uSFI: Ultra-Lightweight Software Fault Isolation for IoT-Class Devices

Zelalem Birhanu Aweke and Todd Austin
University of Michigan
Email: {zaweke, austin}@umich.edu

***Abstract*—Embedded device security is a particularly difficult challenge, as the quantity of devices makes them attractive targets, while their cost-sensitive design leads to less-than-desirable security implementations. Most current low-end embedded devices do not include any form of security or only include simple memory protection support. One line of research in crafting low-cost security for low-end embedded devices has focused on sandboxing trusted code from untrusted code using both hardware and software techniques. These previous attempts suffer from large trusted code bases (*e.g.*, including the entire kernel), high runtime overheads (*e.g.*, due to code instrumentation), partial protection (*e.g.*, only provide write protection), or heavyweight hardware modifications.**

In this work, we leverage the rudimentary memory protection support found in modern IoT-class microcontrollers to build a low-profile, low-overhead, flexible sandboxing mechanism that can provide isolation between tightly-coupled software modules. With our approach, named uSFI, only the trust management code must be trusted. Through the use of a static verifier and monitored inter-module transitions, module code at all privilege levels (including the kernel) is able to run uninstrumented and untrusted code. We implemented uSFI on an ARMv7-M based processor, both bare metal and running the freeRTOS kernel, and analyzed the performance using the MiBench embedded benchmark suite and two additional highly detailed applications. We found that performance overheads were minimal, with at most 1.1% slowdown, and code size overheads were also low, at a maximum of 10%. In addition, our trusted code base is trivially small at only 150 lines of code.

I. Introduction

Embedded devices are everywhere, with low-end embedded devices being used in many critical systems such as medical, industrial, and automotive applications. With the Internet-of-Things (IoT) promising as much as 30 billion connected devices by 2020, the security of low-end embedded devices has become a major concern. The rising number of devices along with increased connectivity has immensely exacerbated the attack surface of this class of devices, making them a genuine target of interest for savvy attackers. Two recent examples of this trend include the Jeep remote kill attack [21] and the Mirai IoT-based botnet [11].

A. Memory Isolation in IoT-Class Devices

While the exposure to attacks is increasing, the security mechanisms in these systems remain mediocre due to the tight resource (*e.g.*, computing power and memory size) and price constraints. Recent works have shown that embedded devices suffer from the same memory corruption and code injection vulnerabilities that have plagued traditional computing systems [5], [7], [24].

To make matters worse, many embedded systems lack fundamental code and data memory protection mechanisms that are available in more powerful computing systems. For example, low-end embedded processors typically do not include a memory management unit (MMU) to reduce cost and in some cases to provide real-time execution time guarantees [17]. Therefore, low-end embedded systems typically operate in a single address space with tightly-coupled software modules (*e.g.*, RTOS kernels, peripheral drivers, libraries, *etc.*) without any form of isolation between modules. These software modules often come from different chip, sensor and I/O device vendors, which ultimately raises serious questions as to whether or not these modules can be trusted. Software modules might contain bugs or even malicious code that could be used to compromise the security of the whole system. Consequently, there is a great need to provide IoT-class devices with low-cost, reliable protections against widely employed attacks, such as memory corruption and code injection.

B. Low-Cost Software Fault Isolation

Previous work has proposed mechanisms to protect code and data in embedded systems, with varying degrees of security assurances and resource requirements. At the heart of these mechanisms, often called *software fault isolation (SFI)*, is support for isolating software modules from adversely affecting each other, through hardware and software means [10], [12], [13], [25]. The approach can be a powerful tool in providing security against memory corruption and code injection because if a software fault (or a bug) allows one module to be compromised, it cannot arbitrarily render other modules inoperable, but instead the faulty module is forced to implement its mischief via the programmer-defined interfaces to the other modules.

TrustLite [10] proposed a hardware extension to provide isolation of trusted modules (Trustlets) from untrusted code including an untrusted OS. While TrustLite provides strong data isolation guarantees, it requires a hardware extension. Most importantly, it is hardly scalable as the required area of the hardware extension grows linearly with the number of protected modules (the area of the hardware extension matches that of the core for nine modules).

ARMor [25] uses SFI to sandbox non-critical code by performing binary rewriting to put checks before store operations identified as being potentially unsafe. At runtime, it uses a separate control stack to protect return addresses. Similarly, [12] and [13] use a separate stack to protect return addresses. These software-based protection mechanisms are attractive as they don't require hardware changes. However, in order to reduce the overhead of the runtime checks, the techniques only provide write protection: a malicious software can read and leak sensitive data. In addition to this, the additional memory guard instructions result in larger code sizes and higher performance overheads. As such, there is a need for a *low-cost* mechanism that provides *strong* (read, write, and execution) protection.

C. Ultra-Lightweight Software Fault Isolation

In this work we present uSFI, a low-cost code and data isolation mechanism for resource constrained embedded devices. uSFI uses readily available hardware support along with static software analysis to provide stronger security guarantees at a lower cost than previous efforts. In a uSFI-enabled system, an application is composed of sandboxed modules. Modules, including privileged modules such as the kernel, are untrusted (*i.e.*, they can be compromised due to a bug in the module or contain malicious code); only a static binary verifier and a small trust management runtime need be trusted. In this way, isolation guarantees can be maintained even when the kernel is compromised (*e.g.*, a compromised kernel cannot arbitrarily read a calling application's private data). Modules communicate through cross-module calls made through well-defined interfaces managed by the uSFI runtime.

978-3-9819263-0-9/DATE18/©2018 EDAA

uSFI includes a compiler that takes as an input module configurations, and produces a binary that satisfies certain restrictions to enforce code and data isolation. The uSFI verifier ensures that compiler-generated binaries satisfy these restrictions for all executions, including those that might employ code gadgets. Through the use of the uSFI verifier, it is possible to safely incorporate third party drivers and modules. With the combined use of the uSFI compiler and verifier, it becomes possible to ensure highly reliable software fault isolation at very low cost; uSFI uses uninstrumented load and store instructions and indirect jumps, while strictly enforcing inter-module code and data isolation.

We implemented uSFI on the widely used ARM Cortex-M microcontrollers. We evaluated uSFI using the MiBench embedded benchmark suite and other real-world embedded applications, showing that it incurs less than a 10% code size overhead and roughly a 1% performance overhead. Moreover, the trusted code in a complete application build is trivially small at only 150 lines of code. In summary, this paper makes the following contributions:

- We propose uSFI, a code and data protection mechanism for low-resource devices. uSFI uses software analysis and widely available hardware support in embedded processors (memory protection units) to provide low-cost code and data isolation as well as I/O access control.
- We implement uSFI for the widely used ARMv7-M architecture. Using the MiBench embedded benchmarks suite and other real-world applications, we show that uSFI has low code size and performance overheads. Moreover, we show that the fraction of code that must be trusted in the system (*i.e.*, the uSFI runtime) is a trivially small fraction of the overall system code size. At 150 lines of code, the attack surface of our trust management system can be easily analyzed and inspected to gain trust.

The remainder of this paper is organized as follows. In Section II, we outline the threat model. Section III discusses the details of uSFI system architecture. In Section IV, we evaluate uSFI using representative embedded benchmarks and other real-world applications. In Section V, we discuss related work , and finally draw conclusions in Section VI.

II. Threat Model

In a uSFI-enabled system, software modules (library code, drivers, etc.) are untrusted (*i.e.*, they might contain software bugs or malicious code that lead to compromised execution). Modules can execute any code within their sandbox, including attacker selected code gadgets [8].

A compromised module will try to execute attacker code by overwriting code memory, executing data, or forming gadgets out of existing code. Furthermore, a corrupted module will try to corrupt the data memory or read sensitive data such as encryption keys from the memory of other modules. Finally, a compromised module will try to gain elevated access to I/O to gain access to a remote controller or leak sensitive information.

Unlike other low-end fault isolation systems, we assume the kernel is not trusted. Similarly, the system compiler is not trusted, instead a trusted verifier is used to verify module code generated by the compiler. The uSFI verifier and the uSFI runtime are the only trusted software components. We assume the underlying hardware is trusted. Finally, we also assume there is a trusted bootloader that verifies and loads binaries at system startup.

III. uSFI System Architecture

The goal of uSFI is to protect code and data from untrusted software modules. Untrusted software modules include core modules, third party libraries, drivers, and operating system kernels. This is achieved by sandboxing modules in their own security domain and using a well-defined interface for cross-domain procedure calls. In this section, we provide details of the uSFI architecture.

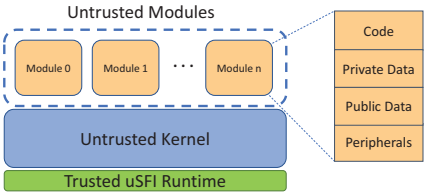

Fig. 1. uSFI System Components.

TABLE I
Access Permission Configuration for Module Memory Regions.

Module Memory Region	Unprivileged/Privileged Permissions
Code	RO, X
Read-only Data	RO, NX
Stack	RW, NX
Data	RW, NX
Peripherals	RW, NX

A uSFI-enabled system has two components: a trusted runtime and untrusted modules. The uSFI runtime is the only trusted component in the system, and it has access to the entire memory and sole access to the memory protection resources. Software modules, including the kernel and drivers, are all untrusted. A module represents a single security domain with its own code, data and peripheral memory regions. Figure 1 illustrates this isolation capability. A module can only execute code in its code region, and it can only access data belonging to itself or public data in other modules. Furthermore, a module can only access peripherals assigned to it.

A. Memory Isolation

uSFI leverages a Memory Protection Unit (MPU) hardware available in many embedded processors to provide isolation between modules. The MPU divides the memory map into regions and defines access permissions and memory attributes for each region. In a uSFI-enabled system, the memory map is divided into uSFI memory and modules' memory. uSFI memory is a small portion of the memory map used by the uSFI runtime. This portion of memory is inaccessible by any of the modules. The rest of memory (modules' memory) is divided among modules, and is freely accessible by the corresponding module code. Each module memory region is divided into code, stack, read-only data, private and public data, and peripheral regions.

In a uSFI enabled system, only a single module is active at a time. Inter-module calls are managed by the uSFI runtime. On the invocation of a new module's function, the MPU configuration is changed to reflect the access permissions of the new module. Table I shows the access permission configuration for each region of the active module. As shown in the table, each region has distinct permission requirements. Data is not executable, therefore all data regions (*i.e.*, read-only data, stack and data) are configured as non-executable (NX). The module regions are configured to be accessible by privileged and unprivileged code, *i.e.*, the active module and the uSFI runtime have access to these regions. The active module only has access to the configured regions. Other modules' regions (unconfigured regions) are made inaccessible to non-privileged code by use of the MPU hardware.

B. Sandboxing Kernel Code

Typically code running at privileged level (*e.g.*, kernel code) has access to all system resources including system control and

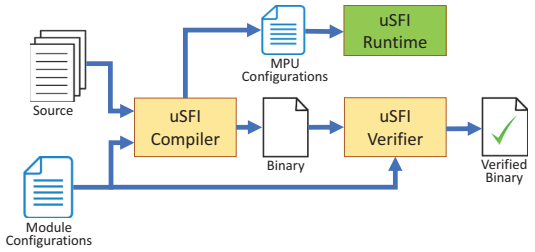

Fig. 2. uSFI Compiler and Verifier.

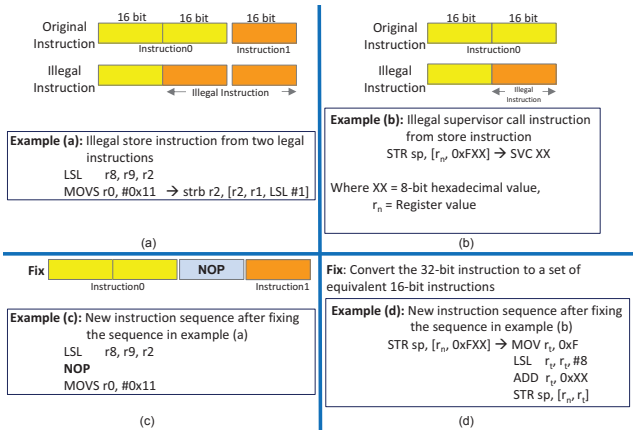

Fig. 3. Potential Illegal Instructions and How to Fix Them.

configuration resources (*e.g.*, MPU configuration registers). In a uSFI-enabled system, however, even the kernel is sandboxed such that it only has access to memory regions assigned to it. One way of achieving this is forcing the kernel to run at unprivileged level and calling the trusted runtime when privileged operations are needed. Unfortunately, this approach results in frequent context switches, incurring significant performance overhead. We need a method that would allow the kernel to perform important privileged operations (*e.g.*, configuring interrupt) without calling the uSFI runtime, but at the same time prevent it from accessing system control resources and user space memory. To do this we use a novel approach that takes advantage of *unprivileged memory access instructions* [18] (page A4-113). With this approach, a privileged module is forced to use unprivileged memory access instructions that grant access to only the memory that the uSFI runtime permits it to access, instead of allowing carte blanche memory access with regular loads and stores.

C. uSFI Compiler and Verifier

uSFI is composed of a uSFI compiler with a verifier and a uSFI runtime. The uSFI compiler generates a binary that conforms to the restrictions discussed below. Figure 2 shows the steps of generating a binary in a uSFI-enabled system. A programmer specifies module configurations for each module through a uSFI API. The configurations include the module number, privilege level, peripheral access permissions, and entry function. Based on this information, the compiler selects module memory region sizes and allocates regions to modules.

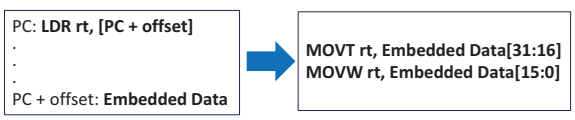

Fig. 4. PC-relative Load Instruction and its Conversion to Safe Move Instructions.

The compiler also generates MPU configurations to be used by the uSFI runtime.

After compilation, a static verifier ensures that the generated binary satisfies restrictions set by uSFI. The restrictions depend on the privilege level of the module and are listed below:

*a) **Modules can only issue supervisor (system) calls that are assigned to them**:* Modules issue supervisor calls when they want to make cross-module procedure calls. To keep cross-module call overhead low, the uSFI runtime keeps the amount of checks required during module transition to the minimum. Modules are identified by a unique module number. When a module wants to call a function in a different module, it issues a supervisor call with the callee's module number as an argument. The runtime identifies the module to switch to using this number. However, it doesn't check the source of the call. It is up to the uSFI verifier to make sure that modules issue only allowed supervisor calls (*i.e.*, supervisor calls with the right module numbers).

*b) **Privileged modules (e.g., kernel code) can access memory using only unprivileged (and thus MPU checked) memory access instructions**:* In a uSFI-enabled system all code except the uSFI runtime has restricted access to memory. To enforce this, privileged modules can not use regular load and store instructions, a requirement that is validated by the uSFI verifier.

In a uSFI-enabled system the uSFI compiler is not part of the trusted computing base. The compiler is expected to generates code that satisfies the above restrictions by ensuring that all code is discoverable at compile time. However, the correctness of the code doesn't solely rely on the rather large compiler. Instead, compiler generated code has to be vetted by the trusted verifier before it is ready for execution. The vetting process ensures that all code is discoverable at compile time (*i.e.*, it is impossible to form new instructions at runtime).

In modern architectures, it is possible to form new instructions at runtime that are not observed at compile time through code gadgets that jump into the middle of code and data. For example, in the ARMv7-M architecture (a widely used embedded processor architecture) this can happen in two ways. First, due to the variable instruction encoding in the ARMv7-M architecture, it is possible to jump into the middle of an instruction and form new instructions. ARMv7-M uses the Thumb-2 instruction set which supports both 32-bit and 16-bit instructions [18] (page A4-100). Instructions are stored half-word (16 bits) aligned and 16-bit and 32-bit instructions can be intermixed freely. Therefore it is possible to form illegal instructions at runtime by: 1) executing two 16-bit instructions as a single 32-bit instruction, or 2) jumping into the middle of a 32-bit instruction and executing the lower half of the instruction.

Figure 3 illustrates this potential vulnerability, and our remedy to prevent it from creating illegal code sequences. In Figure 3 (a) an example is given on how an illegal store instruction can be constructed by jumping into the middle of a 32-bit instruction and combining it with the next 16-bit instruction. Assuming the code resides inside a privileged module, this violates the second restriction stated above. Figure 3 (b) provides an example that shows how an arbitrary system call can be formed by jumping into the middle of a legal 32-bit instruction. This violates the first restriction stated above. Figure 3 (c) and (d) show how the uSFI compiler fixes these potential violations. To fix scenario (a), a *NOP* instruction is inserted between the two 16-bit instructions. The scenario in (b) is fixed by replacing the 32-bit instruction with an equivalent 16-bit instruction sequence. In the given example, a 32-bit store instruction with an immediate offset is replaced with a 16-bit store instruction with a register offset. Additional move and arithmetic instructions are used to load the immediate value into the offset register.

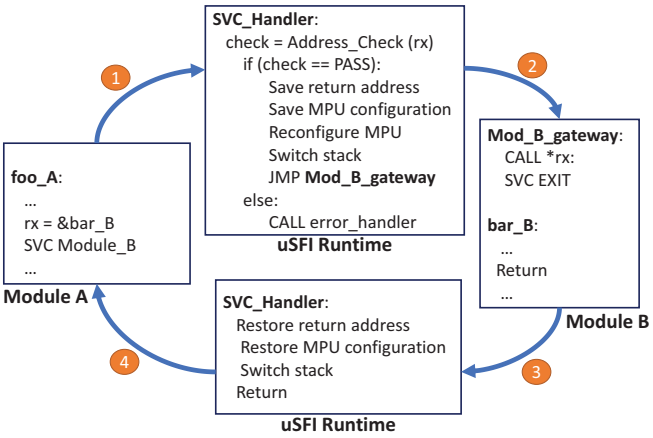

Fig. 5. Inter-module Function Call.

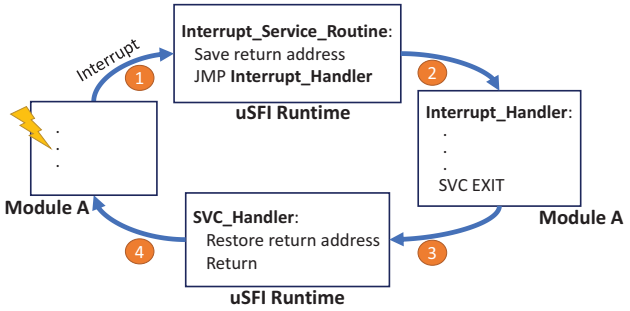

Fig. 6. uSFI Interrupt Handling.

Another avenue to generate illegal instructions is through data embedded in code memory. This occurs when using PC-relative addressing to access data. In PC-relative addressing, the data to be loaded is located in the code region at a fixed offset from the program counter. With data embedded in the executable code region, it is possible to form illegal instructions by jumping into the code-segment embedded data. The uSFI compiler deals with this potential vulnerability by excising all data from the code segment. With our baseline compiler (LLVM), code-segment embedded data is limited to immediate values for register loads. The compiler deals with potential violations by simply replacing the instructions with other safe instructions. For example, in the ARMv7-M architecture PC-relative load instructions can be replaced by two immediate move instructions, as shown in Figure 4. The replacement does not incur any performance overhead as the two move instructions take the same amount of time to execute as a single load instruction

D. uSFI Runtime

The second component of uSFI, the uSFI runtime, manages modules at runtime. The responsibilities of the runtime include handling the switch between modules and handling interrupts/exceptions. It keeps a list of modules and their configurations which include the value of the stack pointer for the module, MPU configurations, and privilege levels of the module. The runtime also keeps a list of exported functions and their entry points for each module. These are functions that can be called from within other modules.

Inter-module Function Calls: One of the tasks of the uSFI-runtime is to handle the switch between modules. A switch between modules is required when a module wants to call an exported function in another module. At compilation, the uSFI compiler installs *gateway functions* in each module to facilitate module switches. The gateway function is an entry point to a module. When a module is created, it is assigned a module number which is used in inter-module function calls.

Figure 5 shows the steps involved in inter-module function calls. In the figure function *foo_A* in module A wants to call the exported function *bar_B* in module B. In (1) Module A passes a pointer to function *bar_B* in register *rx*. Then it issues a supervisor call, with module number of module B as an argument, to the uSFI runtime. (Note that from a programmer's perspective this is a regular function call; the supervisor call instruction and the assignment to register *rx* are automatically inserted by the uSFI compiler). The uSFI runtime verifies that the function pointer points to a function exported by module B. If the check passes, the runtime saves the return address and the MPU configurations of module A. Then, after changing the MPU configuration to enable memory regions of module B and switching the stack, control is transferred to the gateway function in module B (2). At the end of execution of function *bar_B*, module B issues an exit supervisor call to the uSFI runtime (3). Finally, the uSFI runtime restores the stack and MPU configurations of module A and transfers control back to module A (4).

Each module has a public memory region that is freely accessible by other modules. This memory region is used to pass data between modules during inter-module calls. For example, a module that has data to be written to a file would move the data to the public memory region before making a system call to an untrusted kernel.

Interrupt/Exception Handling: Interrupt handling is performed by the uSFI runtime. The uSFI API allows modules to assign their own interrupt handler routines to certain events. Figure 6 shows the interrupt handling process in a uSFI-enabled system. In the figure, an interrupt occurs while module A is executing. As a result, control is transfered to an interrupt service routine in the uSFI runtime. This routine sets the right privileges, saves the return address and calls an interrupt handler routine assigned by the module. All module interrupt service routines end with a supervisor call instruction. At the end of a module interrupt service routine, an exit supervisor call is issued to the uSFI runtime. In the supervisor call handler, the uSFI runtime restores the return address to module A and transfers control back to module A.

IV. Experimental Evaluation

We implemented the uSFI compiler using the LLVM compiler infrastructure [14], and the verifier using the radare2 binary analysis framework [4]. Our implementation targets the widely used ARMv7-M embedded processor architecture (specifically cortex-M4 processors), but can easily be extended to ARMv7-R and the new ARMv8-M/R architectures. The MPUs in Cortex-M4 processors allow configuring up to eight memory regions (region 0 to region 7). Region 0 to region 5 are used for code, read-only data, stack, private data (initialized and uninitialized data and heap), and public data, respectively. The remaining three regions are used for peripheral access control. Cortex-M4 MPUs allow further subdividing of each region into eight equal-sized sub-regions. This approach allows up to 24 distinct peripheral regions that can be enabled or disabled to provide fine-grained peripheral access control. Note that there is only one active module at a time and MPU configuration is changed on module switch. The uSFI runtime is written in C and assembly with the bulk of the code implementing a supervisor call (SVC) handler. It has a total size of less than 150 lines of C and assembly statements.

We used two development boards for our evaluations: STMicroelectronics's NUCLEO-F446RE development board [23], and NXP's FRDM-K64F development board [22]. The NUCLEO-F446RE board uses a microcontroller with an ARM Cortex-M4 processor, 512KB flash memory, and 128KB RAM. The FRDM-K64F board uses a microcontroller

TABLE II
USFI CODE SIZE OVERHEAD.

Benchmark	Original Code Size (Bytes)	Additional Code Size (Bytes)	% Overhead
dijkstra	11,388	576	5.1
susan	51,092	406	0.8
basicmath	22,880	806	3.5
bitcount	9,248	280	3.0
qsort	18,572	744	4.0
stringsearch	17,484	130	0.7
rijndael	41,904	3,224	7.7
sha	8,676	232	2.7
blowfish	16,512	560	3.4
FFT	18,008	410	2.3
CRC32	7,388	228	3.1
mbedtls	362,736	2,682	0.7
FreeRTOS	**45,360**	**4,272**	**9.6**

with an ARM Cortex-M4 processor, 1MB flash memory, and 256KB RAM. We evaluated code size and performance overheads of uSFI using representative embedded benchmarks from the MiBench embedded benchmark suite [9], mbed TLS library [2], the FreeRTOS kernel [1], a widely used embedded RTOS, and two detailed real-world applications.

A. Code Size Overhead

There are two sources of code size overhead in uSFI. First, there is minor overhead when replacing PC-relative load and arithmetic operations in module code. PC-relative loads (16-bit instructions) are replaced with two immediate move instructions (32-bit each) as shown in Figure 4. 32-bit immediate arithmetic operations are replaced with immediate move instructions followed by register arithmetic operations. The other source of code size overhead is when replacing regular load and store instructions with unprivileged load and store instructions in privileged modules. Regular memory access instructions are typically 16-bit instructions in the ARMv7-M architecture, while unprivileged memory access instructions are 32-bit instructions.

Table II shows the code size overhead for benchmarks from MiBench, mbed TLS library, and the FreeRTOS kernel. The *mbedtls* benchmark tests all the cryptographic operations in the mbed TLS library. Except for FreeRTOS, all benchmarks are running as an unprivileged module (*i.e.*, the benchmarks use regular load and store instructions). The FreeRTOS kernel is running as a privileged module but it does not have access to user space memory or system configuration registers as it is forced to use unprivileged load and store instructions.

In the table *rijndael* has a relatively higher code size overhead among the unprivileged benchmarks. This is due to the large number of PC-relative loads of the same pointers to s-box tables. FreeRTOS has a relatively larger code size overhead since unprivileged load and store instructions are twice as large as the ordinary memory access instructions. Overall, compared to the total size of the flash memory in the devices, the additional code size is small.

The results shown in Table II are obtained by naively removing all embedded data from code memory. Although the overhead is small, it can be further reduced to a negligible size by selectively removing embedded data, *i.e.*, remove only data that can potentially be used to form illegal instructions as discussed in Section III. Finally, it is important to note that there is no performance overhead *inside* the sandboxes, since instructions (in particular loads/stores and indirect jumps) are not instrumented in any fashion.

B. Performance Overhead

We also measured the performance overhead of uSFI using other highly detailed real-world applications. To measure execution cycles, we used the Data Watchpoint and Trace Unit (DWT) facility available on ARM Cortex-M4 processors [19]. DWT provides clock cycle measurements among other things.

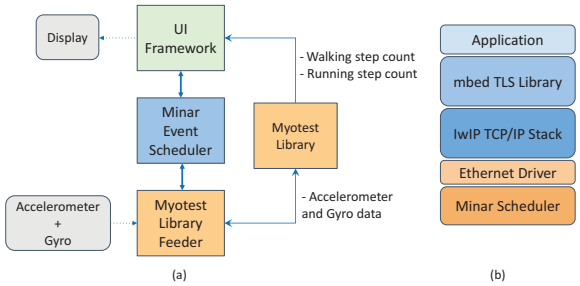

Fig. 7. Detailed Applications used for Performance Evaluation. (a) ARM Wearable Reference Design Step Analysis Application (b) HTTPS File Download Application.

The performance overhead in uSFI comes from module switches during cross-module function calls as shown in Figure 5. Overall, it takes 210 cycles to call a function in a different module and 150 cycles to return from the call. To evaluate the effect of module switching on applications' overall performance, we used two applications: a step analysis test application from ARM's Wearable Reference Design (WRD) [16] and an HTTPS file Download application [2].

C. Case Study 1: Step Analysis

The step analysis application periodically (every 10ms) samples sensor data from accelerometer and gyroscope sensors, and calculates walking and running steps. The result is displayed on a matrix LCD. Figure 7(a) shows a block diagram of the application. The application has three major components. The *minar event scheduler* is a non-preemptive event scheduler from mbedOS 3. The *myotest library* along with *myotest library feeder* is a step analysis library from Myotest [3]. The *UI framework* provides an interface to the matrix LCD display. We evaluated the application using the NUCLEO-F446RE development board operating at 120MHz. As a sensor input we used a dataset of 14,255 samples of sensor data provided by Myotest.

Sandboxing can be applied at different levels for different components of an application. We measured the performance overhead of sandboxing for two cases. In the first case only the myotest library (with the feeder) was sandboxed. In the second case two sandboxes were used; one for the myotest library, and another for the UI framework. Table III shows the results for the two cases. The table compares the two results with a baseline implementation where no application components are sandboxed. As can be observed from the table, the overhead of sandboxing the modules is small. This can be attributed to the low sampling rate of the sensors (sampling is done every 10ms) as compared to the processing speed of the processor. This is typical in many embedded systems applications that utilize sensors.

D. Case Study 2: HTTPS File Download

The second application we tested is an example application for the mbed TLS library. The application downloads a file from an HTTPS server and looks for a specific string in that file. The application runs on ARM's mbedOS embedded operating system. Figure 7(b) shows the different software components of the application. Each software component is contained in its own sandbox. We used the NUCLEO-F446RE development board operating at 120MHz for the test.

The baseline application (without uSFI) takes on average 3.2 seconds to execute. With uSFI enabled, an average of 3276 module switches were recorded. Each module switch takes 3us at 120MHz clock frequency. This results in an average

TABLE III
EXECUTION CYCLES FOR THE STEP ANALYSIS APPLICATION.

	Baseline	uSFI 1	uSFI 2
Total No. of Module Switches	0	42,765	43,778
Additional Clock Cycles	0	15,395,400	15,760,080
Total Clock Cycles	1,434,843,597	1,448,689,953	1,449,054,633
% Overhead	0	1.07	1.10

overhead of only 0.31%. Most of the overhead comes from the Ethernet driver which is invoked every 1ms.

V. RELATED WORK

Several works have proposed hardware and software techniques to provide isolation in embedded devices. TrustLite [10] and Tytan [6] propose a hardware extension, Execution-Aware Memory Protection Unit (EA-MPU), to provide isolation of trusted modules from untrusted code including untrusted OS. Similarly uSFI assumes the OS kernel is untrusted, but doesn't require any hardware changes. Furthermore, Trustlite uses static hardware configuration table to configure the EA-MPU, which means the area overhead grows quickly with the number protected modules. This limits the number of protected modules that can be supported. By allowing a small trusted runtime to configure the MPU, uSFI can support unlimited number of software modules.

Other line of research has looked at software-only solutions to isolate software modules. ARMor [25] uses SFI to sandbox non-critical code. It uses binary rewriting to put checks before store operations identified as being potentially unsafe. At runtime, It uses a separate *control stack* to protect return addresses. Similarly, [12] and [13] use a separate stack to protect return addresses. Other indirect control flow instructions are validated by runtime checks. To reduce the overhead of the runtime checks, these techniques only provide write protection. In addition to this, the added memory guard instructions result in large code size and performance overhead. uSFI provides both memory read and write protections with negligible inner sandbox performance overhead.

ARM mbed uVisor [15] is a software hypervisor that creates independent secure domains called *boxes*. Like uSFI, uVisor uses the MPU to provide isolation and access control to peripherals. However, in uVisor MPU configurations are tied to process switches, *i.e.*, MPU configuration changes are done at process context switches. This makes isolation between *tightly-coupled* software modules expensive. Furthermore, RTOS integrated with uVisor runs with the same privilege as uVisor. In uSFI, however, privileged code such as an RTOS is sandboxed using non-privileged memory access instructions.

Another protection recently added to embedded devices is ARM TrustZone [20]. TrustZone allows partitioning software into secure and normal worlds and provides isolation between the two. Software in the secure world can access memories in both secure and normal worlds, while normal software can only access normal world (non-secure) memories. While both uSFI and ARM TrustZone provide software isolation, TrustZone has only two security domains (secure and non-secure), and therefore it can not provide fine-grained isolation between tightly-coupled modules.

VI. CONCLUSIONS

In this work we presented uSFI, a low-cost code and data isolation mechanism for resource constrained embedded devices. uSFI uses the memory protection unit (MPU) hardware available in many embedded devices along with static software analysis to provide stronger security guarantees at a lower cost than previous work. In a uSFI-enabled system, an application is composed of sandboxed modules. Modules, including privileged modules (*e.g.*, RTOS kernel), are untrusted. Only a static binary verifier and a small runtime are trusted. uSFI doesn't require any hardware changes and incurs only 10% code size overhead and roughly a 1% performance overhead on representative applications.

Looking to the future, we would like to explore how uSFI can be extended to other non-ARM architectures such as x86 and RISC-V. Currently uSFI relies on an ARM-specific memory protection unit. This hardware is not particularly architecture dependent, and we believe it could readily be integrated into any embedded processor. In fact, other non-ARM embedded processors already support this feature today (*e.g.*, AVR32). We expect the support for IoT-class memory protection units to grow in the future, especially as their use becomes more vital to providing efficient security guarantees.

VII. ACKNOWLEDGMENTS

This work was supported in part by C-FAR, one of the six STARnet Centers, sponsored by MARCO and DARPA.

REFERENCES

[1] Freertos. http://www.freertos.org/.
[2] mbed TLS. https://github.com/ARMmbed/mbedtls.
[3] Myotest. http://www.myotest.com/.
[4] radare2. https://github.com/radare/radare2.
[5] Anthony J. Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the Security Issues Surrounding Lights-out Server Management. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, WOOT'13, Berkeley, CA, USA, 2013. USENIX Association.
[6] F. Brasser, B. El Mahjoub, A. R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015.
[7] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association.
[8] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 15–26, New York, NY, USA, 2008. ACM.
[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, Dec 2001.
[10] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.
[11] B. Krebs. DDoS on Dyn Impacts Twitter, Spotify, Reddit. https://krebsonsecurity.com/2016/10/.
[12] R. Kumar, A. Singhania, A. Castner, E. Kohler, and M. Srivastava. A System For Coarse Grained Memory Protection In Tiny Embedded Processors. In *2007 44th ACM/IEEE Design Automation Conference*, pages 218–223, June 2007.
[13] Ram Kumar, Eddie Kohler, and Mani Srivastava. Harbor: Software-based Memory Protection for Sensor Nodes. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 340–349, New York, NY, USA, 2007. ACM.
[14] 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.
[15] ARM Ltd. mbed uVisor. https://www.mbed.com/en/technologies/security/uvisor/.
[16] ARM Ltd. *Wearable Reference Design*.
[17] ARM Ltd. *ARM Cortex-R Series Programmer's Guide*. 2014.
[18] ARM Ltd. *ARMv7-M Architecture Reference Manual*. December 2014.
[19] ARM Ltd. *ARM Cortex-M4 Processor TRM*. 2015.
[20] ARM Ltd. *TrustZone Technology for ARM v8-M Architecture*. 2016.
[21] C. Miller and C. Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. 2015.
[22] NXP. FRDM-K64F:Freedom Development Platform for Kinetis K64, K63, and K24 MCUs. http://www.nxp.com/products/software-and-tools/hardware-development-tools/freedom-development-boards.
[23] STMicro. http://www.st.com/en/evaluation-tools/nucleo-f446re.html/.
[24] Project Zero. *Over The Air: Exploiting Broadcom's Wi-Fi Stack*.
[25] L. Zhao, G. Li, B. De Sutter, and J. Regehr. ARMor: Fully Verified Software Fault Isolation. In *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, pages 289–298, Oct 2011.