Watchmen: Utilizing Data Watchpoints for Shadow Stacks in Embedded Systems

by

Sagar Mohan

May 2024

A thesis submitted to the

Faculty of the Graduate School of

the University at Buffalo, The State University of New York

in partial fulfilment of the requirements for the

degree of

Master of Science

Department of Computer Science and Engineering

To my family and friends

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my thesis supervisor, Dr Ziming Zhao, of the Department of Computer Science and Engineering at the University at Buffalo. His invaluable guidance, continuous support, and immense knowledge were instrumental throughout the research of this thesis. Equally, I would also like to thank Dr. Xi Tan for her constant support and guidance throughout this project. Additionally, I would like to thank Dr. Hongxin Hu, for their valuable feedback during the my defense.

I am also grateful to Xi Tan for their support during the implementation of the project, spending time understanding my work and providing insightful feedback which helped me move forward with the research.

I would like to thank my friends who supported me through this project. Last but not least, I would like to thank my my parents, for their unconditional love and support throughout my life. Thank you.

Table of Contents

List of Tables

List of Figures

Abstract

From smart appliances to other Internet-of-Things (IoT) devices, there are billions of microcontrollers in use today. The C programming language has an entrenched reputation within such embedded systems. This is mainly due to the requirement of such systems for low-latency and real-time operations – all of which `C` can provide. However, due to its proximity to the hardware, `C` extends generous flexibility towards the programmer, which in itself is highly favorable in embedded systems. But, as a consequence, this leads to various security issues, the chief among them being control-flow integrity (CFI), wherein a function callee should always return to the legitimate function caller. There has been a lot of work in this space, especially for microcontrollers. Other works ensure `CFI` while protecting their security mechanisms using existing hardware features like the Memory Protection Unit (MPU).

We propose a novel CFI protection mechanism that uses an existing hardware feature called the Data Watchdog and Trace Unit (DWT) present on Arm Cortex-M microcontrollers. We use a traditional shadow stack to protect against corruption of return addresses, ensuring backward-edge integrity. This shadow stack is protected by the DWT Unit with compile-time instrumentation using the LLVM compiler framework. We analyze our implementation and demonstrate the security of our mechanism with acceptable performance and memory overheads.

Chapter1

Introduction: Security is a State of Mind

The security of microntrollers is crucial in this day and age. ARM processors, which take up a significant portion of the semiconductor market, is often understated. For example, Tom's Hardware reported that in the fourth quarter of 2020 alone, more than 6 billion Arm-based chips were produced [1]. Other major players in this space, such as Intel and AMD, unlike ARM, don't publicly report units sold; but, given their revenue disclosures, it is possible to approximate their numbers in units. Gartner estimated that Intel sold 275 Million units in 2020 [2]. While that is stll a huge number, it's nowhere close to ARM. Therefore, owing to ARM's current hegemony, securing systems which utilize its processors is more important than ever.

Fortunately, security researchers and academics, have strived to formulate and implement low-cost security mechanisms to improve the security of these devices. However, the security mechanisms proposed have not yet been implemented in real-world devices due to a various reasons ranging from these mechanisms having high-performance overhead, or high memory overhead or imposing restrictions in number of available hardware features such as registers, among others. As such, there is still ongoing research in designing an effective, low-cost security mechanism for embedded devices.

When programming, different pieces of code may need to be executed in different orders. This implies there is a flow of execution, and attacks might want to disrupt this flow, to hijack it executing their own code. Ensuring this doesn't happen is the objective of control-flow integrity (`CFI`). Guaranteeing that functions, when called, always return to their legitimate caller, has been a focus of study since the seminal paper on it was published almost 20

years ago [3]. Owing to its importance, there has been much research on the topic within embedded systems [4], with each mechanism having its strengths and weaknesses.

However, from our observations detailed in Chapter 2: Related Work, we see the need for a different way of ensuring backward-edge integrity. In this document, we introduce a novel control-flow protection mechanism that disallows an attacker from hijacking the execution of a program by ensuring the legitimacy of the return address of each protected function. This is achieved by utilizing hardware components present in ARMv7 and ARMv8 architecture boards called the "Data Watchdog (DWT)" to dynamically monitor memory regions which store copies of the return address and other regions that are essential to the monitoring-hardware components.

The document is arranged as follows: Chapter 2 references past work related to our novel mechanism. Chapter 3 provides the background information required to understand our design. Chapter 4 elaborates on the design of the proposed mechanism. Chapter 5 gives details of the implementation of the design. Chapter 6 gives the data from the evaluation. Finally, we give our closing arguments in Chapter 7: Conclusion.

Chapter2

Related Work: Back to the Future

CFI has been in the forefront of execution-flow protection since the 1990s. In particular, backward-edge protections have been in use since the early 2000s, for example, stack canaries [5]. The stack canary/stackguard is a common backward-edge protection mechanism utilized by most industry-standard compilers like LLVM/Clang, and GCC [6]. Likewise, different implementations guaranteeing CFI exist such as Control Flow Guard from Microsoft [7] and Indirect Function-Call Checks by Google [8].

However, CFI for embedded systems is still relatively *immature*. While many RTOS and related firmware implement stack canaries, they do so in an insecure way which allows easy bypasses [9]. Forward-edge integrity is still largely absent on the popular RTOS like FreeRTOS and MbedOS.

CFI for baremetal embedded systems are usually classified based on what mechanism they use to ensure integrity. Broadly, the classifications of the mechanisms are: 1) conventional shadow stacks, 2) register-based shadow stacks, and 3) architecture-based hardware.

Silhouette [10] is a forward and backward-edge CFI mechanism. It offers compile time instrumentation which enables backward-edge protection using parallel shadow stacks and course-grained forward-edge protection using labelling. Silhouette uses the MPU to protect it's shadow stack and modifies all store instructions to unprivileged instructions while only allowing the prologue and epilogue to use privilege instructions.

There are proposals to CFI which do not use conventional shadow stacks. One of these techniques is register-based shadow stacks. There are two significant contributions to this. One is the Zipper Stack [11] and the other is μRai [12] proposes a register-based backward-

edge CFI, since the shadow stack requires hardware isolation.

The other kind of CFI proposed for embedded systems is using processor architecture extensions, for example, ARM TrustZone. TrustZone splits instructions into secure and non-secure mode with each mode having it's own memory region. CFIcare [13] uses the TrustZone to secure the shadow stack. It replaces all function calls with supervisor (`svc`) calls that passes on a parameter to the *Branch monitor* which runs in the privileged context. This branch monitor then uses the parameter to find the valid return address for that supervisor call. In their evaluation, their maximum performance overhead was 513%.

From these previous works we observe that:

1. Most implementations use a form of the shadow-stack to ensure backward-edge integrity during execution.

2. They use the MPU or another architecture-based feature like ARM TrustZone to protect the shadow stack

3. They require reserving a register for special use like a status register or pointer resgister.

From these observations, we propose an alternative solution that:

1. Utilizes a compact shadow stack that utilizes less memory. This is an important distinction in memory-contrained embedded systems. The performance overhead is a trade-off with less memory used for the shadow stack.

2. Propose the use of the DataWatchdog Unit to monitor memory regions instead of the MPU or TrustZone. There could be benefit here since in armv7-M, the MPU can only monitor 8 memory regions, thereby freeing up use of the MPU for other purposes and TrustZone requires making SVC calls which have some overhead due to context switching from non-secure to secure.

3. Utilize unused DWT registers as General Purpose Registers (`GPR`s) for instrumentation thereby not needing to reserve any GPR.

Chapter3

Background: The long ARM of the law

3.1 Embedded Systems

A microcontroller unit is a small computer with just one integrated circuit. It can contain one or more CPUs, one memory region and input/output interfaces. Microcontrollers are generally used in embedded applications which utilize very low resources but require real-time processing.

The history of MCUs goes back more than fifty years, however, the modern variants of what is considered MCUs are about 2 decades old. While MCUs are as old as regular performance computers, their security has been comparatively overlooked. Considering how many billions of microcontrollers are in use today, it begs the question: to what extent can we trust these devices?

3.2 ARM: an Overview

Advanced RISC Machines, or ARM, are processors designed by ARM Ltd. The ARM processors belong to a family of processors called the Reduced Instruction Set Computing (RISC). Unlike other CPU juggernauts like Intel and AMD, ARM does not manufacture their own processors, they only design the Instruction Set Architecture and license other companies (like STMicroelectronics, Nucleo to name a few) to manufacture them with some minor modifications [14]

ARM provides multiple families of processors, one popular such family is the Cortex

processors, which are broadly differentiated in three groups:

- **Cortex-M**: the microcontroller series, intended for low-power, low-latency operations.
- **Cortex-R**: real-time embedded series, intended for high-availability operations.
- **Cortex-A**: for high-performance applications, usually seen in general-purpose application processors like that of Apple computers. [15].

The ARM Architecture is defined in the ARM Architectural Reference Manual ("ARM ARM" for short). In this thesis, we mainly focus on the microcontroller profile version 7, aptly named, ARMv7-M where the "M" stands for microcontroller. These processors are designed for embedded systems with a focus on deterministic real-time execution. Its features include [16]:

1. Provides a simple pipeline design for use in multiple markets and applications.
2. High determinism in operations with minimal latency due to short pipelines.
3. Great support for C/C++ targets.
4. Provides excellent debug and profiling support for event-driven systems.
5. Provides Thumb2 technology which combines 16-bit and 32-bit instructions for code density and performance.
6. Memory Protection Unit (MPU) for memory access control.

Likewise, our implementation can be extended to support ARMv8-M, in fact, the security mechanism becomes simpler due to improvements of v8 over v7. Armv8-M builds upon ARMv7-M, maintaining backwards compatability, with some notable differences, i.e, improvements [16]:

1. An improved, but optional, MPU programmer's model.

2. Stack pointer limit checking.

3. Improvements in the support for multi-processing.

4. Better support for the DWT unit.

3.2.1 ARMv7-M Cortex-M4 Architecture

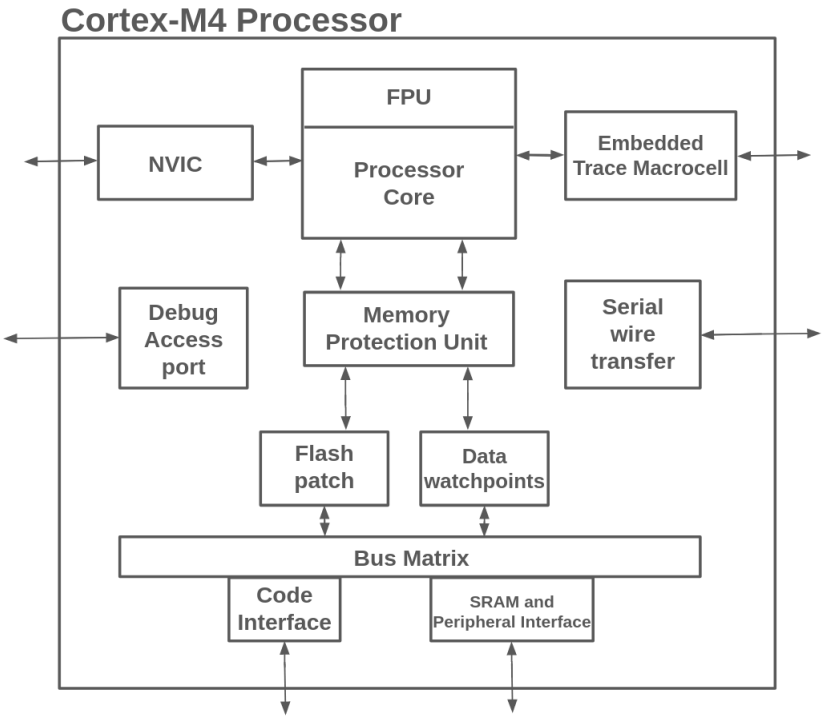

Figure 3.1: STM32 Cortex-M4 Block Diagram

A simplified block diagram of the Cortex-M4 architecture is shown in figure 3.1. The processor is designed by ARM, and connected to it are peripherals manufactured by chip companies like STMicro. The types of peripherals connected to the chip varies based on the board's design.

3.2.1.1 Nested Vector Interrupt Controller

ARMv7-M provides low latency exception and interrupt handling due to the close integration of the Nested Vector Interrupt Controller (NVIC). It provides configurable interrupt

handling. This NVIC in ARMv7-M architecture supports 496 interrupts. The number of external interrupt lines supported is System-on-Chip (SoC) dependent. All NVIC interrupts have a programmable interrupt priority, which is achieved by using an exception number with each priority.

In our implementation, we utilize the NVIC to set the Debug Monitor Exception, which is triggered by the Data Watchpoint and Trace unit (DWT).

3.2.1.2 Data Watchpoint and Trace Unit (DWT)

The DWT supports a wide range or features, the support of each is implementation defined. The features include [16]:

- Watchpoints, where the processor enters the debug state or trigger a DebugMonitor exception
- Data tracing
- For use by an external resource through signalling, like an ETM
- PC value tracking
- Cycle count matching
- Exception trace
- Performance profiling counters

3.2.1.3 Memory Protection Unit

Some Cortex M3 and M4 MCUs have a feature called the Memory Protection Unit (MPU). It is a programmable hardware unit used to define access to memory permissions for different memory regions [17]. In Cortex-M4, the MPU has eight programmable memory regions where each region has its own starting addresses, sizes, and settings.

3.2.2 ARM Processor Modes

In conventional computing, i.e, `x86/x64`, there are two modes of operations, **kernel mode** and **user mode**. The kernel mode has access to all system peripherals and system memory, among other things. Therefore, the Operating System (OS), which needs access to everything, runs in kernel mode and the applications run in user mode. Since applications run in their own isolated-address space, they do not have direct access to system resources, so when they need to access such resources that only the kernel can interact with, there will be a privilege escalation wherein the application executes a `syscall` instruction which grants the application access to kernel-mode resources temporarily [18].

Likewise, in ARM, specifically in the Cortex-M family of processors, there also executes modes of operations: **handler mode** and **thread mode** [19]. When a processor is started, it enters into the thread mode by default. The handler mode is invoked when an interrupt is triggered.

Cortex-M offers an additional level of control known as **privileged** and **unprivileged** instructions. Their properties work in lock step with the two modes described above [19]:

- **Thread mode**: privileged instructions can access all system resourced. Unprivileged mode prevents the modification of protected memory regions. When privileged resourced need to be accessed from unprivileged mode, the `SVC` (Supervisor Call) instruction is executed, which is similar to the `syscall` we saw earlier in `x86`.

- On processor reset, by default it starts in thread mode. When the processor is in handler mode, the instructions are always executed in privilege mode.

This level of granularity is important within the Cortex-M architecture, which ensures that unprivileged instructions do no access protected memory-mapped regions such as system timers and system control registers.

Address	Name	Type
0xE000EDFC	DEMCR	RW
0xE0001020 + 16n	DWT_COMPn	RW
0xE0001024+16n	DWT_MASKn	RW
0xE0001028+16n	DWT_FUNCTIONn	RW

Table 3.1: Data Watchdog and Trace Unit Register Addresses

3.2.3 Data Watchpoint and Trace Unit

The Data Watchpoint and Trace Unit is the most important hardware mechanism of this thesis's goal. It consists of three classes of registers: Comparators, Function, Mask. All three of which are required to work together for monitoring of memory regions. The important memory addresses of this unit are given in the table 3.1.

3.2.3.1 Comparators

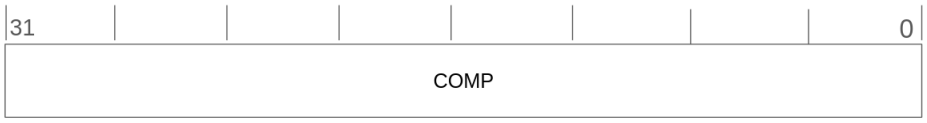

Figure 3.2: DWT_COMPARATORn [16]

The purpose of the comparator is to compare the value it holds with the following:

- A data address
- An instruction Address
- A data value
- The cycle count value

The DWT unit can have anywhere from 0 to 15 comparators. The Armv7-M specification for the board used in our testing has a total of 4 comparators. Each comparator can monitor a 32KB region of memory.

3.2.3.2 Mask Register

Figure 3.3: DWT_MASKn Register [16]

If the DWT unit(s) is being used to monitor a range of addresses, then the mask register is used to specify that range. There is a mark register for every comparator supported by the implementation. The *MASK* field shown in Figure 3.3 is used to define the number of bits to ignore when comparing the data address in the equivalent comparator. In the ARMv7-M implementation we used for our evaluation in Chapter 6, each mask register supports monitoring 32KB of memory.

3.2.3.3 Function Register

Each comparator has a function register associated with it. The function registers are used to control what action to make when a match occurs. A match is when a memory region is accessed as defined in the FUNCTION bits as shown in Figure 3.4. The MATCHED bit indicates if the comparator has matched since the bit was last read. The function register supports a variety of different operations, which are defined in the reference manual [16].

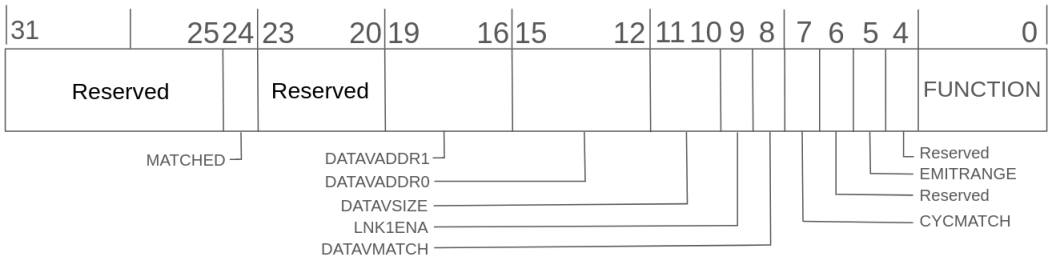

Figure 3.4: DWT_FUNCTIONn Register [16]

3.2.3.4 DEMCR Register

The Debug Exception and Monitor Control Register is used to enabled various debug and trace features. For the DWT unit specifically [16], as shown in Figure 3.5 the TRCENA bit is used to enable/disable the DWT functionality, i.e, monitoring memory regions. The MON_EN bit is used to allow the DWT to generate debug events. When the MON_EN bit is set, the DWT unit can trigger the DebugMon_Handler exception when a match occurs.

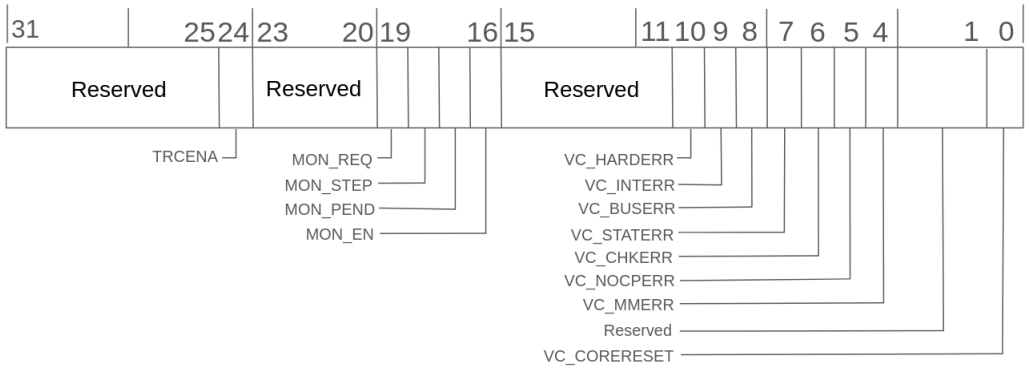

Figure 3.5: DEMCR Register [16]

3.3 LLVM

LLVM is a compiler infrastructure something like the popular GNU Compiler Collection (GCC), but unlike GCC, it is not a compiler itself but offers a collection of modular and reusable compiler tool chain technologies. GCC is limited in that it offers robust compiler frontends used to compile supported programming languages. LLVM on the other hand, is not a compiler itself, but offers a framework with front-ends and back-ends to build compilers. GCC has widespread use, however, lately LLVM has been gaining a bit of popularity. LLVM tends to write more performant code than GCC [20] while GCC tends to generate slightly more optimized code with 1-4% performance increase [21]

3.3.1 Classical Compiler Design

The compilation phase is chiefly broken down into three phases [22]:

1. **Front End**

   The front end is responsible for parsing the source code, checking for errors and building the internal representation of the written code called the Abstract Syntax Tree (AST).

   (a) Lexical Analysis: converts the code (which is in plain text) to token

   (b) Syntax Analysis: performs analysis on the token stream to check if the grammar rules of that language are followed, and then builds the AST.

   (c) Semantic Analysis: checks for semantic errors like type mismatches and adds type information to the AST.

2. **Middle End**

   The middle end or the **optimizer** is responsible for a broad range of transformations which tries to improve the performance of the code by eliminating any redundancies such as duplicate code, dead code, etc.

3. **Back End**

   This phase is responsible for creating target-specific code from the optimized intermediate representation from the optimizer. The back end has to ensure that it generates correct code that is supported on the target architecture. It's commonly divided into sub-phases [23]:

   (a) **Instruction Selection**: translates IR into target machine instructions.

   (b) **Register Allocation**: allocates CPU registers to hold variables and temporary values.

   (c) **Instruction Scheduling**: reorders instructions to minimize execution time while preserving program semantics.

3.3.2 LLVM IR

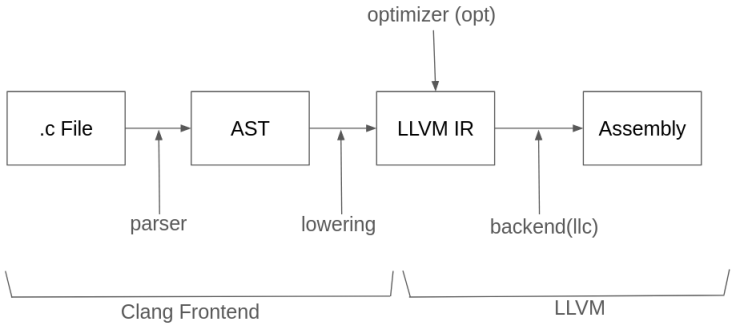

Figure 3.6: LLVM Compilation Process

LLVM offers a high-end compiler for C/C++/Objective-C which is as performant as GCC. Like GCC, it utilizes the classical compiler design discussed before. Clang, as a frontend, is responsible for parsing the source code into an Abstract Syntax Tree (AST) and *lowers* it into an intermediate representation (IR). Then an optimizer transforms this IR into a more optimal version. Finally, the backend takes this optimized IR and converts it into the target architecture's machine code.

From Figure 3.6, we see that LLVM only refers to the optimizer and backend of the compilation process. Now, since LLVM works at the IR level of the compilation process: when creating a backend, we are concerned in modifying this IR so that when the backend (llc) performs instruction translation from IR to machine code, LLVM is instrumenting the instructions required for this design.

```
1 int square(int) {
2     return x * x
3 }
```

Listing 3.1: Regular C Function

```
1 define i32 @square(i32 %x) {
2     %1 = mul i32 %x, %x
3     ret i32 %1
```

```
4 }
```

Listing 3.2: Equivalent IR Representation

In Listing 4, we see a simple C function which takes an integer as an argument and returns the square of that integer. In Listing 5, the LLVM IR of the same C function is shown. This example is meant to demonstrate how LLVM converts code to the IR level. The reasoning behind this particular structure and how the optimizer converts the AST into IR is out-of-scope of this document. Interested readers might want to take a look at Compiler Design books talking about LLVM [24] and compiler-design engineers who write about LLVM [25]

3.3.3 LLVM Table Gen Description Files

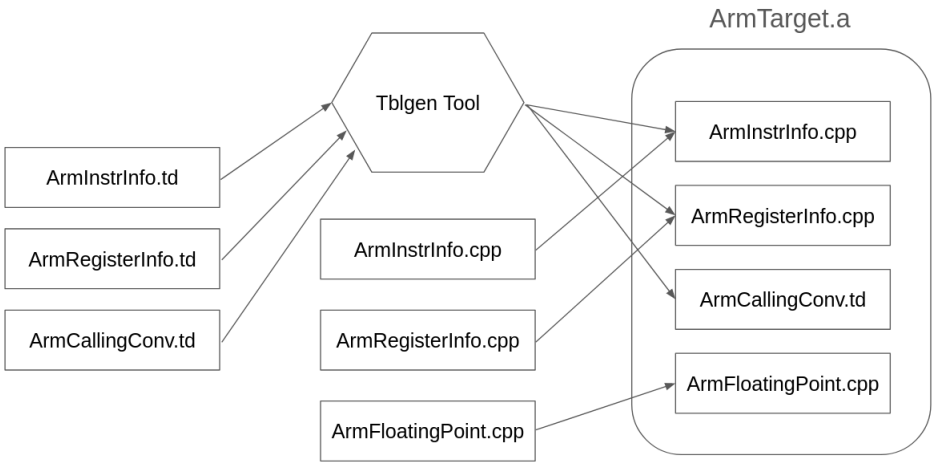

Figure 3.7: ARM Target Definition with TableGen

For the backend to convert the instructions into correct machine code, there needs to be a library of information that LLVM can look at to perform this translation. LLVM offers this functionality in the form of the TableGen. The TableGen is itself an interpreted language processed from target description files (`.td`) which generates relevant C++ code that is used in code generation. This is to enable modularity of generating code so that these TableGen

files can be reused even when compiling to different architectures. The build process for a specific architecture, ARM, is shown in Figure 3.7.

3.3.4 LLVM Code Generation

LLVM uses the Static Single Assignment (SSA) based representation. LLVM provides an infinite number of virtual registers which can hold values of various primitive types, which implies that every operand can be saved in a different virtual register. The code generation process is explained below [26]:

1. **Target-Independent Optimizations**: the IR is optimized using common optimization techniques such as dead code elimination, loop invariant code motion, and subexpression elimination, among others.

2. **Instruction Selection**: The LLVM IR is lowered to `MachineIntr IR` that is target specific. For ARM, the `ARMInstructionSelector` class handles this when the LLVM IR instructions are mapped to ARM machine instructions.

3. **Scheduling and Formation:** The MachineInstrs are put into a basic block ordering and the instruction stream is formed. The `ARMTargetMachine` class coordinates this process.

4. **SSA-based Optimizations:** while the machine code is still in SSA form, certain optimizations like peephole optimizations and instruction combining are performed.

5. **Register Allocation:** LLVM uses a greedy register allocator by default. LLVM allows use of an infinite number of virtual registers which are mapped to ARM physical registers.

6. **Prolog/Epilog Insertion:** The function prologue and epilogue, which handle function setup and stack management, are inserted. This process is handled by `ARMFrameLowering` class.

7. **Late Machine Code Optimizations:** Optimizations like dead code elimination and peephole optimizations are done again but this time on non-SSA machine instructions.

8. **Code Emission:** The final machine instructions are encoded into executable code, either in assembly or object code format. The `ARMAsmPrinter` class prints the assembly code.

3.4 Control Flow Integrity

This document mainly deals with proposing a new security mechanism for ensuring Control Flow Integrity (CFI) in embedded systems. CFI was first systematized in 2009 [3]. It ensures that during run-time, a particular program has a definitive execution path defined in a Control-Flow Graph (CFG). The CFG, being a policy, needs to be determined ahead of time so that it can be enforced. This is implementation-dependent and has much research since it's proposal [27, 28, 29, 30]. As defined in [3], our proposed mechanism enforces CFI by instrumenting machine code LLVM. The limitation of this approach is that the source code is required to do compile-time instrumentation using LLVM. There are ways of achieving this without source code: by utilizing binary code instrumentation [13]; however, that is out of the scope of this document.

In particular, we focus on ensuring the backward-edge integrity of embedded systems. This is achieved using a shadow stack, which maintains a copy of the return addresses of the original stack. This shadow stack is protected by some mechanism that prevents the attack from corrupting it. Therefore, when a function needs to return to its caller, it uses this shadow stack to get the legitimate return address even if the attacker has corrupted the return address on the original stack.

Chapter4

Design: Measure Once, Cut Twice

> There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
> - *C. A. R. Hoare*

4.1 Threat Model

In our threat model, we consider a powerful attacker has the following capabilities:

- Full knowledge of memory layout
- Ability to read and write to arbitrary memory locations
- The Hardware-Abstraction Library and `C` libraries are part of the Trusted-Computing Base

One of the attacker's goals is to hijack the control-flow of the embedded software by injecting shell code or return-oriented programming (ROP) gadgets.

4.2 Compact Shadow Stack

The C/C++ languages are vulnerable to control-hijacking attacks due to the way they handle function calls and returns. In these languages, the function callee stores the return address to the function caller and upon finishing executing, loads this saved return address into

the instruction pointer so that the next instruction gets executed from the saved point. In conventional hijacking attacks, an attacker finds a way to rewrite this saved return address by using buffer overflow vulnerabilities. In particular, ensuring the integrity of these return addresses on a function's stack is called *backward-edge* integrity. Function pointers, for example, constitute *forward-edges* and are protected by `CFI`. Most `CFI` mechanisms assume that the backward-edges are protected.

A shadow stack is one kind of backward-edge protection – stack canaries and safe stacks being the other heavily utilized options – which maintains a copy of the return addresses. There are two main kinds of shadow stacks: 1) compact shadow stacks and 2) parallel shadow stacks [31].

The compact shadow stack shown in figure 4.1 stores a copy of all the return addresses of the functions called during execution. The storage mechanism is elementary, storing each return address next to each other while maintaining a pointer to the current top of the "stack" of return addresses. The parallel shadow stack is a similar mechanism; however, it stores the return addresses relative to the offset of the stack pointer of each function. As it implies, the parallel shadow stack requires more memory space as it has a parallel copy of the main stack, while the shadow stack has a smaller memory overhead but requires an additional pointer. Both these designs suffer in support for multi-threading and exception handling, but, in the case of microcontrollers, which are by design single-threaded do not suffer from this lack of support.

We chose the compact shadow stack for our design as it offers a minimal memory overhead and due to the DWT Comparators in armv7-M only supporting monitoring of 32KB of memory per comparator.

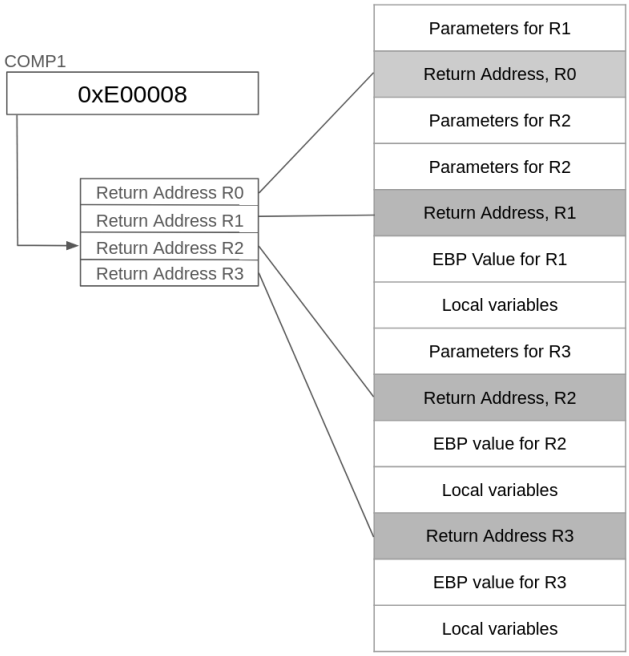

Figure 4.1: Traditional Shadow Stack

4.3 Data Write Monitoring

The primary objective in monitoring the compact shadow stack is to prevent an attacker from modifying the return address to prevent control-flow hijacking. We handle this requirement by utilizing the Data Watchdog and Trace Unit to monitor the shadow stack region. The configuration of registers of the DWT unit are set at the startup file of each application/program. The values of each of these registers is given in table 4.1.

4.4 Function Prologue

The function prologue needs to be modified to do the following:

1. Disable `COMP0` which protects the shadow stack
2. Store `LR` to the address pointed by `COMP1`
3. Increment `COMP1` value by 4 bytes

4. Enable `COMP0` again

The reasoning for modification of the prologue is as follows:

- Since the return address of the function caller has to be stored in the shadow stack, write monitoring the shadow stack region needs to be disabled first.
- In ARM, the return address of a function is stored in the `LR` register, so we store its value to the shadow stack using the pointer `COMP1`

4.5 Function Epilogue

The function epilogue has to be modified to do the following:

1. Decrement `COMP1` value by 4 bytes
2. Load `LR` with the value at the address pointed to by `COMP1`

The reasoning behind the modification of the epilogue is as follows:

- Since the value has to be read from the shadow stack, write protection for the shadow stack does not need to be disabled.
- We load the return address from the shadow stack using the pointer to either `LR` or `PC` depending on the function being instrumented.

Name	Address	Value	Function
DWT_COMP0	0xE0001020	0XE00000	Set to start of Shadow Stack
DWT_MASK0	0xE0001024	0X1f	Set to monitor first 32KB of Shadow Stack
DWT_FUNCTION0	0xE0001028	0X06	Set to monitor writes of address in COMP0
DWT_COMP1	0xE0001030	0xE00000 + 4n	Serves as the Shadow Stack Pointer
DWT_COMP2	0xE0001040	0xE000EDFC	Set to address of DEMCR register
DWT_MASK2	0xE0001044	0x01	monitors 4 bytes from address in COMP2
DWT_FUNCTION2	0xE0001048	0x06	monitors writes for COMP2
DWT_COMP3	0xE0001050	N/A	N/A

Table 4.1: DWT Comparator Values and Functions

Chapter5

Implementation: Guarding the Gates

5.1 Code Base

To implement our design, we utilized the existing code base from Silhouette CFI [10] with their code publicly available on Github [32]. Silhouette uses LLVM `9.0.1` to instrument the instructions in the prologue and epilogue. Since an older version of LLVM is being used, the pass is written in accordance with the legacy PassManager [33]. If this implementation were to be ported to a newer version, then the new PassManager should be utilized [34].

5.2 Components of the LLVM BackEnd Pass

ARM Back-end code generation is achieved by overriding the functions in `ARMPassConfig` which are virtual functions from `TargetPassConfig`. The `TargetPassConfig` class specifies target-independent code generation options for the sole purpose of use by other code generation passes. The steps for registering a new LLVM Backend pass with the Legacy PassManager are given below:

1. All the backend passes are usually kept in the `llvm/lib/Target/ARM` directory. One can see other passes of the ARM Backend CodeGen process such as `ARMCallingConv.cpp` and `ARMLoadStoreOptimizer.cpp`.

2. Create a new file related to the pass being implemented, in our case, `ARMShadowStack.h` and `ARMShadowStack.cpp` being the header files and the definitions files respectively.

3. Define the functions in these files and within the Main Class of the header file, make sure to include a virtual function called `runOnMachineFunction`. The LLVM PassManager calls this function to run the pass. Therefore, this function should have the logic defined in other functions in the definition file.

5.3 DWT Unit Configuration

As mentioned in Chapter 4, during startup the DWT configuration is set. This involves 56 lines of C code that does the following:

1. Sets the DWT registers with the values mentioned in Table 4.1
2. Enables the `MON_EN` and `TRCENA` bits in the `DEMCR` register
3. Sets the Priority of the DebugMonitor Exception.
4. The configuration also includes a custom `DebugMon_Handler` function that is triggered if there is illegal access of the shadow stack.

5.4 Function Prologue

To implement the design of the function prologue, we need to set the relevant bits of the Comparators, Function and Mask registers. This is achieved by using the relevant machine instructions, as shown in Listing 5.1.

```
// prologue - needs 2 free registers
    movw    Rx, #4128   // COMP0 lower address: 0x1020
    movt    Rx, #57344  // COMP0 higher address: 0xe000
    mov.w  Ry, #0 // null offset
    str.w   Ry, [Rx, #8] // disable FUNC0: ss write-protection
    mov.w  Ry, #16 // offset to access COMP1
    ldr.w   Ry, [Rx, Ry] //load stack pointer
    str.w   LR, [Ry] // store LR at dereferenced stack pointer location
    addw    Ry, Ry, #4 // incr shadow stack pointer by 4 bytes
    str.w   Ry, [Rx, #16] // store it back to COMP1
    mov.w   Ry, #6 // monitor writes in FUNC0
    str.w   Ry, [Rx, #8] // re-enable FUNC0
```

Listing 5.1: DWT Function Prologue

The function prologue requires two free registers for instrumentation. If an instrumented function does not have enough free registers available, we reserve the required number by pushing a register, i.e, the `push` instruction, in particular, registers `R4` through `R6` and use them in the prologue. After the prologue, we restore the registers by popping their values using the `pop` instruction.

5.5 Function Epilogue

The function epilogue, shown in Listing 5.2, reads the return address from the shadow stack based on the current shadow-stack pointer value and loads it into the `LR` register. Due to this process just being a read, and not a write, the DWT configuration does not need to be modified.

```
// epilogue - needs 3 free registers
    add sp, #4 // to balance the stack
    movw    Rx, #4144   // COMP1 lower address: 0x1030
    movt    Rx, #57344  // COMP1 upper address: 0xe000
    mov.w   Ry, 0 // null offset
    ldr.w   Ry, [Rx, Ry]  // load current shadow stack pointer
    mov.w   Rz, #0 // null offset
    subw    Ry, Ry, #4 // decr ement pointer first
    str.w   Ry, [Rx] // # store updated ss pointer to COMP1
    ldr.w   LR, [Ry, Rz] // # load ss value to LR
```

Listing 5.2: DWT Function Epilogue

The function epilogue requires three free registers. Like the prologue, we reserve and restore registers to meet this requirement. The reason for the third register is due to `LLVM` not allowing the use of a static offset during instrumentation; it only allows the offset in a register, as seen in line 9 of Listing 5.2.

5.6 Limitations

- The Shadow stack is only 32KB in size, which means that only 8192 functions can be protected (assuming 4 bytes per return address).
- For every function that is instrumented, 2 and 3 registers need to be reserved for the prologue and epilogue respectively, this causes more instructions of pushing and popping from the stack, adding to performance overhead.
- Every store instruction is a privileged store instruction: this implies that an attacker could alter the memory-mapped DWT regions, causing the mechanism to fail. A possible solution to this could be implementing a more fine-grained mechanism to protect the DWT region without using the MPU.

Chapter6

Evaluation: To Shreds you say?

The board used for the evaluation was a STMicroelectronics STM32F469 Discovery kit, which uses the STM32F469NIH6 MCU that is based on the ARM Cortex-M4 core with DSP and FPU. The MCU has a 180 MHz max CPU frequency, 2MB Flash memory, 384KB SRAM. It also has a 4MB SDRAM. We evaluated our implementation using the BEEBS benchmark [35].

6.1 Validation

We validate the security mechanism by writing a test program that tries to modify contents on the shadow stack, which is at a known fixed point in memory defined in the linker script. The DWT monitoring this region will immediately trigger a `DebugMon_Handler` exception and control will be transferred to a custom handler. Depending on user requirements, this handler can be modified to meet any criteria, but for demonstration purposes, this custom handler resets the system.

6.2 Methodology

Building on Silhouette, we used Clang 9.0.0 to compile the BEEBS Benchmark suite. We compiled and evaluated three versions of the benchmark: Baseline, Shadow Stack Only and DWT enabled. We want to demonstrate any differences in performance between a parallel shadow stack and a compact shadow stack. We also used the default compile-time

optimization (`-O3`) options, as well as the default LLVM link-time optimization flag (`-flto`).

The benchmark, BEEBS [35], short for "*Bristol/Embecosm Embedded Benchmark Suite*" is an open source benchmark suite designed to evaluate the energy consumption of embedded processors. This benchmark has been used in previous works to evaluate research studies [10, 36].

6.3 Results

Previous works [31] have mentioned that parallel shadow stacks will outperform compact ones and from our analysis, we observe the same. However, it is important to note that this performance decrease is a trade-off for lower memory utilization compared to the parallel shadow stack.

We have included the results from running each benchmarks 30 times and recording their execution times. Table 6.1 shows the code-size overhead for each program, and Table 6.2 shows the run-time overhead for each program. However, as the authors of Silhouette note, many of the programs in BEEBS are small in both input size and processing, leading to small execution times (in *ms*) that offer insufficient data to be meaningful. As such, we have chosen to retain only those programs which had an execution time of 1s or more. This is shown in Table 6.3.

Our design is seen to have a geometric increase of 1.8% in code size and a geometric increase of 12.2% in performance overhead. Notably, the maximum overhead seen is only an 85% increase over the baseline. Compared to Silhouette, there is a significant difference in performance overhead. This is because of the extra store and load operations in every function prologue and epilogue, causing an increase in CPU cycles.

Table 6.1: BEEBS: Code-size increase comparison

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Text Size	Text Size	Increase (%)	Text Size	Increase (%)
aha-compress	32472	32884	1.27	33048	1.77
aha-mont64	37052	37464	1.11	37628	1.55
bs	31604	31940	1.06	32092	1.54
bubblesort	33028	33516	1.48	33692	2.01
cnt	32300	32788	1.51	32964	2.06
compress	33356	33768	1.24	33932	1.73
cover	31632	31968	1.06	32120	1.54
crc	32028	32512	1.51	32688	2.06
crc32	32796	33208	1.26	33372	1.76
ctl-stack	32672	33084	1.26	33248	1.76
ctl-string	32948	33552	1.83	33744	2.42
ctl-vector	32892	33316	1.29	33476	1.78
cubic	45932	46412	1.05	46596	1.45
dijkstra	33604	34168	1.68	34356	2.24
dtoa	45152	45864	1.58	46072	2.04
duff	32052	32540	1.52	32716	2.07
edn	36040	36452	1.14	36616	1.6
expint	32140	32552	1.28	32716	1.79
fac	31832	32244	1.29	32408	1.81
fasta	32480	32892	1.27	33056	1.77
fdct	32628	33040	1.26	33204	1.77

Continued on next page

Table 6.1 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Text Size	Text Size	Increase (%)	Text Size	Increase (%)
fibcall	31564	31900	1.06	32052	1.55
fir	34992	35404	1.18	35568	1.65
frac	32684	33172	1.49	33348	2.03
huffbench	36084	36496	1.14	36660	1.6
insertsort	32328	32664	1.04	32816	1.51
janne_complex	31560	31896	1.06	32048	1.55
jfdctint	33540	34028	1.45	34204	1.98
lcdnum	32012	32348	1.05	32500	1.52
levenshtein	33360	33692	1	33868	1.52
ludcmp	35084	35496	1.17	35660	1.64
matmult-float	33536	34032	1.48	34208	2
matmult-int	34856	35344	1.4	35520	1.9
mergesort	36988	37480	1.33	37660	1.82
miniz	63216	64192	1.54	64452	1.96
minver	32500	32912	1.27	33076	1.77
ndes	36724	37288	1.54	37476	2.05
nettle-aes	44436	44924	1.1	45100	1.49
nettle-arcfour	34172	34584	1.21	34748	1.69
nettle-cast128	39904	40316	1.03	40480	1.44
nettle-des	36836	37248	1.12	37412	1.56
nettle-md5	31760	32096	1.06	32248	1.54
nettle-sha256	35672	36160	1.37	36336	1.86

Continued on next page

Table 6.1 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Text Size	Text Size	Increase (%)	Text Size	Increase (%)
newlib-exp	31592	31928	1.06	32080	1.54
newlib-log	32200	33104	2.81	33280	3.35
newlib-mod	31592	31928	1.06	32080	1.54
newlib-sqrt	32752	33164	1.26	33328	1.76
ns	36984	37456	1.28	37620	1.72
nsichneu	41196	41608	1	41772	1.4
picojpeg	47696	48916	2.56	49200	3.15
prime	31924	32232	0.96	32412	1.53
qrduino	49376	50016	1.3	50216	1.7
qsort	32232	32644	1.28	32808	1.79
qurt	31624	32036	1.3	32200	1.82
recursion	31660	32192	1.68	32368	2.24
select	32056	32468	1.29	32632	1.8
sglib-arraybinsearch	32324	32736	1.27	32900	1.78
sglib-arrayheapsort	32596	33008	1.26	33172	1.77
sglib-arrayquicksort	32464	32876	1.27	33040	1.77
sglib-dllist	33028	33440	1.25	33604	1.74
sglib-hashtable	33148	33560	1.24	33724	1.74
sglib-listinsertsort	32464	32876	1.27	33040	1.77
sglib-listsort	32836	33248	1.25	33412	1.75
sglib-queue	33584	33996	1.23	34160	1.72
sglib-rbtree	33060	33768	2.14	33956	2.71

Continued on next page

Table 6.1 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Text Size	Text Size	Increase (%)	Text Size	Increase (%)
slre	35348	36080	2.07	36280	2.64
sqrt	32332	32744	1.27	32908	1.78
st	37220	37636	1.12	37800	1.56
statemate	32832	33244	1.25	33408	1.75
stb_perlin	35716	36208	1.38	36380	1.86
stringsearch1	34932	35344	1.18	35508	1.65
strstr	31812	32224	1.3	32388	1.81
tarai	31776	32308	1.67	32484	2.23
trio-snprintf	36896	37580	1.85	37780	2.4
trio-sscanf	37460	38248	2.1	38472	2.7
ud	34480	34892	1.19	35056	1.67
Geometric Mean			**1.31**		**1.81**

Table 6.2: BEEBS: Run-time comparison

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Text Size	Text Size	Performance (%)	Text Size	Performance (%)
aha-compress	524	528	100.76	528	100.76
aha-mont64	655	658	100.46	658	100.46
bs	5	5	100	5	100
bubblesort	2571.67	2670.1	103.83	2762	107.4
cnt	47	54	114.89	53.77	114.4

Continued on next page

Table 6.2 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Run Time	Run Time	Performance (%)	Run Time	Performance (%)
compress	310.67	322.73	103.88	332.73	107.1
cover	65	73.9	113.69	65	100
crc	37.27	44.47	119.32	42.17	113.15
crc32	636	713.5	112.19	787.2	123.77
ctl-stack	449.53	510.8	113.63	479.8	106.73
ctl-string	1211.2	1446.6	119.44	1528.4	126.19
ctl-vector	773.07	873.6	113	797.73	103.19
cubic	22996.4	25811.7	112.24	28592	124.33
dijkstra	40600	40967	100.9	40978	100.93
dtoa	727	767	105.5	771	106.05
duff	16	23	143.75	23.73	148.33
edn	2500.13	2592.4	103.69	2681	107.23
expint	93	97	104.3	96.87	104.16
fac	91.47	46.73	51.09	44.6	48.76
fasta	14110.13	15201.4	107.73	16277	115.36
fdct	130	133	102.31	133	102.31
fibcall	2	2	100	2	100
fir	15127.6	15873.7	104.93	16210	107.16
frac	8817	8853	100.41	8853	100.41
huffbench	46130	46134	100.01	46133	100.01
insertsort	21	21	100	21	100
janne_complex	8.6	3.6	41.86	3	34.88

Continued on next page

Table 6.2 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Run Time	Run Time	Performance (%)	Run Time	Performance (%)
jfdctint	101.8	115.93	113.88	124.8	122.59
lcdnum	5.4	13.87	256.79	6	111.11
levenshtein	6269.2	8410.2	134.15	9356	149.24
ludcmp	239	243	101.67	243	101.67
matmult-float	299	321	107.36	324	108.36
matmult-int	5901	5909	100.14	5909	100.14
mergesort	31222	31413	100.61	31513	100.93
miniz	28	47	167.86	46.37	165.6
minver	52.6	55.7	105.89	55.7	105.89
ndes	1938	2051	105.83	2054	105.99
nettle-aes	7027	7035	100.11	7037	100.14
nettle-arcfour	562	565	100.53	565	100.53
nettle-cast128	342	348	101.75	346	101.17
nettle-des	369	369	100	372	100.81
nettle-md5	3	3	100	3	100
nettle-sha256	710.47	745.4	104.92	770.33	108.43
newlib-exp	3.8	28.6	752.63	3	78.95
newlib-log	34.2	63.67	186.16	64	187.13
newlib-mod	3	9.5	316.67	3	100
newlib-sqrt	60.2	74.07	123.03	83.8	139.2
ns	187.67	218.33	116.34	243	129.48
nsichneu	382.33	415	108.54	449.67	117.61

Continued on next page

Table 6.2 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Run Time	Run Time	Performance (%)	Run Time	Performance (%)
picojpeg	28743.67	44227.33	153.87	53301	185.44
prime	11102	363	3.27	11101	99.99
qrduino	43586	43633	100.11	43632	100.11
qsort	57	60	105.26	59.9	105.09
qurt	11.47	13.6	118.6	11.6	101.16
recursion	168.87	508.53	301.14	511.07	302.65
select	17	20	117.65	19.9	117.06
sglib-arraybinsearch	787.2	822.33	104.46	846.67	107.55
sglib-arrayheapsort	886.2	891.57	100.61	892.73	100.74
sglib-arrayquicksort	762.33	761.53	99.9	756.73	99.27
sglib-dllist	1289	1311.87	101.77	1330.73	103.24
sglib-hashtable	273	486.13	178.07	455.4	166.81
sglib-listinsertsort	1298.87	1332.8	102.61	1362.73	104.92
sglib-listsort	1078.33	1072.03	99.42	1061.73	98.46
sglib-queue	2063.47	2102.13	101.87	2137.8	103.6
sglib-rbtree	7424.4	10818.67	145.72	11224	151.18
slre	4172	5228	125.31	5270	126.32
sqrt	55665	55869	100.37	55717	100.09
st	20063	20066	100.01	20029	99.83
statemate	25	28	112	27.9	111.6
stb_perlin	2985.73	3562.13	119.31	3697	123.82
stringsearch1	303	306	100.99	306	100.99

Continued on next page

Table 6.2 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Run Time	Run Time	Performance (%)	Run Time	Performance (%)
strstr	57	61	107.02	60.87	106.78
tarai	68.47	146.07	213.34	142.8	208.57
trio-snprintf	900.7	998.7	110.88	1021.8	113.45
trio-sscanf	1308.6	1543.23	117.93	1538.9	117.6
ud	355.6	333.33	93.74	290.4	81.66
Minimum			3.27		34.88
Maximum			752.63		302.65
Geometric Mean			110.83		110.21

Table 6.3: BEEBS: Meaningful Run-time comparison

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Run Time	Run Time	Performance (%)	Run Time	Performance (%)
bubblesort	2571.67	2670.1	103.827474	2762	107.4010273
ctl-string	1211.2	1446.6	119.4352708	1528.4	126.1889036
cubic	22996.4	25811.7	112.242351	28592	124.3325042
dijkstra	40600	40967	100.9039409	40978	100.9310345
edn	2500.13	2592.4	103.6906081	2681	107.2344238
fasta	14110.13	15201.4	107.7339472	16277	115.3568394
fir	15127.6	15873.7	104.9320447	16210	107.1551337
frac	8817	8853	100.4083021	8853	100.4083021

Continued on next page

Table 6.3 – continued from previous page

	Baseline	Shadow Stack		DWT Enabled	
Benchmark	Run Time	Run Time	Performance (%)	Run Time	Performance (%)
huffbench	46130	46134	100.0086711	46133	100.0065034
levenshtein	6269.2	8410.2	134.1510879	9356	149.2375423
matmult-int	5901	5909	100.1355702	5909	100.1355702
mergesort	31222	31413	100.6117481	31513	100.9320351
ndes	1938	2051	105.8307534	2054	105.9855521
nettle-aes	7027	7035	100.1138466	7037	100.1423082
picojpeg	28743.67	44227.33	153.868069	53301	185.4356107
qrduino	43586	43633	100.1078328	43632	100.1055385
sglib-dllist	1289	1311.87	101.7742436	1330.73	103.2373933
sglib-listinsertsort	1298.87	1332.8	102.6122707	1362.73	104.9165813
sglib-listsort	1078.33	1072.03	99.41576326	1061.73	98.46058257
sglib-queue	2063.47	2102.13	101.8735431	2137.8	103.6021847
sglib-rbtree	7424.4	10818.67	145.7177684	11224	151.1771995
slre	4172	5228	125.3116012	5270	126.3183126
sqrt	55665	55869	100.366478	55717	100.093416
st	20063	20066	100.0149529	20029	99.83053382
stb_perlin	2985.73	3562.13	119.3051616	3697	123.8223148
trio-sscanf	1308.6	1543.23	117.9298487	1538.9	117.5989607
Minimum	710.47	745.4	99.41576326	770.33	98.46058257
Maximum	55665	55869	153.868069	55717	185.4356107
Geometric Mean			109.0743296		112.2028587

Chapter7

Conclusion: Thats All Folks

In conclusion, we present a novel backward-edge integrity security mechanism that utilizes the Data Watchdog and Trace unit to monitor the shadow stack. We implemented our design on an ARMv7-M board from STMicro. The evaluation of which shows that our design has very insignificant code-size overhead with a geometric increase of 1.81%. In performance analysis, we observed a geometric increase of 12.1% over the baseline, with the maximum overhead observed being an increase of 85%. Considering our architecture that utilizes a smaller memory-size shadow stack, we find this performance overhead to be an acceptable trade-off.

We also propose some directions for future work: reduce the CPU cycles for each prologue and epilogue by reserving a GPR to store the DWT address that is used for instrumentation. Also, disallow configuring of the DWT monitor except at the prologue of each protected function.

Bibliography

[1] Anton Shilov. *842 Chips Per Second: 6.7 Billion Arm-Based Chips Produced in Q4 2020 — tomshardware.com*. [Accessed 07-04-2024]. URL: https://www.tomshardware.com/news/arm-6-7-billion-chips-per-quarter.

[2] Gartner Inc. *Gartner Says Worldwide PC Shipments Grew 10.7 in Fourth Quarter of 2020 and 4.8 for the Year*. [Accessed 07-04-2024]. 2021. URL: https://www.gartner.com/en/newsroom/press-releases/2021-01-11-gartner-says-worldwide-pc-shipments-grew-10-point-7-percent-in-the-fourth-quarter-of-2020-and-4-point-8-percent-for-the-year.

[3] Martín Abadi et al. "Control-flow integrity principles, implementations, and applications". In: *ACM Trans. Inf. Syst. Secur.* 13.1 (2009). ISSN: 1094-9224. DOI: 10.1145/1609956.1609960. URL: https://doi.org/10.1145/1609956.1609960.

[4] Xi Tan et al. *Where's the "up"?! A Comprehensive (bottom-up) Study on the Security of Arm Cortex-M Systems*. 2024. arXiv: 2401.15289 [cs.CR].

[5] Crispin Cowan et al. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, p. 5.

[6] Perry Wagle and Crispin Cowan. "StackGuard: Simple Stack Smash Protection for GCC". In: *Proceedings of the GCC Developers Summit*. GNU. 2003, pp. 243–256. URL: https://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf.

[7] Microsoft C++ Team. *Visual Studio 2015 Preview: Work-in-Progress Security Feature*. Nov. 2014. URL: https://devblogs.microsoft.com/cppblog/visual-studio-2015-preview-work-in-progress-security-feature/.

[8] Caroline Tice et al. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[9] Xi Tan et al. "Is the Canary Dead? On the Effectiveness of Stack Canaries on Microcontroller Systems". In: *ACM/SIGAPP Symposium On Applied Computing (SAC)*. 2024.

[10] Jie Zhou et al. "Silhouette: Efficient Protected Shadow Stacks for Embedded Systems". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1219–1236. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie.

[11] Jinfeng Li et al. "Zipper Stack: Shadow Stacks Without Shadow". In: *Computer Security – ESORICS 2020*. Ed. by Liqun Chen et al. Cham: Springer International Publishing, 2020, pp. 338–358. ISBN: 978-3-030-58951-6.

[12] *μRAI: Securing Embedded Systems with Return Address Integrity*. en-US. URL: https://www.ndss-symposium.org/ndss-paper/murai-securing-embedded-systems-with-return-address-integrity/.

[13] Thomas Nyman et al. *CFI CaRE: Hardware-supported Call and Return Enforcement for Commercial Microcontrollers*. 2017. arXiv: 1706.05715 [cs.CR].

[14] Arm Ltd. *Licensing Arm Technology — arm.com*. [Accessed 07-04-2024]. 2024. URL: https://www.arm.com/products/licensing.

[15] *Apple starts its two-year transition to ARM this week — engadget.com*. https://www.engadget.com/apple-arm-transition-timeline-191106454.html. [Accessed 13-04-2024].

[16] ARM Ltd. *ARMv7-M Architecture Reference Manual*. https://documentation-service.arm.com/static/64b7f5c638511951cb79fc45. [Accessed 07-04-2024].

[17] Joseph Yiu. *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. 3rd ed. Newnes, Apr. 2024. ISBN: 978-0124080829. URL: https://www.amazon.com/Definitive-Guide-Cortex%C2%AE-M3-Cortex%C2%AE-M4-Processors/dp/0124080820.

[18] Andrew S. Tanenbaum. *Modern Operating Systems — pearson.com*. https://www.pearson.com/en-us/subject-catalog/p/modern-operating-systems/P200000003311/9780133591620. [Accessed 13-04-2024]. 2014.

[19] Yigeng Zhu. *amazon.com*. https://www.amazon.com/Embedded-Cortex-M-Microcontrollers-Assembly-Language/dp/0982692676/. [Accessed 13-04-2024]. 2023.

[20] *Comparing clang to other open source compilers — opensource.apple.com*. URL: https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html.

[21] Alibaba Cloud. *GCC vs. Clang/LLVM: An in-depth comparison of C/C++ compilers*. 2024. URL: https://www.alibabacloud.com/blog/gcc-vs--clangllvm-an-in-depth-comparison-of-cc%2B%2B-compilers_595309.

[22] James Alan Farrell. *Compiler Basics*. 1995. URL: https://www.cs.man.ac.uk/~pjj/farrell/compmain.html.

[23] Chris Lattner. *LLVM*. The Architecture of Open Source Applications (Volume 1). 2011. URL: https://aosabook.org/en/v1/llvm.html.

[24] Mayur Anand et al. *LLVM Cookbook*. Packt Publishing, May 2015. ISBN: 9781785285981.

[25] Miguel Young de la Sota. *A Gentle Introduction to LLVM IR*. Aug. 2023. URL: https://mcyoung.xyz/2023/08/01/llvm-ir/.

[26] Jonathan Chuang. *About - Low Level Virtual Machine Backend Tutorial*. URL: https://jonathan2251.github.io/lbd/about.html.

[27] László Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[28] Volodymyr Kuznetzov et al. "Code-pointer integrity". In: *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Association for Computing Machinery and Morgan & Claypool, 2018, 81–116. ISBN: 9781970001839. URL: https://doi.org/10.1145/3129743.3129748.

[29] Hong Hu et al. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[30] Nicholas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. ISBN: 978-1-939133-11-3. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[31] Nathan Burow, Xinping Zhang, and Mathias Payer. "SoK: Shining Light on Shadow Stacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 985–999. DOI: 10.1109/SP.2019.00076.

[32] URSec. *Silhouette*. https://github.com/URSec/Silhouette. 2020.

[33] LLVM Project. *Writing an LLVM Pass*. 2021. URL: https://llvm.org/docs/WritingAnLLVMPass.html.

[34] LLVM Project. *Writing an LLVM Pass*. Apr. 2024. URL: https://llvm.org/docs/WritingAnLLVMNewPMPass.html.

[35] James Pallister, Simon Hollis, and Jeremy Bennett. *BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms*. 2013. arXiv: 1308.5174 [cs.PF].

[36] Xi Tan and Ziming Zhao. "SHERLOC: Secure and Holistic Control-Flow Violation Detection on Embedded Systems". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. CCS '23. Copenhagen, Denmark: Association for Computing Machinery, 2023, 1332–1346. ISBN: 9798400700507. DOI: 10.1145/3576915.3623077. URL: https://doi.org/10.1145/3576915.3623077.