, we can restrict a subset of the program’s memory accesses to unprivileged memory while simultaneously allowing security instrumentation to access security-critical data structures (e.g., a shadow stack)—all without the need for expensive context switching.

Using store promotion, we built \textit{Silhouette}: a software defense that mitigates control-flow hijacking attacks using a combination of hardware configuration, runtime instrumentation, and code transformation. Silhouette enforces Control-Flow Integrity and provides an incorruptible shadow stack for return addresses. We implemented Silhouette on an ARMv7-M board and our evaluation shows that Silhouette incurs an arithmetic mean of 9.23% performance overhead and 16.93% code size overhead. Furthermore, we built \textit{Silhouette-Invert}, an alternative implementation of Silhouette, which incurs just 2.54% performance overhead and 5.40% code size overhead, at the cost of a minor hardware change.

1 Introduction

Embedded systems are typically developed in C, meaning they suffer from the same memory errors that have plagued general-purpose systems [23, 33, 63, 65]. Indeed, hundreds of vulnerabilities in embedded software have been reported since 2017\textsuperscript{1}. Unlike most general-purpose systems, however, there is often little distinction between application and kernel code with both running in the same address space and with the same hardware privilege level. Further, exploitation of an embedded system might directly lead to physical consequences in the real world. For example, the control system of a car is crucial to people’s safety; the security and robustness of the programs running on a smart lock is essential to the safety of people’s homes. As these systems grow in importance—both Amazon and Microsoft have recently touted operating systems targeting embedded devices [10, 11]—their security vulnerabilities become increasingly dangerous [48, 60, 67].

Recent work has focused on adapting instrumentation-based software defenses from general-purpose systems to protect embedded code. For example, EPOXY [25] provides an embedded control-flow hijacking defense that adapts ideas from Code-Pointer Integrity (CPI) [51]; RECFISH [79] is an implementation of Control-Flow Integrity (CFI) [15, 21] that targets real-time systems, and µXOM [52] is a scheme for execute-only memory. This work is on-going, and new research is likely to draw upon the extensive existing work on fighting memory safety attacks in general-purpose systems [15, 21, 27, 30, 35, 41, 50, 51, 57, 61, 82, 84].

At the core of many of these software defenses is the need for \textit{intra-address space isolation}. For example, shadow stacks protect return addresses, but shadow stacks themselves reside in the same address space as the exploitable program and must be protected from corruption [22]. Traditional memory isolation techniques, such as \textit{process-based} or \textit{kernel-space} isolation, can be adapted to embedded systems through the use of hardware privilege levels and a system-call-like interface. However, this approach is poorly suited to defenses wherein the security instrumentation requires frequent crossings between protection domains (e.g., a write to the shadow stack for every function call). Sometimes \textit{information hiding} is used to approximate intra-address space isolation as

\textsuperscript{1}The work was done when Lele Ma was a visiting student at the University of Rochester.

\textsuperscript{2}Examples include CVE-2017-8410, CVE-2017-8412, CVE-2018-3898, CVE-2018-16525, CVE-2018-16526, and CVE-2018-19417 [12].
it does not require an expensive context switch. In information hiding, security-critical data structures are placed at a random location in memory under the assumption that it will be difficult for the adversary to guess the exact location [51]. Unfortunately, information hiding itself is poorly suited to embedded systems as most systems have a limited amount of memory, and that memory is directly mapped into the address space—e.g., the board used in this work has just 128 KB of SRAM [72].

In this paper, we present an efficient solution to provide intra-address space isolation on embedded systems. Based on a mechanism we call store promotion\(^2\), our solution creates a logical separation between the code and memory used for security-critical tasks and that used by application code. Unlike kernel-space isolation, store promotion does not require expensive switches between protection domains. Further, unlike the probabilistic protections of information hiding, protections based on store promotion are hardware-enforced. Our design leverages the unprivileged store operations supported by modern embedded ARM architectures. When a program is running in the processor’s privileged mode, these store instructions are executed as if the processor were in unprivileged mode, i.e., the processor checks the user-mode permission bits for the memory regardless of whether the processor is running in privileged mode. The unprivileged store takes the same number of cycles as the equivalent normal store, meaning this switch introduces very little execution time overhead for most types of stores.

We use our intra-address space isolation mechanism as the foundation for a new software defense, Silhouette, that mitigates control-flow hijacking attacks. Silhouette uses a shadow stack [22] to store return addresses, a static program transformation based on store promotion to protect the shadow stack from corruption, and CFI checks [15, 21] to check the validity of function pointers that remain in unprotected memory. We focus on the ARMv7-M architecture [44] and implemented Silhouette on an ARMv7-M development board given the architecture’s popularity and wide deployment; however, our techniques are also applicable to a wide range of ARM architectures, including ARMv5 [43], ARMv7-A [19], and the new ARMv8-M Main Extension [45]. 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 have developed an efficient intra-address space isolation technique dubbed store promotion which can protect memory regions from corruption.
- Using store promotion, we have built a system, Silhouette, which provides embedded systems applications with a protected shadow stack on which applications save their return addresses to protect them from corruption. Additionally, Silhouette uses label-based CFI checks [15, 21] on forward control flow branches (e.g., calls through function pointers).
- We have evaluated Silhouette’s performance overhead and code size overhead and found that Silhouette incurs an arithmetic mean of 9.23% performance overhead and an arithmetic mean of 16.93% code size overhead.
- We prototyped and evaluated the Silhouette-Invert variant and saw additional performance improvements with an average performance overhead measured at 2.54% by arithmetic mean and code size overhead measured at 5.4%.

Importantly, using store promotion for intra-address space isolation would also benefit other defenses. For example, it could be used to isolate the safe stack and sensitive pointer store in defenses based on CPI [51]. Similarly, Silhouette could be used as part of an existing embedded operating system, such as Amazon FreeRTOS [10].

The rest of the paper is organized as follows: Section 2 provides background information on the ARMv7-M architecture. Section 3 introduces the threat model. Section 4 explains how we can provide two protection domains efficiently on ARMv7-M systems. Section 5 presents how we use our intra-address space isolation to build Silhouette. Section 6 describes our prototype implementation. Section 7 presents the experimental results, and Section 8 explains how Silhouette could mitigate several real-world vulnerabilities. Section 9 discusses related work, Section 10 describes future work, and Section 11 concludes.

2 ARMv7-M Architecture

Our work targets the ARMv7-M architecture [44]. We briefly summarize the privilege and execution modes, address space layout, and memory protection features of the ARMv7-M.

2.1 Embedded Applications and Privilege Modes

ARMv7-M supports the execution of both privileged and unprivileged code [44]. 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; FreeRTOS runs application code in privileged mode by default [13]. 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 [14]), linked directly into an application, and runs within its address space.
2.2 Address Space

ARMv7-M is a memory-mapped architecture, lacking support for virtual memory and using a 32-bit address space [44]. 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 flash memory that cannot be modified at runtime. 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) [44]. Since ARMv7-M is a memory-mapped architecture, all of the security-critical registers, such as the MPU control register, are also mapped to the System region.

2.3 Memory Protection Unit

An ARMv7-M-based device can optionally have a Memory Protection Unit [44]. The MPU is a programmable memory protection component that enforces memory access permissions [18,44]. 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. The memory regions configured by the MPU do not need to exactly match the default memory regions described in Section 2.2. 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 [44]. 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 [44]. 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 or the interrupt vector table [44].

3 Threat Model

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. It is quite common for all of this code to run in the same address space, without isolation, and with the same privilege level. For example, under the default configuration of the Amazon FreeRTOS real-time operating system (v1.4.7) all code runs as privileged in ARMv7-M [13]. These embedded idiosyncrasies heavily inform our threat model and the design decisions for Silhouette and Silhouette-Invert.

Our threat model assumes the presence of a strong adversary with the ability to 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 anytime from the application code. The attempted write may fail, for example, if the write targets memory protected by the MPU. We assume the adversary is unable to change program code as that code is stored in read-only flash memory. However, code injection attacks are possible because, by default, embedded systems usually do not enforce a write-xor-execute policy such as Data Execution Prevention [26].

The goal of the adversary is to manipulate control-flow in order to obtain arbitrary code execution. Manipulation of control-flow is achieved by using the aforementioned memory error to overwrite memory (e.g., a return address) such that an adversary-supplied value is loaded into the program counter. Non-control data attacks [24,46] are out of scope of this work.

Silhouette’s goal is two-fold. The first objective is intra-address space isolation, i.e., creating a protected region in memory for the shadow stack and preventing the adversary from overwriting values in that region. The second is ensuring that all control-flow transfers are legal with respect to a pre-computed control-flow graph (CFG). Notice, that Silhouette is not trying to preclude the adversary’s use of the memory error. Instead, Silhouette is trying to prevent that error from resulting in arbitrary code execution for the adversary. For example, the adversary can overwrite the value of any function pointer stored outside of the shadow stack and code regions; however, Silhouette ensures the set of possible targets (i.e., code destinations) is limited to those defined in the CFG.

Another consequence of our threat model is that Silhouette cannot trust register values spilled to memory accessible to the attacker.

We assume that the developer’s compiler toolchain has been modified to include Silhouette. Further, we assume that the hardware abstraction layer (HAL)—which provides a device-specific interface to the hardware—is either compiled separately from the application code or is annotated, allowing Silhouette to forgo transformations on the HAL that might preclude privileged hardware operations. Similarly, library and kernel code is also compiled separately, allowing the
We discuss the security implications of these assumptions in Section 8.

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 four regions (summarized in Section 6.1) of the address space. Our threat model is consistent with past work on defenses against control-flow hijacking that assumes a strong adversary.

4 Intra-Address Space Isolation

Many security enforcement mechanisms rely on intra-address space isolation to protect security-critical state; 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 [22] need a safe region to store copies of return addresses, and CPI [51] needs a protected region of the address space in which 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 user and kernel mode—are often too inefficient for implementing these security mechanisms. Rather than incur the performance penalty of true memory isolation, some defenses settle for hiding the security-critical data structures at random locations in the address space [22, 51].

Defenses for embedded systems face additional challenges: the usable address space is on the order of kilobytes (which limits the total number of possible randomized code layouts), and there are limited entropy sources for generating random numbers.

We devise a new protection method for embedded ARM systems utilizing unique features of ARMv7-M [44]. The ARMv7-M architecture provides 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 user-mode permission bits configured in the memory protection unit 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 for code running in privileged mode. One unprivileged domain contains regular application code and only uses the STRT, STRHT, and STRBT instructions for writing to memory. The second domain is privileged and uses regular store instructions. This method allows us to enforce memory isolation without costly context switching between privilege modes.

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. Techniques such as shadow stacks combined with CFI [15, 21] can prevent such attacks. Second, a trusted code scanner must ensure, prior to loading and executing unprivileged code, that the code contains no system instructions that could be used to modify important program state without the use of a store instruction. For example, an adversary could use the MSR instruction [44] 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.

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 Promotion: The store promotion pass modifies all store instructions—except those used in the shadow stack instrumentation—to use variants that check the unprivileged-mode permission bits [44].

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 misaligned instructions and 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. It ensures the application code is free of privileged instructions that the adversary might seek to manipulate 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. Also note that the HAL 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.

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 [22] 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 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. The instructions added by the shadow stack transformation are marked with a special flag so that a later pass (namely, the store promotion pass) knows that these instructions implement the shadow stack functionality. Once the transformation is complete, the program uses a shadow stack, but the shadow stack is not protected. For that, Silhouette employs the store promotion pass.

5.2 Protection via Store Promotion

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 promotion 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 two mechanisms ensure shadow stack isolation, even if the entire program is executed in the processor’s privileged mode.

Silhouette applies store promotion to all stores within the application code except those used as part of Silhouette’s shadow stack instrumentation—the latter stores 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 to the store promotion pass.

As discussed in Section 3, we assume the HAL code is not compiled with Silhouette and 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 access the System, Device, and Peripheral memory regions. However, the HAL code is also rendered unreachable by the adversary as it lacks the CFI labels added by Silhouette’s control-flow integrity pass.

5.3 Control-Flow Integrity for Forward Branches

Shadow stacks [22] protect the integrity of function returns, but memory safety attacks can still corrupt function pointers. To mitigate such attacks, Silhouette employs label-based CFI checks [15, 21] on indirect function calls and other forward-computed branches. Label-based CFI serves three purposes. First, CFI ensures that a memory safety error cannot jump to and misuse a store that can write to the shadow stack; this ensures that corruption of function pointers does not negate Silhouette’s protection of return addresses. Note that even an imprecise control flow graph suffices as all that is needed is to ensure that indirect function calls jump to the first address of a valid function. Second, label-based CFI prevents application code from jumping into arbitrary locations within the HAL code or from using HAL functions in a return-to-libc-style attack [33]; the HAL is compiled separately and has no CFI labels in its code. This allows the HAL code to enforce policies on how I/O or hardware configuration is performed. Third, label-based CFI allows Silhouette to mitigate code-
reuse attacks that corrupt only function pointers e.g., return- to-libc attacks [33]. Any function whose address is not taken does not need a CFI label, and a precise call graph can further restrict the targets to which an indirect function call can jump.

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. On ARMv7-M [44], there is only one such instruction that must be removed: MRS (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 writable by unprivileged code. The CPS instruction can change a task’s priority, and the MPU will elide protection checks for tasks executing with priority less than 0 if MPU_CTRL_HFNMIENA is set to 0 [44]. However, Silhouette disables this feature by setting MPU_CTRL_HFNMIENA bit to 1, rendering the CPS instruction safe. A third instruction, MRS (Move to Register from Special Register), can read special registers [44] 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 [15]. Therefore, a linear scan of the assembly is sufficient for ensuring that the application code is free of dangerous privileged instructions.

5.5 Improving Performance with Silhouette-Invert

Swapping a privileged store with a single equivalent unprivileged store introduces no additional overhead. However, some privileged stores require additional instructions—e.g., floating point instructions, see Section 6.3—to be added when being converted to an unprivileged store. These additional instructions are the source of store promotion’s overhead. For example, if a program is heavy in floating point operations, then Silhouette’s overhead will reflect the large number of floating point stores that must be transformed.

However, we can remedy these issues and minimize store promotion overhead by inverting the process. In particular, 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 or time overhead for most programs. We refer to this variant as Silhouette-Invert.

Unfortunately, the ARMv7-M architecture does not support making a memory region accessible to unprivileged stores but inaccessible to privileged stores, so we must reason about the potential performance benefits using a hypothetical implementation—see Section 7.1 for our results. The MPU changes needed should be small; for example, an unused bit in the MPU permission bits could be used to invert which permissions apply to privileged mode and unprivileged mode. Such a change should have minimal cost in execution time and cause no changes to instruction encoding.

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 [44]. 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. On ARMv7-M, this is automatically done by hardware (See Section 2.3).

6 Implementation

We implemented Silhouette by adding three new MachineFunction passes [39] to the LLVM 4.0 compiler [40, 53]: 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 instruction patterns. We measured the size of the Silhouette passes and code scanner using Sloc count [81]. Silhouette adds 1,424 source lines of code to the code generator; the code scanner is 95 source lines of Bourne shell code.
6.1 MPU Configuration

Silhouette uses four MPU regions to configure the MPU to prevent unprivileged stores from corrupting the shadow stack, program code, and hardware configuration. First, Silhouette configures the shadow stack region to be non-executable and accessible only by privileged code. Second, all other regions of RAM are set to be non-executable and writable from both privileged and unprivileged instructions. The third MPU region is configured to make the code and exception vectors in the interrupt vector table read-only. The fourth region covers the memory mapped peripherals and is set to execute-never. Finally, the ARMv7-M architecture mandates that all unprivileged accesses to the private peripheral bus (PPB) result in a BusFault error [44]. This protects the MPU configuration registers and the vector table offset register which are memory-mapped in that region.

6.2 Shadow Stack Transformation

Our prototype implements a parallel shadow stack [31] 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. Since our prototype uses a 4 KB stack and 4 KB shadow stack (which fits within the 128 KB memory of our STM32L475 evaluation board [72]), Silhouette can save the return address to the shadow stack with a single store instruction. Using a constant offset to locate the shadow stack does not reduce security; since Silhouette prevents corruption of the shadow stack, there is no need to hide the shadow stack’s location.

Our shadow stack transformation pass inserts a single store instruction to save the return address to the shadow stack. Silhouette transforms the function epilogue to load the saved return address to either the PC (program counter) or LR, depending on the instructions used in the original epilogue code. Silhouette also handles epilogue code within IT blocks [44]. Since the code to load the saved return address from the shadow stack may not fit an IT block that contains other instructions, Silhouette creates a second IT block and moves the epilogue code that loads the return address into the program counter or link register into this new IT block. Silhouette then transforms the code in the new IT block.

6.3 Store Promotion

Silhouette must transform all possible variations of regular stores to one of three types of unprivileged store instructions: STRT (store word), STRHT (store halfword), and STRBT (store byte) [44]. When possible, Silhouette swaps the normal store with the equivalent unprivileged store. However, some store instructions are not amenable to a direct 1-to-1 translation; Silhouette must then insert additional instructions. We describe several cases below which add overhead to Silhouette:

```
1 add sp, #512 // update the base register
2 strt r0, [sp, #0] // do an unprivileged store
3 sub sp, #512 // restore the base register
```

Listing 1: Store Promotion of \texttt{str r0, [sp, #512]}

Large Immediate Operands All three unprivileged store instructions compute the target address by adding a base address stored in a register to an 8-bit immediate value. When a regular store instruction uses an immediate operand larger than 255, Silhouette must add instructions prior to the unprivileged store that compute the target address and place that address into the unprivileged store’s base register. Further, Silhouette must add instructions after the unprivileged store that restore the prior value of the base register. Listing 1 shows an example of store promotion of a store with an immediate operand outside the range of the immediate supported by STRT.

Store Multiple Instructions ARMv7-M also supports a Store Multiple instruction which can write several registers to memory. To promote such a store, Silhouette transforms it into multiple unprivileged store instructions.

Floating-Point Stores When transforming floating point store instructions, Silhouette must create code that moves the floating point value from the floating point registers to one or two integer registers; an unprivileged store then writes the value in the integer register(s) to memory. This transformation spills the integer register(s) to memory, incurring some execution time overhead.

6.4 CFI Transformation

Our prototype implements label-based CFI [15, 21] for indirect forward control-flow transfers using a single label for all targets. Supporting multi-label CFI is straightforward, e.g., we could use a call graph analysis built for the LLVM compiler [54], and it should have similar runtime overhead. Silhouette uses 0x4600 as its CFI label; this sequence encodes the “mov r0, r0” instruction which has no side effects. Silhouette inserts CFI labels at the beginning of every function and every basic block that is a successor of an indirect branch. Silhouette then inserts a CFI check before every indirect branch and indirect function call to ensure that the control flow transfers to a legitimate target.

6.5 Silhouette-Invert

For the Silhouette-Invert variant, our prototype makes the assumption 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. 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. Due to the limited range of immediate offsets in unprivileged store instructions, the shadow stack pass in Silhouette-Invert needs to insert an additional ADD instruction to set the stack pointer to the shadow stack before storing the return address and an additional SUB afterward to restore the stack pointer. The MPU is also configured so that the shadow stack memory region is writable in unprivileged mode, and other regions of RAM are now 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. Finally, as the results in Section 7 show, this variation of Silhouette reduces overhead considerably.

6.6 Limitations

Our Silhouette and Silhouette-Invert prototypes share a few limitations. First, they currently do not instrument inline assembly code. The LLVM code generator represents inline 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. Consequently, the current prototypes rely on developers to use unprivileged stores in the inline assembly code. Future implementations could implement store promotion within the assembler which would promote stores in both compiler-generated and hand-written assembly code.

Second, our current implementations do not instrument the startup code or the C standard library. These libraries are provided with our development board as pre-compiled native code. In principle, a developer can recompile the startup files and the C library from source code to add Silhouette and Silhouette-Invert protections.

Third, since the HAL [71] should use privileged stores, our design assumes that Silhouette is not used to compile the HAL. However, we could not configure our board’s integrated development environment to compile the HAL with a separate compiler. We therefore manually constructed a list of HAL functions that need to use privileged stores and modified Silhouette and Silhouette-Invert so that they do not transform these functions.

7 Experimental Results

We evaluated Silhouette on an STM32L475 board [72] that can run at speeds up to 80 MHz. By default, the board runs at 4 MHz [73]. The board is equipped with a Cortex-M4 processor [18] and has 128 KB of SRAM, 1 MB of Flash memory, 1 KB of instruction cache, and 256 bytes of data cache [72,73]. We used STM32CubeMX [74], an initialization code generator provided by the board manufacturer, to configure the board to run at 80 MHz. We left other options in their default configurations: the instruction and data caches are enabled but the prefetcher is disabled [73]. We also determined that the board has branch prediction disabled.

We chose version commit 049ded9 of the BEEBS benchmark [58,64] to evaluate Silhouette’s and Silhouette-Invert’s performance and code size overhead. We used unmodified Clang 4.0 [38] to compile all the BEEBS test programs as the baseline, and we compare this baseline with programs compiled by Silhouette and Silhouette-Invert. For all experiments, we used the standard -O3 optimizations. For reasons outlined in Section 6.6, the HAL is compiled with the benchmark source code. Other than functions in HAL that writes to protected memory region, the benchmark, along with the HAL, are transformed during compilation to include Silhouette and Silhouette-Invert protections.

As Section 5.5 explains, the hardware changes needed by Silhouette-Invert should have minor impact on execution time and no impact on instruction encoding. Therefore, our evaluation of the Silhouette-Invert prototype should provide an accurate estimate of its performance and memory overheads.

We exclude 13 of the 80 BEEBS programs due to issues with unmodified Clang. These issues fall into three categories. First, Clang failed to compile the program (mergesort, trio-snprintf, trio-sscanf, and wikisort). Second, the program compiled but execution encountered a HardFault (ctl-stack and ctl-vector). Third, Clang optimized away most of the code so that the program performed no real computation (bs, cover, ns, fibcall, janne_complex, newlib-exp and newlib-mod).

We also excluded crc32 as the program failed the verify_benchmark() check, again when compiled with unmodified Clang. This function is provided with the benchmark suite and verifies the correctness of the result computed by the program. Of the remaining 66 programs included in our results, 43 provide a verify_benchmark() function and all 43 report passing results for both unmodified Clang and the two Silhouette variants.

7.1 Performance Overhead Analysis

We measured the performance overhead incurred by each benchmark program when transformed with only the shadow stack pass, only the store promotion pass, only the CFI pass, Silhouette with all three passes, and Silhouette-Invert with its corresponding shadow stack and CFI passes.

To record the execution time of an individual benchmark, we added calls to a timer in the HAL. Specifically, each BEEBS benchmark has a initialise_benchmark() func-
with Silhouette enabled (i.e., negative overheads). We discuss wrapped the loop with pair of calls to HAL_GetTick() which performs the benchmark’s computation. We wrapped variation observed between the sets, that is as all three resulted observation that 47 of the 66 benchmarks show overhead less than 1%, and only 3 programs have overhead higher than 5.18%, and 55 of the 66 benchmarks incur less than 1% overhead. We believe the main reason is that indirect branches are infrequent in BEEBS programs. Shadow Stack Performance The shadow stack instrumentation only incurs an arithmetic mean overhead of 1.06%. We observe that 47 of the 66 benchmarks show overhead less than 1%, and only 3 programs have overhead higher than 10%. The two programs with the highest overhead, tarai and recursion, both have recursive function calls, which explains the high overhead from shadow stack.

CFI Performance CFI incurs almost negligible overhead. The arithmetic mean overhead is 0.14%. The highest overhead is only 5.18%, and 55 of the 66 benchmarks incur less than 1% overhead. We believe the main reason is that indirect branches are infrequent in BEEBS programs.

Store Promotion Performance The primary overhead for most programs comes from store promotion. Store promotion incurs an arithmetic mean of 7.87% overhead. Among the 66 tested programs, 18 programs’ overhead are less than 1%, and 50 programs slow down by less than 10%. Only 10 programs see an overhead greater than 20%.

When a store instruction uses an addressing mode that is also supported for its equivalent unprivileged store, store promotion incurs no overhead. We verified this with a microbenchmark that uses either a regular store or an STRT instruction within a loop that iterates 10 million times; both loops execute in 500 ms. However, as Section 6.3 explains,
transforming a floating-point store usually requires more instructions than an integer store. Three of the 10 programs whose overhead are over 20% use floating-point operations heavily. To verify that the overhead for these benchmarks is primarily caused by promoting floating point stores, we disabled store promotion of all floating-point stores for the three programs. When floating point stores are not promoted to unprivileged stores, their overhead drops from 35.24%, 29.87%, and 22.73% to 17.62%, 1.27%, and 4.55%, respectively.

All the programs that do not have floating-point operations and whose store promotion overheads are over 20% exhibit similar instruction patterns: they have loops containing store instructions for which store promotion must insert additional instructions (such as a pair of arithmetic instructions as discussed in Section 6.3) in addition to changing the store instruction to an unprivileged store. The loops are often quite small; the store instructions consequently comprise a considerable portion of the loop body. For example, one function in the program with the highest overhead (nettle-arcfour) has two loops. The first loop has only one statement which stores an integer into an array element referenced by a pointer. This statement is translated into two stores by the code generator, and both stores use an offset register operand [44]. Transforming this type of store requires inserting one more pair of arithmetic instructions. The other loop also has stores that require auxiliary instructions to perform store promotion.

Full Silhouette Performance Putting the three passes together, Silhouette incurs an overhead of 9.23% by arithmetic mean. The highest overhead is 48.60% from nettle-arcfour. Fifteen of the 66 programs slow down by less than 1%, and 37 programs have overheads within 5%. Only 12 programs show overheads greater than 20%.

Silhouette-Invert Performance Silhouette-Invert incurs an overhead of 2.54% by arithmetic mean. Thirty-nine of 66 programs slow down by less than 1%; 61 programs slow down by less than 10%, and only 2 programs, tarai and recursion, show overheads greater than 20%. The highest overhead is 33.66% from recursion. As Silhouette-Invert does not use store promotion, which incurs the highest overhead on average within the three passes, Silhouette-Invert incurs significantly lower average overhead than Silhouette. In the most dramatic case, the overhead of nettle-arcfour drops from 48.60% with Silhouette to just 0.06% with Silhouette-Invert. Eight programs show higher overhead with Silhouette-Invert compared to Silhouette, likely as a result of the unexpected performance gains from store promotion (discussed further below); other than recursion, these programs show very low or negative overhead from store promotion. For recursion, we attribute the performance difference to the additional arithmetic operations needed in Silhouette-Invert’s shadow stack instrumentation (see Section 6.5) combined with the high number of shadow stack accesses in that benchmark.

Sources of Performance Improvement Four benchmarks show unexpected performance improvements after Silhouette transformation. For fir and insertsort, we attribute the increase to improved cache performance due to changes in code and data layout. To reach this conclusion, we ran both the uninstrumented and Silhouette-protected programs again with the instruction cache and data cache disabled using the LL_FLASH_DisableInstrCache() and LL_FLASH_DisableDataCache() functions from the HAL [71]. With the cache disabled, fir and insertsort’s performance changed from -7.23% and -2.90% to 6.48% and 31.65% respectively. For nsichneu and aha-mont64, we believe the performance increase is due to pipeline effects rather than caching. In particular, both benchmarks continued to show a speedup with the caches disabled. Further, we saw a performance decrease after using the ISB [44] instruction at the beginning of the loop of nsichneu and the loop in the modul64 function in aha-mont64 to flush the pipeline (in addition to disabling the caches). In this configuration, nsichneu changed from -1.51% to 2.92%, and aha-mont64 changed from -0.05% to 3.28%. We therefore conclude that some programs have better pipeline performance across loop iterations even if Silhouette adds more instructions to the code.

7.2 Code Size Overhead Analysis

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 BEEBS benchmarks, the code in the HAL library that is transformed by Silhouette and linked into the benchmark executable, the MPU configuration file, and the common main.c file. As Section 6.6 states, Silhouette currently does not instrument the startup files or the C standard library, so our measurements do not include the sizes of their functions. Due to space limitations, we only show the highest, the lowest, and the average code size increase of the BEEBS benchmarks in Table 2.

Table 2: Code Size Overhead on BEEBS Benchmarks. SS: Shadow Stack; SP: Store Promotion.

|                | Baseline (%) | SS (%) | CFI (%) | SP (%) | Silhouette (%) | Invert (%) |
|----------------|--------------|--------|---------|--------|----------------|------------|
| Average        | 6393.30      | 2.55   | 1.20    | 12.75  | 16.93          | 5.40       |
| Highest        | 3.76         | 5.64   | 19.53   | 23.41  | 7.67           | 5.76       |
| Lowest         | 0.82         | 0.38   | 7.76    | 10.57  | 1.73           | 1.70       |

Table 2: Code Size Overhead on BEEBS Benchmarks. SS: Shadow Stack; SP: Store Promotion.
highest overhead is only 3.76%. The CFI transformation induces negligible overhead; the arithmetic mean is only 1.20%. Silhouette’s store promotion incurs the most space overhead. As Section 6.3 explains, ARMv7-M has many variations of regular store instructions but only supports three unprivileged stores. Therefore, 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 are the most problematic. In comparison, as Silhouette-Invert does not have a store promotion pass, Silhouette-Invert’s average code size is only 5.40%.

8 Security Discussion

As Section 3 explains, in our attack model, the attacker attempts to leverage a write-what-where vulnerability in application code to achieve arbitrary code execution. Silhouette mitigates such attacks using a combination of hardware configuration, runtime instrumentation, and code transformation.

First, Silhouette prevents code injection as it configures the MPU to make code unwritable or because the code is stored in a read-only memory e.g., EEPROM. Second, all writeable memory regions are configured at startup (via the MPU) to be non-executable. These two properties limit an attacker to using code-reuse attacks such as return-to-libc [33] or return-oriented programming [65].

However, Silhouette’s instrumentation (Section 5.3) tightly controls execution of existing application code. Specifically, the CFI instrumentation guarantees that all forward indirect branches in the application code target a valid CFI label. Further, labels are not inserted into HAL code, preventing application code from calling HAL code via corrupted function pointers. Note that this labeling scheme does not preclude an application from directly calling a HAL function.

Silhouette stores return addresses on the shadow stack (Section 5.1) which is isolated from application code by Silhouette’s store promotion pass (Section 5.2). As Section 6.1 describes, Silhouette replaces all stores in application code with unprivileged equivalents and configures the MPU to prevent unprivileged writes to the shadow stack. The shadow stack instrumentation and HAL code still uses privileged stores, but the attacker cannot call these instructions directly due to the aforementioned CFI label checks. Further, any unprivileged stores to the memory-mapped MPU registers will also be prevented, raising a fault (Section 6.1).

Importantly, our current design adds no protections to the HAL. While adding Silhouette protections to most libraries e.g., stdlib, only requires recompiling the library with Silhouette, libraries such as the HAL require privileged memory accesses which are incompatible with store promotion. A defense like EPOXY’s privilege overlays [25] might be more appropriate for protecting libraries like the HAL.

To show how Silhouette can mitigate real world vulnerabilities, we investigated several exemplar vulnerabilities reported on ARM-based embedded devices between 2017 and 2019. For those attacks that leverage a write-what-where vulnerability in application code to achieve arbitrary code execution, Silhouette could mitigate them effectively.

For example, CVE-2018-16525 [5] and CVE-2018-16526 [6] are two buffer overflow vulnerabilities in FreeRTOS [13] which can execute arbitrary code by corrupting the return address on the stack. If compiled and executed by Silhouette, the vulnerable code would use an unprivileged store to write past the end of the buffer. As the attacker would need to overwrite the address on the shadow stack, and because unprivileged stores cannot write to the shadow stack, the attack would cause a trap to occur when it attempts to write to the shadow stack. Many similar vulnerabilities for ARM-based devices exist, such as those on the Yi Home Camera [8, 9], DLink cameras [2–4], and the Contiki-NG OS [7].

Other attacks corrupt function pointers. In CVE-2017-6264 [1], an Android GPU driver performs an out-of-bound read; the value read is then used as a function pointer. An attacker could use this error to divert control flow to start a code-reuse attack. Silhouette’s CFI protections would prevent control flow from jumping to an arbitrary memory location. Instead, the application would only call labeled addresses e.g., functions. Furthermore, since the HAL functions are not labeled, they cannot be used in an attack.

9 Related Work

Several solutions exist for intra-address space isolation. Software Fault Isolation (SFI) [78] adds bit-masking instructions before stores to prevent them from corrupting protected memory regions. SFI requires no special hardware support and works on multiple platforms [68]. Unlike SFI, Silhouette mitigates code reuse attacks that corrupt function pointers as well as return addresses. Furthermore, Silhouette-Invert achieves 2% average overhead compared to fast SFI implementations (e.g., Portable Native Client’s 5% average overhead).

Several hardware mechanisms have been employed for intra-address space isolation on x86. Google’s original Native Client (NaCL) [83] used x86 segmentation [47]. Secure Virtual Architecture (SVA) [29], HyperSafe [80], and Nested Kernel [32] stored data needing protection within read-only pages; authorized code disables the MMU’s write-protection enforcement by modifying control register 4 [47] to write to the data when needed. While this approach can be relatively efficient, recent work [37] has shown that this method incurs overhead due to serializing instructions. LOTRx86 [55] leverages rings 1 and 2 on the x86 architecture to create an additional privileged execution layer. ERIM [77] and Hodor [42] use the x86 Memory Protection Key (MPK) feature [47] to protect memory in user-space applications; like Silhouette, they scan code to ensure that instructions that could under-
mine their security do not appear within the program’s code segment. Apparition [37] uses the Intel MPX bounds checking hardware [47] to implement hardware-accelerated SFI.

ARM lacks segmentation, a write-protect bit, and bounds-checking hardware. However, older ARM processors have a feature similar to Intel MPK. ARMLock [85] uses ARM domains [19] 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 [19, 44] 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.

ASLR [20, 76] randomizes the location of code to mitigate code reuse attacks. \( \mu XOM \) [52], which independently discovered how to use ARMv7-M’s unprivileged loads and stores to protect memory, prevents buffer overread attacks [75] from learning the code layout. However, ASLR requires sufficient entropy to mitigate brute-force attacks [70]. Embedded systems have limited address space which limits the entropy attainable. For example, our evaluation board only has 1 MB of memory for code; if each instruction occupies two bytes, randomizing the code segment provides at most 19 bits of entropy. In contrast, Silhouette’s defenses are effective regardless of available entropy for code layout.

Memory safety provides strong protection but incurs high overhead. Solutions using shadow memory [16, 17, 36, 57, 69] may consume too much memory for embedded systems. Other solutions [34, 35, 49, 62, 66] incur too much performance overhead. \( \text{nesCheck} \) [59] is a memory safety compiler for TinyOS [56] applications which induces 6.3% performance overhead on average. However, \( \text{nesCheck} \) cannot support binary code libraries as it adds additional arguments to functions. Furthermore, \( \text{nesCheck} \)’s performance relies heavily on static analysis. We believe that, due to their simplicity, the benchmarks used in the \( \text{nesCheck} \) 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 precise static analysis.

Several recent studies have proposed control-flow hijacking defenses for embedded systems. RECFISH [79] uses different privilege modes and supervisor calls to provide a protected shadow stack for real-time ARM systems and incurs higher overhead than Silhouette (approximately 30% on the CoreMark benchmark suite). EPOXY [25] includes a backward-edge control-flow hijacking defense that draws inspiration from CPI [51]. Unlike Silhouette and RECFISH, EPOXY does not use hardware-enforced intra-address space isolation to protect security critical data; consequently, an adversary with a write-what-where vulnerability (as assumed in the Silhouette threat model) can bypass EPOXY protections. The performance of RECFISH and the security of EPOXY would benefit from using store promotion.

10 Future Work

We see three primary directions for future work. First, we can optimize store promotion by alleviating register spills and by experimenting with selectively using SFI [68, 78] on stores that perform poorly when promoted, e.g., multi-register stores. Second, we use store promotion to protect other memory structures, such as the process state saved on interrupts and context switches, in order to provide full CFI protection to embedded kernels—ideally at a lower cost than KCoFI [28] and RECFISH [79]. We can also use store promotion to implement CPI [51]. Finally, we will build prototype hardware of Silhouette-Invert which should provide fast protection while maintaining ARMv7-M’s dense instruction encoding.

11 Conclusions

We presented Silhouette: a software defense for embedded systems which uses store promotion to provide an incorruptible shadow stack and instrumentation to guarantee control-flow integrity. To minimize overhead, we proposed Silhouette-Invert, a system which provides the same level of protection as Silhouette with significantly lower overhead at the cost of a minor hardware change. We implemented Silhouette for an ARMv7-M development board. Our evaluation shows that Silhouette incurs an arithmetic mean of 9.23% performance overhead and 16.93% of code size overhead, and Silhouette-Invert incurs an arithmetic mean of 2.54% performance overhead and 5.40% code size overhead.

References

[1] CVE-2017-6264. Available from MITRE, CVE-ID CVE-2017-6264., February 2017.
[2] CVE-2017-8410. Available from MITRE, CVE-ID CVE-2017-8410., May 2017.
[3] CVE-2017-8412. Available from MITRE, CVE-ID CVE-2017-8412., May 2017.
[4] CVE-2017-8416. Available from MITRE, CVE-ID CVE-2017-8416., May 2017.
[5] CVE-2018-16525. Available from MITRE, CVE-ID CVE-2018-16525., September 2018.
[6] CVE-2018-16526. Available from MITRE, CVE-ID CVE-2018-16526., September 2018.
[7] CVE-2018-19417. Available from MITRE, CVE-ID CVE-2018-19417., November 2018.
[8] CVE-2018-3898. Available from MITRE, CVE-ID CVE-2018-3898., January 2018.
[9] CVE-2018-3899. Available from MITRE, CVE-ID CVE-2018-3899., January 2018.

[10] Amazon FreeRTOS. https://aws.amazon.com/freertos/, 2019.

[11] Azure Sphere. https://azure.microsoft.com/en-us/services/azure-sphere/, 2019.

[12] Common Vulnerabilities and Exposures List. https://cve.mitre.org/cve/, 2019.

[13] FreeRTOS real time operating system. http://www.freertos.org/, 2019.

[14] Hardware abstraction layer code generator for hercules mcus. http://www.ti.com/tool/HALCOGEN, 2019.

[15] Martin 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.

[16] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In SP ’08: Proceedings of the 2008 IEEE Symposium on Security and Privacy, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[17] 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 Eighteenth Usenix Security Symposium, August 2009.

[18] Arm Holdings. Cortex-M4 Technical Reference Manual, 2010. Revision r0p1.

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

[20] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In Proceedings of the 12th Conference on USENIX Security Symposium, volume 12 of SSYM’03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[21] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. ACM Comput. Surv., 50(1):16:1–16:33, April 2017.

[22] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[23] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In 23rd USENIX Security Symposium (USENIX Security 14), pages 385–399, San Diego, CA, 2014. USENIX Association.

[24] 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 - Volume 14, Baltimore, MD, 2005. USENIX Association.

[25] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer. Protecting Bare-Metal Embedded Systems with Privilege Overlays. In 2017 IEEE Symposium on Security and Privacy (SP), pages 289–303, May 2017.

[26] Microsoft Corporation. Data Execution Prevention, May 2018. https://docs.microsoft.com/en-us/windows/desktop/memory/data-execution-prevention.

[27] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7, SSYM’98, pages 5–5, 1998.

[28] 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, 2014.

[29] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory safety for low-level software/hardware interactions. In Proceedings of the 18th USENIX Security Symposium, Security’09, pages 83–100, 2009.

[30] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP ’07, pages 351–366, 2007.

[31] 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, ASIA CCS ’15, pages 555–566, 2015.

[32] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege
Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 191–206, Istanbul, Turkey, 2015.

[33] Solar Designer. Return-into-libc attack, August 1997. https://insecure.org/sploits/linux.libc.return.lpr.sploit.html.

[34] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the International Conference on Software Engineering (ICSE)*, Shanghai, China, May 2006.

[35] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[36] Baozeng Ding, Yeping He, Yanjun Wu, A. Miller, and J. Criswell. Baggy bounds with accurate checking. In *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012.

[37] Bingfei Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding software from privileged side-channel attacks. In *Proceedings of the 27th USENIX Security Symposium*, Security’18, pages 1441–1458, 2018.

[38] LLVM Developer Group. Clang: a c language family frontend for llvm, 2019. https://clang.llvm.org/.

[39] LLVM Developer Group. LLVM MachineFunctionPass Class, 2019. http://llvm.org/docs/WritingAnLLVMPass.html#the-machinefunctionpass-class.

[40] LLVM Developer Group. The LLVM Compiler Infrastructure, 2019. http://llvm.org/.

[41] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX*, 1992.

[42] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[43] Arm Holdings. *ARM Architecture Reference Manual*. Arm Holdings, 2005. ARM DDI 0100I.

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

[45] Arm Holdings. *ARMv8-M Architecture Reference Manual*. Arm Holdings, 2019. ARM DDI 0553B.f.

[46] 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 *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[47] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*, January 2019. Order Number: 325462-069US.

[48] Yier Jin, Grant Hernandez, and Daniel Buettel. Smart Nest thermostat - A smart spy in your home. In *Black Hat USA*, 2014. https://www.blackhat.com/docs/us-14/materials/us-14-Jin-Smart-Nest-Thermostat-A-Smart-Spy-In-Your-Home.pdf.

[49] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[50] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring Code Safety Without Runtime Checks for Real-time Control Systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES ’02, pages 288–297, 2002.

[51] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI’14*, pages 147–163, 2014.

[52] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoun Lee, Yeonggil Cho, and Yunheung Paek. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 231–247, Santa Clara, CA, August 2019. USENIX Association.

[53] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, Mar 2004.

[54] Chris Lattner, Andrew Leftherlath, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings
of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’07), San Diego, California, June 2007.

[55] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS ’18, pages 1441–1454, New York, NY, USA, 2018. ACM.

[56] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In Ambient intelligence, pages 115–148. Springer, 2005.

[57] Zhengyang Liu and John Criswell. Flexible and efficient memory object metadata. In Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, pages 36–46, New York, NY, USA, 2017. ACM.

[58] mageec. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms, 2019. https://github.com/mageec/beebs.

[59] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory safety for embedded devices with nescheck. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS ’17, pages 127–139, New York, NY, USA, 2017. ACM.

[60] Charlie Miller and Chris Valasek. A Survey of Remote Automotive Attack Surfaces. In Black Hat USA, 2014. http://illmatics.com/remote%20attack%20surfaces.pdf.

[61] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’09, pages 245–258, 2009.

[62] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In ACM SIGPLAN Notices, volume 37, pages 128–139. ACM, 2002.

[63] Aleph One. Smashing the Stack for Fun and Profit. Phrack, 7, November 1996. http://www.phrack.org/issues/49/14.html [Online; accessed 21-April-2019].

[64] James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. arXiv preprint arXiv:1308.5174v2, August 2013.

[65] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., 15(1):2:1–2:34, March 2012.

[66] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In In Proceedings of the Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, 2004.

[67] 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, 2015.

[68] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In Proceedings of the 19th USENIX Security Symposium, Security’10, pages 1–11, Washington, DC, 2010. USENIX Association.

[69] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pages 309–318, Boston, MA, 2012. USENIX.

[70] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS ’04, pages 298–307. ACM, 2004.

[71] STMicroelectronics. Description of STM32L4/L4+ HAL and low-layer drivers, September 2017. UM1884 Rev 7.

[72] STMicroelectronics. Discovery kit for IoT node, multi-channel communication with STM32L4, March 2018. UM2153 Rev 4.

[73] STMicroelectronics. STM32L4x5 and STM32L4x6 advanced Arm®-based 32-bit MCUs, April 2018. RM0351 Rev 6.

[74] STMicroelectronics. STM32Cube MCU & MPU Packages, 2019. https://www.st.com/en/embedded-software/stm32cube-mcu-mpu-packages.html.

[75] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the Memory Secrecy Assumption. In Proceedings of the 2nd European Workshop on System Security (EUROSEC), pages 1–8, Nuremberg, Germany, 2009.
[76] The PaX Team. ASLR. http://pax.grsecurity.net/docs/aslr.txt, 2003. [Online; accessed 06-April-2019].

[77] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In 28th USENIX Security Symposium (USENIX Security 19), pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[78] 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.

[79] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. Ward. Control-flow integrity for real-time embedded systems. In 31st Conference on Real-Time Systems (ECRTS’19), July 2019.

[80] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P), pages 380–395, May 2010.

[81] David A. Wheeler. SLOCCount Version 2.26, 2004.

[82] Wei Xu, Daniel C. DuVarney, and R. Sekar. An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs. In Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT ’04/FSE-12, pages 117–126, 2004.

[83] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In Proceedings of the 30th IEEE Symposium on Security and Privacy, SP’09, pages 79–93, 2009.

[84] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’19, pages 631–644, 2019.

[85] 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, New York, NY, USA, 2014. ACM.