HARM: Hardware-Assisted Continuous Re-randomization for Microcontrollers

Jiameng Shi
Computer Science
University of Georgia
jiameng@uga.edu

Le Guan
Computer Science
University of Georgia
leguan@uga.edu

Wenqiang Li
Institute of
Information Engineering, CAS
liwenqiang@iie.ac.cn

Dayou Zhang
Computer Science
University of Georgia
dayou.zhang@uga.edu

Ping Chen
Institute for Big Data
Fudan University
pchen@fudan.edu.cn

Ning Zhang
Computer Science & Engineering
Washington University in St. Louis
zhang.ning@wustl.edu

2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1614-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EUROSP53844.2022.00039

***Abstract*—Microcontroller-based embedded systems have become ubiquitous with the emergence of IoT technology. Given its critical roles in many applications, its security is becoming increasingly important. Unfortunately, MCU devices are especially vulnerable. Code reuse attacks are particularly noteworthy since the memory address of firmware code is static. This work seeks to combat code reuse attacks, including ROP and more advanced JIT-ROP via continuous randomization. Previous proposals are geared towards full-fledged OSs with rich runtime environments, and therefore cannot be applied to MCUs. We propose the first solution for ARM-based MCUs. Our system, named `HARM`, comprises a secure runtime and a binary analysis tool with rewriting module. The secure runtime, protected inside the secure world, proactively triggers and performs non-bypassable randomization to the firmware running in a sandbox in the normal world. Our system does not rely on any firmware feature, and therefore is generally applicable to both bare-metal and RTOS-powered firmware. We have implemented a prototype on a development board. Our evaluation results indicate that `HARM` can effectively thaw code reuse attacks while keeping the performance and energy overhead low.**

***Index Terms*—microcontroller security, code reuse attack, TrustZone, randomization**

1. Introduction

Compared to PCs and smartphones, microcontroller-based embedded devices (MCUs) are often used to perform specific tasks with less complexity. Therefore, they are widely used in applications needing high reliability and security, such as industrial control systems and medical equipment. In recent years, with the emergence of Internet-of-Things (IoT) technology, we have also witnessed a wave of consumer products powered by MCUs, such as wearables and smart home devices.

However, there are several fundamental challenges both technologically and operationally towards securing these deeply embedded devices. First, to meet the performance requirements, the firmware is typically developed in C/C++, which is more likely to have memory-related bugs. Second, these devices often exclude hardware support of modern defense mechanisms to reduce cost and energy consumption, making it easier to exploit potential vulnerabilities. Third, firmware tends to run in the privileged mode in a flat memory layout to reduce the overhead of switching between the unprivileged and privileged mode [1]. Therefore, a control hijacking attack usually gains the highest privilege over the system. Fourth, there are multiple stakeholders involved during firmware development, including chip vendors, third-party library/OS providers, device manufacturers, etc. This fragmented responsibility makes security hard to be guaranteed.

Memory errors can often lead to arbitrary code execution. This has become a real threat to MCU devices as demonstrated in recent attacks [2]–[6]. Since even low-end MCUs are equipped with *memory protection units* (MPU) that can be used to enforce DEP (aka XN or W^X) [7], attackers cannot simply inject malicious code to the memory of MCU devices. Instead, they tend to rely on code reuse attacks (CRA) [8]–[13] which perform malicious behaviors by leveraging existing code contents. In particular, in a *return oriented programming* (ROP) attack, attackers chain code snippets or gadgets scattered over the existing code sections. MCU devices, unfortunately, are vulnerable to these attacks [12], [14]. There are two general approaches towards defending against CRAs: prevention and mitigation.

Attack prevention techniques aim to deny exploit execution. Whenever an anomaly is detected, the program crashes to prevent further damage. Control flow integrity (CFI) [15], stack canary, and memory error detector [16], [17] are among the most studied prevention techniques. To reduce overhead, some prevention features are even integrated in the hardware. For example, the ARMv8.1-M pointer authentication (PAC) and branch target identification (BTI) extension [18] facilitates efficient CFI implementation. However, there have also been demonstrated attacks against these protections, such as control-flow-bending attack [19], PAC bypassing [20], exploitation of single master canary [21], [22], etc. Attack and defense on software security continue to be an arms race.

Attack mitigation techniques assume that exploits can eventually happen, but aim to mitigate the breach of system. Our work follows this direction. Randomization is one of the most popular methods, where the layout of the target program code is randomized (so useful gadgets

DOI 10.1109/EuroSP53844.2022.00039

cannot be collected) [23]–[28]. In these solutions, a unique code layout is generated for each program execution. This approach cannot be directly applied to MCU firmware, since the firmware is typically statically linked in a position-dependent fashion and stored in the flash memory for in-place execution. Recent research proposed to statically add code diversity to MCU firmware [29], where each device has a different copy of the firmware. Unfortunately, it can be circumvented by more advanced attacks that combine a memory disclosure attack and CRA. Specifically, the attacker can repeatably exploit a memory disclosure vulnerability to read the firmware's code and then compile a ROP chain on-the-fly remotely. Conceptually, this is similar to *Just-In-Time ROP* (JIT-ROP) [30], [31] and *Blind ROP* (BROP) attacks [32] on PC or smartphones, although the attack cannot be conducted locally on the victim MCU device due to the lack of scripting environment. Execution-only-memory (XOM) [33], [34] as an attack prevention technique, has the potential to defeat naïve direct JIT-ROP attacks because it prevents code leakage. However, it can be circumvented by *indirect* JIT-ROP attacks [31], [35] where code pointers can be harvested from (readable) data regions.

This work proposes a new system called HARM for ARM-based MCU devices to mitigate all the mentioned CRA attacks with continuous re-randomization. To enable randomization, HARM moves the code from flash to SRAM for execution. To prevent advanced CRA attacks, HARM performs randomization continuously. By keeping the randomization period shorter than the time required for compiling a ROP chain, HARM effectively invalidates JIT-ROP attempts. Compared with existing re-randomization solutions geared towards x86/x64 platforms, our approach addresses many non-trivial challenges unique to MCU.

Existing re-randomization solutions [36]–[41] heavily rely on the rich runtime environment in commodity OSs, such as the dynamic loader, signals, kernel, etc. None of them is standard in MCUs. HARM instead leverages ARM TrustZone to build a specifically engineered runtime system to assist re-randomization. ARM TrustZone for MCUs has been announced in 2015 with the introduction of ARMv8-M architecture [42]. It creates a trusted execution environment (TEE) that can provide security-critical services to the firmware. The TEE and the firmware run in the secure world and normal world respectively. Our runtime system resides in the secure world and periodically invokes non-bypassable requests for randomization. Upon the request, it suspends the firmware execution in the normal world temporarily to complete the randomization. Finally, it updates the internal tables and adjusts impacted code/data references of the target code before resuming the firmware execution. By bookkeeping security-related data, such as the real locations of functions in the secure world, attackers cannot easily learn the code locations and launch CRA attacks.

MCU firmware mostly adopts static linking in favor of run-time performance. Therefore, the code sections, which are position dependent, are merged into a continuous region in flash. If we follow existing randomization approaches that relocate the entire code region in the available address space, not enough entropy can be provided since MCUs only have SRAM of hundreds of KB. We therefore conduct more fine-grained randomization at function level. By shuffling the functions, more entropy can be added. Position dependent code in firmware also means after each randomization, HARM has to track and fix lots of broken code/data references caused by relocation. To minimize the performance overhead, we propose a set of binary rewriting rules to pre-process the binary. It essentially adds an indirection layer that remaps encoded code pointers to their real targets, facilitating smooth re-randomization.

However, if the attackers know the fixed mapping information between the code pointer and the target, they can corrupt the control data with validly encoded code pointers. This degrades HARM to a coarse-grained CFI mechanism. To prevent this from happening, we make sure that the rewritten firmware, when leaked, can never disclose such mapping information. To defeat brute-force attempts that learn the mapping gradually, HARM triggers a complete binary rewriting once HARM detects an unexpected reboot. Through complete binary rewriting, HARM directly updates the mapping information stored in the secure flash. This will nullify prior efforts in the brute-force attack.

HARM does not rely on any firmware-specific features or interfaces. Rather, it only assumes the hardware interface specified in ARM manuals. Therefore, it is inherently OS-agnostic. Common in the multi-party development environment, the device manufacturers need to integrate third-party software. Through binary rewriting, HARM enables protection without relying on the access of the source code; thus, intellectual property can be kept secret.

We have implemented a HARM prototype on an ARMv8-M development board, and evaluated the performance with five real world applications and a benchmark under different re-randomization frequencies. Our results suggest HARM can support re-randomization at a higher frequency than needed to defeat ROP attacks, while still keep the performance and energy overhead low. Specifically, we observed a maximum overhead of 5.8% (others were negligible) when HARM performs a randomization every 200 ms, which is much shorter than the measured time needed by a real JIT-ROP like attack.

We have made the following contributions:

- We proposed the first OS-agnostic continuous randomization solution for ARM MCUs based on TrustZone-M extension.
- We proposed a set of new binary rewriting techniques to work with the proposed continuous re-randomization solution.
- We implemented a prototype on an ARM Cortex-M based development board.
- We measured the minimum time required for an attacker to launch a JIT-ROP like attack in the idealistic setting (~600 ms). We demonstrated that HARM can defeat such attacks with small overhead.

The source code of our HARM prototype is available at https://github.com/MCUSec/HARM.

2. Background

2.1. ARM Microcontrollers

MCUs follow the System-on-Chip (SoC) design that integrates processor, RAM, and I/O peripherals on the

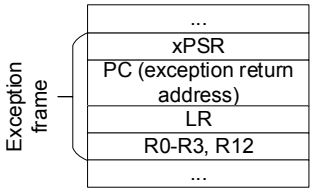


Figure 1. Exception frame automatically pushed by hardware

same silicon. This does not only cut the cost, but also decreases power consumption and improves system reliability. In this work, we focus on ARM Cortex-M series MCUs, which receive the broadest adoption and have widespread support from software companies [43].

Architecture: Mainstream ARM MCUs are powered by the ARMv7-M or ARMv8-M architecture. They adopt the Thumb-2 instruction set which is optimized for resource-constraint chips. It has 13 general-purpose registers ranging from `R0` to `R12`, in addition to the stack pointer (`SP` or `R13`), the link register (`LR` or `R14`), and the program counter (`PC` or `R15`). Besides, there are several system registers used to configure the working mode.

Operating Modes: The processor runs either in thread mode or handler mode. The handler mode is specifically designed to deal with exceptions and always runs with high privilege. The thread mode can either be privileged or unprivileged.

Exception Model: An exception traps the processor into the privileged handler mode. There are 15 system-defined exceptions along with the peripheral specific interrupts. The addresses of the handlers are stored in a table called *Vector Table*, whose base address is determined by the *Vector Table Offset Register* (VTOR), with the default value of zero. After powering on, the hardware reads the address of the reset handler from the vector table and starts booting from reset handler.

When an exception is triggered, the control flow is transferred to the corresponding handler according to the vector table. The operating mode is also automatically switched to the handler mode. Then, the processor context before the exception is pushed into the post-exception stack. This is called an exception frame as shown in Figure 1, which includes the status registers (`xPSR`), program counter (`PC`), link register (`LR`), and general-purpose registers `R12` and `R0-R3`. In the meanwhile, the link register `LR` is set with a special value called `EXC_RETURN`. When returning from an exception handler (i.e., jumping to `EXC_RETURN`), the values contained in the exception stack frame are restored to the corresponding registers so that the control flow can resume from the point right before exception.

Memory Protection Unit: The *Memory Protection Unit* (MPU) enforces memory access permissions (read/write/execution) for different privilege levels. For example, DEP can be easily enforced by MPU.

2.2. ARM TrustZone for Microcontrollers

TrustZone-M is a security extension to Cortex-M series MCUs. It separates the system resources into secure ones and non-secure ones. Correspondingly, the processor runs in two states, namely secure state and normal state. When the processor runs in the secure state, it can access all the resources. Otherwise, it can only access non-secure resources. To access secure resources, the normal world software requests a secure service in the secure world. When the normal world accesses secure resources, a newly introduced exception called *SecureFault* will be triggered.

2.2.1. Resource Partitioning. A memory region can be one of the three types: *Secure* (S), *Non-secure* (NS), or *Non-secure Callable* (NSC). An NSC region is a special kind of secure resource used for calling secure functions. It may hold an instruction called *Secure Gateway* (`SG`). The normal world can directly jump to an `SG` instruction in an NSC region and then further switch to the secure world. But jumping from the normal world to an S region directly or to any other instructions in an NSC region triggers a *SecureFault*. Following the `SG` instruction is usually a jump table to secure functions. This instruction sequence in the NSC region is called a *veneer*. To configure a region, *Secure Attribute Unit* (SAU) or *Implementation Defined Attribution Unit* (IDAU) can be used. Both of them are only accessible in the secure world.

2.2.2. World Switching. The firmware can switch between worlds via either the interrupt mechanism or dedicated instructions such as `SG`. ARM optimizes the world switching mechanisms to ensure the real-time property. For example, the `SG` instruction requires merely 3 CPU cycles (cf. §2.2 of [44]). As a comparison, a normal memory load/store instruction takes 1-2 cycles and the function call instruction (`BL`) takes 3 cycles.

3. System Overview

3.1. Threat Model and Assumptions

We consider a strong threat model in which the attacker can exploit memory errors to hijack the control flow of the firmware execution. An attacker can easily achieve this goal through buffer overflow or pointer subterfuge. For example, he can overwrite a return address on the stack and use write-what-where style vulnerability to overwrite an entry in function tables. He can further leverage CRA like ROP to run any existing code. We further assume the firmware has memory disclosure vulnerabilities that an attacker can exploit to launch both direct and indirect JIT-ROP like attacks [30], [31], [35].

We assume that the MCU devices are equipped with MPU, which is a common security feature even on low-end MCU chips. Typically, MPU is used to isolate the privileged code and data. However, most firmware is currently developed to run entirely in the privileged mode to reduce the overhead of privilege switching. Therefore, `HARM` leverages this hardware feature exclusively to enforce DEP [7]. This assumption is widely adopted in related work on MCU security [1], [21], [29], [33], [45]–[47]. `HARM` depends on TrustZone, which is available on ARM's current-generation MCU chips. The code running inside the secure world is assumed to be bug-free. Although there have been many vulnerabilities disclosed in the TEE implementation of smart-phones [48], TEE is still considered more secure because of the reduced

attack surface. This assumption is commonly accepted in TrustZone-based security solutions [49]–[51]. Finally, we trust our compiler and the added instrumentation.

We support multi-party development environment where the device manufacturers integrate libraries provided by other companies. The third-party vendors do not need to disclose their source code. However, they are required to keep the libraries unstripped. This holds in practice because the third-party libraries need to keep essential symbols to be linkable with libraries from others.

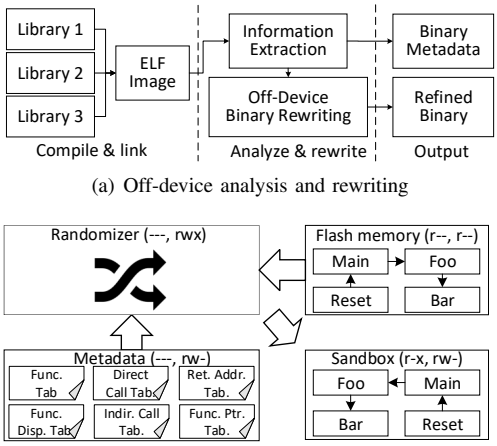
3.2. Challenges

Technical Challenge I: Pointer Tracking. Re-randomization changes the code layout so existing pointers would become invalid after a randomization cycle. To retain the originally refereed objects, previous work tracks and updates the pointers via heavy instrumentation [36], [38]. To reduce the extra run-time overhead caused by pointer tracking, in HARM, we use an indirection layer to encode pointers into fixed indexes to a reference table. Combining off-device binary rewriting and simple run-time reference update, HARM eliminates the need for dynamic pointer tracking.

Technical Challenge II: Fixed Code Pointer Indexes. Although using an indirection layer effectively avoids the overhead of pointer tracking, fixed mapping between the code pointer and the target means attackers can still hijack the control flow if they can recover the mapping information (consider HARM as a coarse-grained CFI mechanism). To prevent this from happening, we make sure that the rewritten firmware, when leaked, can never disclose such mapping information. To defeat brute-force attempts that learn the mapping gradually, HARM triggers a complete binary rewriting once it detects an unexpected reboot. Through complete binary rewriting, HARM directly updates the mapping information stored in the secure flash. This will nullify prior efforts in the brute-force attack. Although a complete re-randomization consumes more time, it only happens during abnormal rebooting.

Technical Challenge III: Bookkeeping of Confidential Metadata. Unlike full-fledged OSs, MCU firmware runs on single address space, meaning that all the code including the user tasks and kernel (if any) are mixed. This makes it quite challenging, if not impossible, to *conceal confidential information* such as the bookkeeping of mapping information. HARM leverages TrustZone to address this problems. Specifically, we run the secure runtime to isolate the confidential information from the firmware with strong hardware-level guarantee. An alternative is to place the HARM runtime in privileged mode, and the target firmware in unprivileged mode, with the help of MPU. However, this not only requires refactoring existing code, but also necessitates applying *Software Fault Isolation* (SFI) [52] to sandbox the untrusted interrupt handlers (which must be executed in privileged mode). Using MPU removes HARM's dependence on TrustZone but incurs higher performance overhead since SFI performs less optimally on ARM Cortex-M [33].

Technical Challenge IV: Interrupt. Traditional software rewriting tools assume a user-space view in which the execution is oblivious to interrupts. However, MCU firmware mixes user tasks and interrupt handlers (or OS functions) together. This brings about unexpected control flow deviations that would cause system crashes if not handled properly. Our approach automatically identifies exception handlers and applies instrumentation correspondingly. Being interrupt-aware, HARM transparently supports multi-tasking firmware powered by RTOSs.



(a) Off-device analysis and rewriting

(b) On-device re-randomization (for each memory region, we mark the access permissions for normal world and secure world respectively)

Figure 2. HARM workflow

3.3. Design Overview

HARM features continuous function-level re-randomization as a moving-target mitigation to CRAs for MCUs. It includes an off-device analysis and rewriting module, and a secure runtime module. To relocate functions around on-the-fly, we intentionally move the code section from the read-only flash memory to an SRAM region in the TrustZone normal world. We call this SRAM region a **sandbox**. Since the addresses of functions (and the data objects mingled with the code) are changed during re-randomization, we have to correspondingly update the code/data pointers that refer to them. Traditionally, tracking and updating the influenced pointers incurs significant performance overhead. In our solution, we leverage an offline analysis stage to pre-process the ELF-format firmware. The outputs include a refined firmware image and the accompanying metadata for fast reference adjustment. Our secure runtime, which resides in the TrustZone secure world, periodically issues re-randomization requests. In each re-randomization cycle, it shuffles all the functions in the sandbox and leverages the metadata to update the influenced pointers efficiently. We also leverage TrustZone protection [53], [54] to prevent the firmware in the sandbox from accessing MPU. Otherwise, DEP could be bypassed.

Off-device Analysis and Rewriting: As shown in Figure 2(a), the off-device analysis and rewriting module takes the linked firmware image (in ELF format) as input and outputs a refined binary and metadata (i.e., several tables) to facilitate efficient re-randomization. First, HARM parses the symbol table of the target image and

extracts the information about the location and size for each function. Then, it disassembles each function and identifies location-sensitive instructions. We must accommodate these instructions to keep the firmware valid after each re-randomization. Specifically, HARM ensures that code/data pointers are adjusted to the valid ones prior to dereferencing. We employ two strategies to ensure this. For some instructions (e.g., direct calls), the adjustment is performed *on-device* during re-randomization, leveraging the metadata. For the rest (e.g., indirect calls and function returns), HARM directly overwrites them *off-device*, meaning that they do not need to be changed at run-time. This is enabled by explicitly invoking secure services. Treating location-sensitive instructions differently allows us to achieve optimized performance and reduce memory consumption.

On-device Re-randomization: As shown in Figure 2(b), the target firmware runs in the sandbox of the normal world and the secure runtime runs in the secure world. In this way, the secure runtime does not need to concern its own randomization. It has access to the metadata extracted off-device and is responsible for bootstrapping the firmware in the normal world and performing the continuous re-randomization. When the device powers up, the secure runtime performs the initial randomization, which copies the functions from the flash memory into the randomized locations in the sandbox. It also maintains a table to store the current address for each function. Then the firmware starts executing from the newly allocated reset handler. During device operation, the randomization is performed periodically, triggered by a non-bypassable secure timer. In each randomization cycle, after shuffling the functions and updating the tables, the secure runtime performs on-device reference adjustment *in place*. This process is assisted by the extracted metadata. The metadata is confidential, so it should not be leaked overtime under a persistent attack. Therefore, HARM also updates the metadata during each device reboot. Our hypothesis is that whenever the attacker tries to learn the mapping between a function and its index value via brute-force, the firmware is likely to crash due to a wrong target function being called (Section 6.3).

4. Off-device Binary Analysis and Rewriting

In this section, we explain binary rewriting, in particular, which instructions need to be rewritten, why they need to be rewritten, and how they are rewritten. The output of this module includes a refined binary and the accompanying metadata to assist efficient re-randomization.

4.1. Metadata

HARM manages functions by assigning each of them with a unique index. The metadata are maintained in six tables. Five of them are static and extracted during off-device analysis and one is dynamically constructed during firmware execution. All these tables are only accessible by the secure runtime. First, the *function table* records the location and size of each function in the flash memory. Second, the *direct call table* records the offsets of direct call instructions within each function. It is used for on-device direct call adjustment. Third, the *return address table* maps a function return address, represented by an index, into a function index plus the offset in this function. It is used by the secure runtime to recover the return addresses of function calls. During each randomization cycle, the secure runtime copies each function from flash according to the function table to a random location in the sandbox. The current addresses of the functions are maintained in the fourth table called *function dispatch table*, which is constructed and maintained dynamically at run-time. It is used by the secure runtime to calculate real code pointers, including indirect call targets and return addresses.

The remaining two tables are *indirect call table* and *function pointer table*, which record the offsets of indirect call instructions and function pointer initialization points respectively. These two tables are only referenced in an abnormal system reboot to update the indexes for functions. This prevents determined attackers from gradually learning the function index information (Section 6.3).

4.2. Binary Rewriting

When functions are shuffled, references to existing data/code objects become invalid. We collectively call the influenced instructions as *location-sensitive* instructions. Generally speaking, these instructions can be classified into two categories – those refer to absolute addresses and those refer to relative addresses. Absolute address accessing instructions include *indirect calls*, *function returns*, *exception returns*, (some) *jump tables*, etc. Relative address accessing instructions include *direct calls*, (some) *jump tables*, etc. HARM disassembles each function to obtain those instructions needing instrumentation. After applying instrumentation, the memory layout of the binary changes, which invalidates the offset fields of some instructions. Therefore, HARM performs a second round of rewriting to adjust these offsets. Note that the second round of rewriting preserves memory layout.

4.2.1. Code Pointer Encoding. The code pointers, such as the function pointers and return addresses, refer to absolute addresses. They become invalid when the locations of involved functions are changed. We address this problem by encoding these addresses to be invariable despite the continuous randomization. Specifically, a code pointer is represented as an index to the corresponding function and an offset within the function (i.e., `<index|offset>`). The corresponding dereferencing instruction is overwritten by a *secure call*, which transforms the encoded pointers into the actual address based on the aforementioned function dispatch table. Specifically, the actual address is calculated as `function_base(index) + offset`. For return address recovery, it has to go through an additional table called return address table to get `<index|offset>` before applying the result to above formula.

4.2.2. Direct Calls. A direct call invokes a function whose address is relative to the current PC. The offset is directly encoded as part of the call instruction, for example, the BL instruction, as shown in line 5 of Listing 1. After re-randomization, all the branching targets of BL instructions become invalid. In addition to branching to

the target, the BL instruction implicitly stores the return address into the link register (LR). The LR register is used by the callee to return to the caller. Since the caller might be moved, the return address in LR becomes invalid as well.

```
   10000: push {r4-r7, lr}
   ...
   1001A: mov r0, #1
   1001C: mov r1, #2
-  1001E: bl foo
-  10022: ...
+  1001E: movw lr, #ret_index ; LR=ret_index
+  10022: b foo
+  10026: ...
```

Listing 1. Direct Call Rewriting

We address this problem by replacing the BL instruction with two instructions as shown in Listing 1. First, to make the return address valid across randomizations, we encode it (0x10026 in the example) with a unique return address index (ret_index) to the LR register (line 7). The return address table maintains the mapping information from ret_index to the corresponding return site encoded by <index|offset>. Note that all of these can be determined statically. In Section 4.2.4, we discuss how the callee recover the real return address from the encoded LR register. It is also important to note that if we directly encode the return address as <index|offset> in instrumentation, the mapping information between function indexes and functions is immediately leaked, making CRA possible (see Section 6.3). Adding an indirection layer like return address table prevents this from happening. Second, the branching instruction is replaced with a B instruction which directly jumps to the target. The secure runtime adjusts the offset fields in the B instructions after each randomization, based on the direct call table and the current function dispatch table.

4.2.3. Indirect Calls. Indirect calls are always translated to "BLX Rn" in ARM (line 6 of Listing 2), where the branching target is stored in register Rn. In HARM, we encode function pointers at their initialization points. With this design, the encoded function pointers can propagate as before and a BLX instruction always jumps to an encoded pointer, which is invalid from the viewpoint of the processor. Thus, code pointers need to be decoded before dereference.

```
   10000: push {r4-r7, lr}
   ...
   1001A: ldr r3, foo
   1001C: mov r0, #1
   1001E: mov r1, #2
-  10020: blx r3
-  10024: ...
+  10020: movw lr, ret_index ; LR=ret_index
+  10024: mov r12, r3        ; R12=R3
+  10028: b secure_indirect_call_veneer
+  1002C: ...                ; position of foo return
```

Listing 2. Indirect Call Rewriting

To find out the initialization points of function pointers, HARM searches the relocation table for entries of type R_ARM_ABS32. Once HARM finds such an entry and confirms that it corresponds to a real function pointer, the pointer value in the binary is encoded. For an indirect call target, the offset field of the encoding is always zero because the target always points to the start of a function. To decode function pointers at dereference points, the "BLX <Rn>" is replaced by two instructions (line 9 and 10 in Listing 2). In line 9, we assign the *intra-procedure call scratch register* R12 with the encoded pointer. In line 10, the R12 register is used as the parameter to the veneer function that calls secure service to jump to the real target (Section 4.2.5). Similar to direct calls, the BLX instruction also implicitly stores the return address into the LR register. We follow the same design to encode the return address, as indicated in line 8.

4.2.4. Function Returns. ARM does not have a dedicated return instruction but relies on popping the return address from the stack or jumping to the return address in the LR register directly. Typically, in the prologue of a non-leaf function, the callee-saved registers and the LR register are pushed onto the stack. In the epilogue, callee-saved registers are restored from the stack, and the saved LR register is directly popped to PC, resulting in a return from the callee. The process is shown in line 2 and 4 of Listing 3. Since LR is encoded, we cannot directly pop it from the stack. We rewrite the epilogue by (1) removing PC from the popped register list (line 5), (2) manually retrieving the saved LR from the stack (line 6), and (3) branching to the veneer function that calls the secure service to jump to the real return address (line 7). A leaf function typically returns using the instruction "BX LR" without the stack. We directly replace this instruction with a call to the veneer function.

```
-- Case 1: Non-leaf function --
  10000: push {r4-r7, lr}
  ...
- 10030: pop {r4-r7, pc}
+ 10030: pop {r4-r7}       ; remove PC from pop list
+ 10032: ldr lr, [sp], #4  ; retrieve LR from stack
+ 10036: b secure_return_veneer

-- Case 2: Leaf function --
  ...
- 10030: bx lr
+ 10030: b secure_return_veneer
```

Listing 3. Function Return Rewriting

4.2.5. Secure Call Veneers. As shown in Listing 2, 3 and, we use the B instruction to branch to veneer functions (secure_indirect_call_veneer and secure_return_veneer) as a springboard to secure services. We discuss how secure runtime can decode the code pointers and jump to the intended addresses in Section 6.2. The reason of using the veneer functions is that the addresses of secure services are typically far from the normal world, and the offset cannot be encoded into a single B instruction (the offset must be within $\pm$16MB). In veneer functions, we invoke indirect calls to the secure services so that the offset limitation can be removed. Note that veneer functions are written in assembly and is position-independent. Therefore, they can also be randomized.

4.2.6. Jump Tables. The jump table can be implemented using either the *Table Branch* instructions or memory load (LDR) instruction, depending on the optimization level and the table size.

Using Table Branch Instructions: There are two table branch instructions, namely TBB (Table Branch Byte) and TBH (Table Branch Half-word). Table branch instructions

(e.g., `TBB <Rn, Rm>`) cause a `PC-relative` branching based on a table of offsets. The location of the table is specified by `Rn`, while the index of the table is specified by `Rm`. `TBB` and `TBH` are very similar except for the size of table entries. In `HARM`, the real offset to the target might change due to the instrumentation. However, since the offset is relative, we can statically fix the references in the table. We illustrate an example of using `TBB` to implement jump tables in Appendix A.

Using Load Instructions: A limitation with jump table implementations using table branch instructions is that the reachable offset is limited (510 bytes for `TBB` and 131,070 bytes for `TBH`). Also, they can only branch forward. To address these limitations, based on our empirical study, compilers often emulate the jump table using normal load instructions. Specifically, the jump targets are indicated with a table of *absolute addresses* as shown in Listing 4.

```
1    10034: mov r4, #0          ; default case
2    ...
3  - 10050: add r3, pc, #0    ; r3 points the table
4  - 10052: ldr pc, [r3, r2, lsl #2]
5  - 10054: .word 0x10060+1   ; branch to case 0
6  - 10058: .word 0x10064+1   ; branch to case 1
7  - 1005C: .word 0x10034+1   ; branch to default case
8  - 10060: ldr r4, [r0]        ; case 0
9  - 10062: b 0x10066
10 - 10064: ldr r4, [r0, #4]    ; case 1
11 - 10066: ...
12 + 10050: movw r3, #0         ; R3[15:0]=0
13 + 10054: movt r3, #2         ; R3[31:16]=0x2
14 + 10058: ldr r3, [r3, r2, lsl #2]
15 + 1005C: add pc, r3
16 + 1005E: ldr r4, [r0]        ; case 0
17 + 10060: b 0x10064
18 + 10062: ldr r4, [r0, #4]    ; case 1
19 + 10064: ...
20   ...
21 + 20000: .word 0xFFFFFFFE+1 ; branch to case 0
22 + 20004: .word 0x00000002+1 ; branch to case 1
23 + 20008: .word 0xFFFFFFD4+1 ; branch to default case
```

Listing 4. Rewriting Jump Table Using Load Instructions

In the example, the register `R3` is loaded with the base of the table (`line 3`), which is `0x10054`. The table contains absolute addresses of the jumping targets (line 5-7). Then a table entry indexed by `R2` is loaded to `PC` to jump to the target (`line 4`). Unfortunately, the base of the jump table (`R3`) becomes invalid after a randomization cycle, and thus `line 4` would branch to an unpredictable location. We solve this problem by fixing the location of the jump table and making the jump table entries position-independent. More specifically, we follow the steps below. (1) Move the jump table to an unmapped fixed location (e.g., `0x20000`). (2) Use the `MOVW` and `MOVT` instructions to load the base address of the jump table (`line 12-13`). (3) Load the table entry (`line 14`). (4) Add the value of the loaded entry to `PC` to branch to the real target (`line 15`). (5) Correct the jump table and make it position-independent. Specifically, the new entry should be the offset from the target to the address of the instruction "`ADD PC, R3`" (`line 21-23`).

The rewritten jump table entries are shown in `line 21-23` of Listing 4. After rewriting, the jump table becomes position-independent and thus the entries do not need to be updated during the randomization. We place the table in the read-only flash to reduce memory consumption and prevent memory corruption on it.

4.2.7. Reference Adjustment. The aforementioned binary rewriting is not in place. That is, the lengths of new instruction sequences are not equal to that of the original instruction sequences. This inevitably changes the memory layout of the target binary and invalidates several references. We run a second round of binary rewriting to fix these issues. Reference adjustment does not introduce new changes to the memory layout.

Internal Branches: The internal branches jump relatively inside a function. These instructions include `B`, `CBNZ`, `CBZ`, `TBB`, `TBH`. We overwrite the corresponding offsets encoded in the instructions.

Literal Pools: Restricted by the instruction length, ARM instructions cannot encode arbitrary immediate values. The *literal pool* is a widely used workaround that embeds arbitrary constants within the text section. ARM code then uses PC-relative addressing to load these values instead. With the changed memory layout, we have to adjust these offsets correspondingly. Moreover, literals themselves must be word-aligned, which might be violated by the instrumentation. `HARM` meets this requirement by padding the two-byte `NOP` instructions before the literal pool.

Read-only Data: The read-only data, such as the string constants or other global constants are placed at the end of the text section. The absolute addresses of these data objects are placed in the literal pool so that the program can refer to them indirectly. We have to adjust these absolute addresses to reflect the new memory layout. Similar to identifying function pointers, we search the relocation table for entries that (1) are of type `R_ARM_ABS32`, and (2) refer to the read-only data region.

Data Section: In the MCU firmware, the data section needs to be copied from the flash to the SRAM for write accesses. This is typically completed during device booting and the firmware relies on the symbol `_sdata` or `_etext` to identify the location of the data section. Since the expansion of the text section may overlap with the data section, we have to move the latter to a higher address and correspondingly update the symbol of `_sdata` and `_etext`.

4.2.8. Power Saving Mode Support. One of the defining characteristics of microprocessors is their ability to compute with a small amount of energy, and to aggressively optimize on energy, the device can be placed into sleep mode frequently. If randomization frequently disrupts the sleep mode, such benefit diminishes. `WFI` (i.e., Wait For Interrupt) is the special instruction that the system software uses to put the system into deep sleep until an interrupt arrives. To prevent the secure timer from immediately waking the system, `WFI` instructions are instrumented to invoke the energy service in the secure runtime via a secure call veneer named `secure_wfi_veneer` to allow for secure enter and exit of energy saving state.

5. Interrupt Support

So far, as with other traditional binary rewriting tools, `HARM` instruments the firmware following the semantics of instructions, ensuring the same logic with the original one. However, interrupt, which is indispensable for MCU devices, conflicts with `HARM` in terms of code pointer encoding. Before presenting these conflicts, we briefly

review how interrupt works. As described in Section 2.1 and Figure 1, when an exception happens, the hardware pushes processor context before the exception into the post-exception stack, so-called exception frame. Exception return is just a normal return to EXC_RETURN, which triggers the hardware to restore the context to the original mode.

When an exception takes place, the exception return address on the exception stack is an absolute value. If a re-randomization is performed in the middle of interrupt handling, the return address becomes invalid. Directly resuming from it crashes the execution. To solve this, we identify all the exception handlers and instrument the handler entries to invoke a secure service that encodes the exception return addresses as <index|offset> on the exception stack.

When exception returns, the encoded return address cannot be used to load PC directly. We need to correspondingly decode it. Since an exception handler is also a function that has been instrumented following the rules of function return, secure_return_veneer mentioned in Section 4.2.4 is invoked. However, the encoding of function return address and exception return address are different (ret_idx in the LR register vs. <index|offset> on the exception stack). The veneer function secure_return_veneer therefore checks whether this is an exception return or a normal function return (i.e., if the LR register is EXC_RETURN or not) and treats them correspondingly. More details are provided in Section 6.2.

Supporting interrupts allows the firmware to respond to external events timely. More importantly, it is the foundation for implementing OS functions such as task scheduling. Supporting OSs is a remarkable feature of HARM, considering that OSs are being widely used to meet the multi-tasking requirements of IoT development. In Appendix D, we showcase how HARM transparently supports task scheduling in MCU OSs.

6. Secure Runtime

The secure runtime executes in the TrustZone secure world, and thus is isolated from the sandbox in the normal world. It configures security-related registers to separate secure and non-secure resources, enforces DEP for the sandbox via MPU, and disables MPU access from the normal world. It also performs re-randomization periodically, maintains current function locations, adjusts offsets of direct call instructions, and provides secure services to encode/decode code pointers. When system reboots, it needs to re-randomize the indexes for functions.

6.1. Firmware Randomizer

The firmware randomizer is the core of the secure runtime. It is responsible for shuffling the firmware functions periodically. The randomization is triggered by the secure SysTick exception, which is essentially a programmable timer. Note that secure exceptions are non-bypassable by the normal world. Inside the secure SysTick handler, HARM makes sure that the processor was not serving a secure function (because a re-randomization invalidates the result of the secure function). If this is satisfied, the randomizer performs randomization as follows. (1) Retrieve the suspended address (i.e., PC) from the exception stack frame and encode it as <index|offset>. (2) Based on the function table (see Section 4.1), randomizer copies each of the functions in the flash to a random location in the sandbox. Meanwhile, the function dispatch table, which maintains the current location of each function, is updated. (3) Adjust the offset fields of the direct call instructions based on the direct call table and the new function dispatch table (see Section 4.2.2). (4) Based on the new function dispatch table, decode <index|offset> as the new address to resume execution. The new address is stored in the PC slot of the exception stack frame.

6.2. Secure Services

As illustrated in Section 4.2, HARM relies on services provided by the secure runtime to enable power saving mode and decode code pointers, including indirect call targets, function return addresses, and exception return addresses. The secure services are implemented as NSC veneer functions (see Section 2.2.1). They are written in pure assembly for optimized performance. Moreover, they are declared as naked to prevent the compiler from generating extra code. Our assembly code ensures that no unnecessary context maintenance code is emitted.

Indirect Call Target Decoding: It is called by secure_indirect_call_veneer in Listing 2. It uses the scratch register R12 to pass the encoded code pointer. After extracting the function index (offset is always zero), it looks up the function dispatch table for the current address of the target function. The result is stored in R12, which is used to branch to the target function via "BXNS R12".

Return Address Decoding: It is called by secure_return_veneer in Listing 3. If LR is not equal to EXC_RETURN, it is a normal function return and LR contains ret_idx. HARM queries the return address table to get the index of the caller function and the offset, following by looking up the function dispatch table to calculate the actual return address. The result is loaded to register LR. Finally, it branches to the intended caller function via the "BXNS LR" instruction. If LR is equal to EXC_RETURN, indicating an exception return, HARM locates the exception return address on the stack and performs decoding using the current function dispatch table directly. Then an exception return can branch to the intended address.

Exception Return Address Encoding: This secure service is called at the entry of exception handlers. It encodes the exception return address as <index|offset> on the exception stack, which is later decoded by secure_return_veneer on exception returns.

Power Saving Mode Support: When secure_wfi_veneer is invoked, this service disables interrupts and suspends the firmware randomizer. Subsequently, it turns the MCU into sleep mode via a WFI instruction. When the MCU is woken up by an interrupt, it re-enables the secure SysTick exception to resume the firmware randomizer. Finally, it enables interrupt to handle the pending interrupts.

6.3. Function Index Re-randomization

Encoded Code Pointers: In a control flow hijacking attack, control-data (data that are loaded to `PC` at some points) include the function pointers, return addresses on the stack, and the return addresses on the exception frame. They are encoded as `<index|0>`, `<ret_idx>` and `<index|offset>` respectively. Their leakage does not reveal anything about code itself, defeating JIT-ROP like attacks. However, the attacker does have the capability to overwrite the control-data with a malicious but validly encoded data (e.g., a different index). If that happens, the firmware blindly invokes an unknown function or returns to another valid return site (i.e., any instruction following a function call). To really control the hijacked program to run intended code, like return-to-libc or ROP, the attacker needs to further determine the mapping information between the indexes and functions. `HARM` makes sure that such mapping information cannot be inferred from the leaked code, and leverages TrustZone to prevent the mapping table from being directly read out. This forces attackers to brute-forcing.

Persistent Brute-force Attacks: Our function-level randomization works by shuffling functions, resulting in sufficient entropy even within a small address space (SRAM). Entropy is measured by possible indexes of a function. In the current prototype, the search space for one function is therefore 2^{16}. With a control-flow hijacking vulnerability, a determined attacker could try all possible index values. Such attempts can be persistent. It lasts until useful function mapping information is recovered to launch a CRA attack as mentioned before.

Solution: We observe that when a brute-force trial fails, the firmware is highly likely to crash. Due to the watchdog mechanism on MCUs, this will eventually lead to a device reboot. Therefore, `HARM` re-randomizes the function indexes on each unexpected device reboot. This process is assisted by the *indirect call table* and *function pointer table* following steps below: (1) The *function table* is shuffled and the *return address table* is updated correspondingly to reflect the new indexes of caller functions. (2) All function pointer initialization points are updated through the *function pointer table* to reflect the new indexes. (3) All entries of the *return address table* are shuffled and all `movw lr, #ret_index` instructions are rewritten to reflect the new return indexes. The offsets of `movw` instructions can be determined through the *direct call table* and *indirect call table*. Note that the flash memory usually has limited erase cycles. Our design would shorten the service life of the MCU chip under persistent attacks. This problem can be alleviated by switching to a fail-safe operating mode instead of constant rebooting. The fault handlers can be readily instrumented to trigger the fail-safe mode. We leave the implementation of fail-safe mode firmware as our future work.

7. Implementation

We have implemented a `HARM` prototype for the NXP LPC55S69-EVK development board [55], which is equipped with a dual-core ARM Cortex-M33 MCU with TrustZone extension running at up to 150 MHz. The board is equipped with a 640 KB on-chip flash memory, a 320 KB SRAM, and numerous peripherals including a microSD slot, a Full-Speed USB interface, etc. We note that our implementation does not depend on any board-specific features and therefore can be easily ported to other ARM MCUs with TrustZone extension.

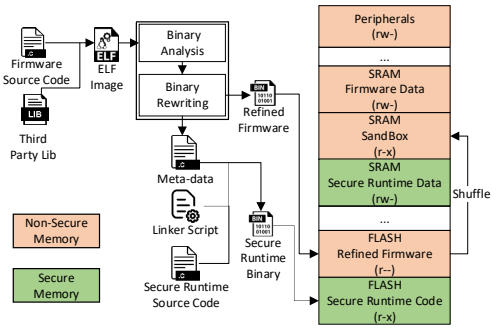


Figure 3. Workflow of `HARM` Prototype

The off-device binary analysis and rewriting module is based on several open-source projects and we contributed ~2,000 new lines of Python code. The secure runtime comprises of ~1,200 lines of C code and ~558 lines of assembly code. In Figure 3, we show the workflow of our prototype.

7.1. Binary Analysis and Rewriting

Binary Analysis: We used `pyelftools` [56] to parse the ELF-format firmware and `Capstone` [57] to disassemble instructions. We extracted the following information: (1) we obtain the base/size information of functions to construct the function table; (2) we find the direct call instructions to construct the direct call table; (3) we find function call instructions to construct the return address table; (4) we find the initial vector table from address zero and recognize all the exception handlers to instrument the handler entries; (5) from the relocation table, we identify the initialization points of function pointers and global read-only data.

Binary Rewriting: To better manage the complex reference relationship among different instruction/data objects, we followed the idea of internal representation (IR) proposed in `RevARM` [58]. Specifically, we represent an instruction/data object as a class that contains relevant properties obtained during binary analysis. For example, an object of the call instruction includes the information about whether it is a direct call or an indirect call, and also a reference to the calling target. In this way, we can focus on manipulating the IR object without worrying about the assembly syntax.

After the instrumentation information has been incorporated in the IR objects, `HARM` recalculates the address for each IR. This also updates the memory layout of the firmware. The alignment is also performed by inserting or removing the `NOP` instructions. The final step is to translate the IR into the machine code. For this purpose, we implemented an IR method that outputs the assembly code for an object, and then we use `Keystone` [59] to assemble them. The result is a *Binary Large OBject* (BLOB) without any ELF header information. It can be directly downloaded into the target device. Along with

this process, a C header file containing the metadata used by the secure runtime is generated. It is compiled into the secure runtime (shown at the bottom of Figure 3).

7.2. Secure Runtime

The secure runtime runs in the TrustZone secure world. It sets up the environment of the sandbox and periodically enforces re-randomization.

Bootstrapping: When the device powers up, it runs in the secure world and performs a function index re-randomization using In-Application Programming (IAP) [60]. Then, it fetches the reset handler from the vector table at address zero. It then starts booting from the reset handler, which initializes the hardware, partitions the memory, and configures the access permissions to restrict the sandbox. Subsequently, it shuffles the function table, return address table and rewrites the corresponding function pointers and instructions by invoking the flash driver resides in the boot ROM. Before giving control to the firmware, it performs the first randomization. To get a random seed, `HARM` polls a random number from the RNG on the board. Based on this seed, it copies each function in the flash into a random location in the sandbox. Correspondingly, the secure runtime updates the vector table based on the new addresses of exception handlers and moves it to a random location. Finally, the normal-world `VTOR` is updated according to the new address of vector table. At this stage, a new function dispatch table is also created.

Before giving control to the firmware, the secure runtime sets up a timer using `SysTick`. This is used as the trigger for periodical re-randomization requests. We must ensure that it has a higher priority than any exceptions in the normal world. This is done by de-prioritizing all non-secure exceptions by setting bit 14 (i.e., `PRIS`) of the *Application Interrupt and Reset Control Register* (AIRCR). Finally, the secure runtime switches to the normal world where the reset handler of the firmware is executed. During normal operation, the randomizer is triggered periodically by the secure timer. In each re-randomization cycle, the randomizer does the same thing as the first randomization.

Memory Partitioning & Security Enforcement: As part of bootstrapping, we used the SAU registers to partition the memory space into secure one and non-secure one. Besides the secure runtime and metadata, the secure memory space also includes the boot ROM and AHB Secure Controller [53], which is used to configure peripheral accesses. Note that SAU can only be accessed in the secure world and therefore cannot be manipulated by the firmware. Moreover, we used MPU to enforce DEP for the data regions of the firmware. To prevent attackers from disabling DEP by writing to the MPU registers, `HARM` disables the write access to the non-secure MPU registers from the normal world via the AHB Secure Controller. We also disabled execution permission for the flash memory, which cannot be randomized, to prevent firmware from reusing the code in there. The peripherals are exclusively used by the firmware. Therefore they are readable and writable in the normal world. The address information of each region is defined as symbols in a linker script. The secure runtime uses it for configuration during booting. We depict the access permissions, property, and the usage for each memory region in the right of Figure 3.

Table Lookup: The secure runtime also needs to respond to secure calls frequently. To encode/decode the code pointer, they need to look up the function dispatch table. In our prototype, we implemented a *Red-Black Tree* to store the key-value pair `<address, index>` for efficient table lookup. Therefore, the lookup overhead increases only logarithmically with the number of functions.

8. Evaluation

We evaluated the `HARM` prototype against real-world applications as well as two popular benchmarks. We are interested in the following questions. (1) What is the minimal re-randomization frequency to prevent a remote attacker from conducting JIT-ROP like attacks? (2) What is the highest re-randomization frequency `HARM` can support before the performance overhead becomes unacceptable? (3) How much code is added by instrumentation? (4) Will `HARM` consume more power than the original firmware? What if the firmware supports power saving mode?

We selected five real-world firmware images, covering various IoT application scenarios. PinLock was used in many other MCU research projects [22], [29], [46]. It simulates a smart lock that accepts user passcode via a serial port. The SHA-256 of the passcode is embedded in the firmware and is used to check the validity of the unlocking request. U-Disk and FatFs-uSD are provided with the SDK of the development board [61]. U-Disk simulates a USB disk peripheral, which can be enumerated by a PC. FatFs-uSD initializes and accesses a FAT file system on a microSD card inserted on the board. Both of them are provided with two versions – bare-metal and FreeRTOS. We developed MQTT-Echo by ourselves. It implements an MQTT client that subscribes the "echo" topic on the cloud server and echoes any received message back. MQTT-Echo is based on the `MQTT-C` library [62], a popular MQTT implementation for embedded systems. We used Iperf [63] as a benchmark to measure the network throughput. It was ported by NXP to its chips. Both MQTT-Echo and Iperf use FreeRTOS as the underlying OS. Two benchmarks we used were BEEBS [64] and CoreMark [65]. The former includes a wide range of programs to test the performance of deeply embedded systems and the latter is an industry-standard benchmark that measures the overall computing performance for MCUs.

All the firmware were compiled by the ARM GCC toolchain with the `-Os` optimization level, the most common compiler option in MCU that minimizes code size.

8.1. Re-randomization Frequency

We assume a powerful attacker who can read out the whole code contents and launch a JIT-ROP fashion attack. To gain a perception of how fast this can be against MCUs, we experimented with a real board under idealistic settings. As mentioned before, we cannot craft the ROP chain locally on the MCU as was done in the original JIT-ROP paper [30], since MCUs lack the scripting environment on Windows/Linux. However, we made efforts to set up the experiment environment in favor of the attacker. For

example, the memory disclosure bug allows for leaking an arbitrary length of data at a time and the attacker was assumed to know the exact location of the sandbox. In our LAN, RTT was less than 2 ms and TCP bandwidth was ~6 Mbit/s (speed limited by the NIC of MCU). The attacking PC has an Intel Core i7-8750H CPU with 16 GB RAM.

We intentionally brought in a memory disclosure bug and a buffer overflow bug to the firmware. We remotely triggered the memory disclosure bug using the PC in the LAN to steal all the code in the sandbox (64 KB). Attackers may only need to steal part of the code on demand, however, it would involved multiple rounds of communications and take longer. Then, on the PC, we used `ROPGadget` [66] to find and compile a ROP chain that disables DEP protection and jumps to an infinite loop on the stack. After sending the exploitation payload back, the device hung as expected.

The attack took approximately 600 ms to complete, starting from triggering the memory disclosure bug to crashing the stack. The 600 ms roughly consisted of 150 ms for computation (disassembling and finding ROP chains) and 450 ms for network transmission. We note the majority of the time was spent on network, so it is unlikely to significantly reduce the measured time even with a more powerful attack PC. Our experiment simulated an idealistic environment for attackers and thus the result was conservative. This gives us a guideline of how to set up re-randomization frequencies in the following evaluation experiments: as long as we do re-randomization more often than every 600 ms, `HARM` defeats real-world JIT-ROP like attacks. In another experiment, we ran the attack script on a remote machine located in New York (over 1,300 km away from our experiment site), the attack took more than 1,200 ms (RTT was 24 ms).

8.2. Code Size & Memory Overhead

We measured the number of additional instructions inserted during binary rewriting, and provided a break-down in Table 1. Compared to the uninstrumented baseline, the overall code size increases by less than 16.0% in all the samples. Direct call instrumentation incurs the most overhead, taking approximately 78% of the whole overhead. This is expected considering the frequent uses of function calls. There is an identical copy of secure API veneers for each firmware. However, we observed two additional bytes in CoreMark. This was caused by a padding of a `NOP` instruction to make the literal pool word-aligned. Our approach wastes the normal-world flash memory because the code there is moved to SRAM for execution. However, we did not observe a shortage of memory in our evaluation with real-world samples. Moreover, many MCUs come with an external DRAM interface, which can alleviate the problem. Our instrumentation does not introduce any overhead on run-time memory such as stack and heap.

The secure runtime resides in the secure flash. It consists of a fixed code size (19 KB) plus variable sized tables (i.e., metadata) compiled from the header files extracted from the firmware. Table 2 shows the number of functions and the size of metadata for each sample. As can be seen, the latter grows with the former.

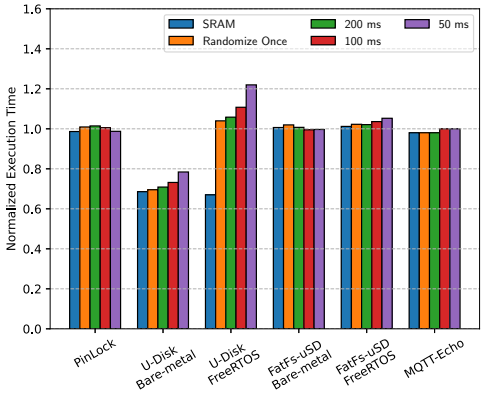


Figure 4. Normalized Execution Time under Different Re-randomization Frequencies (the lower, the better).

8.3. Performance

Depending on the nature of the application, we designed firmware specific experiments to measure the run-time performance overhead. For PinLock, we followed the same method as in *μRAI* [22]. Specifically, we timed the whole process of receiving 1,000 PINs that alternate between incorrect PINs, a correct PIN, and a locking command that requests the PIN again. For U-Disk (both bare-metal and FreeRTOS), we connected the board to the host through the USB interface. After the PC recognizes the board as U-Disk, it formats the U-Disk with the FAT32 file system and copies a file of 10 MB to the board. We measured the time spent on copying. For FatFs-uSD (both bare-metal and FreeRTOS), the firmware creates a file system on the SD card. Then it creates a new file and writes 100 bytes of random data to that file. Finally, it reads back the contents and checks whether they are the same. We measured the time from creating the file system to the completion of content verification. For MQTT-echo, we measured the time from receiving the subscribed message to echoing it back. For Iperf, we recorded the throughput. For CoreMark, it directly gave us a synthesized score. Finally, we recorded the execution time for BEEBS.

We compared the results among the *baseline*, *SRAM*, *randomize once*, and different frequencies. The *baseline* was measured with the original firmware on the flash. *SRAM* was measured when the original firmware is copied into the SRAM for execution. *Randomize once* was measured when the *instrumented* firmware is copied to the sandbox, but without any further re-randomization. The randomization periods we chose were 200 ms, 100 ms, and 50 ms. All of them are far less than 600 ms that we measured in the aforementioned real exploitation. Note except for *baseline*, the code is held in SRAM for all the other settings. We ran each experiment 20 times and calculated the average. Since PinLock, U-Disk, FatFs-uSD, MQTT-echo and BEEBS were measured by the execution time, we demonstrate the normalized results in Figure 4 and Table 4. For Iperf and CoreMark, we show the raw data in Table 3.

Generally speaking, for all the firmware, the performance degrades when the re-randomization frequency increases. When the randomization frequency is below 5

Table 1. BREAK-DOWN OF CODE SIZE INCREMENT

	Baseline (bytes)	Direct Call (bytes)	Indirect Call (bytes)	Function Return (bytes)	Secure API Veneers (bytes)	Overall Overhead (%)
PinLock	12,584	996	12	167	32	10.92
U-Disk Bare-metal	24,470	2,096	50	340	32	11.68
U-Disk FreeRTOS	33,618	3,412	100	477	32	13.38
FatFs-uSD Bare-metal	27,608	2,624	36	340	32	12.21
FatFs-uSD FreeRTOS	33,928	3,856	38	293	32	13.30
Iperf	64,920	8,100	106	909	32	15.49
MQTT-Echo	68,684	8,372	80	922	32	15.04
CoreMark	21,864	1,676	50	219	34	10.05

Table 2. FUNCTION NUMBER AND METADATA SIZE

	Total Functions	Function Tab. (bytes)	Ret. Addr. Tab. (bytes)	Direct Call Tab. (bytes)	Indirect Call Tab. (bytes)	Func. Ptr. Tab. (bytes)
PinLock	198	2,772	1,016	2,418	64	4
U-Disk Bare-metal	311	4,354	2,192	5,100	156	100
U-Disk FreeRTOS	400	5,600	3,612	7,806	264	152
FatFs-uSD Bare-metal	297	4,158	2,693	5,664	120	20
FatFs-uSD FreeRTOS	377	5,278	3,948	8,058	132	64
Iperf	713	9,982	8,312	16,950	300	152
MQTT-Echo	752	10,528	9,252	18,636	264	160
CoreMark	201	2,814	1,776	3,702	136	4

Table 3. COREMARK AND IPERF PERFORMANCE UNDER DIFFERENT RE-RANDOMIZATION FREQUENCIES (THE HIGHER, THE BETTER).

	Baseline	SRAM	Rand. Once	200 ms	100 ms	50 ms
CoreMark (Points)	262.30	428.00	374.70	373.46	372.41	363.66
Iperf (Mb/s)	5.48	5.61	5.60	5.30	5.02	4.54

Table 4. BEEBS RESULTS (THE HIGHER, THE WORSE)

	SRAM (ms)	Rand. Once (×)	200 ms (×)	100 ms (×)	50 ms (×)
bubblesort	1,832	1.00	1.00	1.01	1.02
dijkstra	20,777	1.16	1.17	1.17	1.19
edn	1,305	1.01	1.02	1.02	1.04
fasta	2,915	1.01	1.01	1.02	1.03
fir	7,611	1.36	1.37	1.37	1.39
huffbench	8,960	1.00	1.01	1.01	1.02
levenshtein	1,324	1.07	1.08	1.08	1.09
matmulti-int	3,516	1.21	1.22	1.23	1.23
nettle-aes	2,033	2.03	2.06	2.07	2.09
picojpeg	22,851	2.04	2.05	2.05	2.07
qrduino	21,888	1.19	1.20	1.21	1.23
sglib-dllist	962	1.21	1.21	1.22	1.23
sglib-listinsertsort	1,270	1.22	1.23	1.23	1.24
sglib-listsort	2,419	1.08	1.08	1.09	1.10
sglib-rbtree	3,924	1.31	1.32	1.32	1.33
slre	1,056	1.80	1.88	1.88	1.90
sqrt	45,332	1.24	1.25	1.25	1.26
st	9,424	1.27	1.27	1.28	1.29
whetstone	75,860	1.17	1.17	1.18	1.19
Min	-	1.00	1.00	1.01	1.02
Max	-	2.04	2.06	2.07	2.09
Geomean	-	1.25	1.26	1.27	1.28

Hz (or the period is longer than 200 ms), the maximum overhead is merely 5.8%. Note that 200 ms is much less than the measured 600-ms cut-off point in a real JIT-ROP like attack. In U-Disk and CoreMark, we even observed improved performance. This is because the code in *baseline* runs in the flash memory whereas others run in the SRAM, which is faster in fetching instructions. This improvement actually outweighs the overhead introduced by instrumentation and randomization operations. We note that U-Disk[1] and CoreMark are computation/memory intensive. Therefore, instruction fetching significantly influences the execution speed. For other I/O intensive

1. U-Disk seems to be I/O intensive. However, we found that it copies large amount of data received from the USB buffer to the SD card buffer.

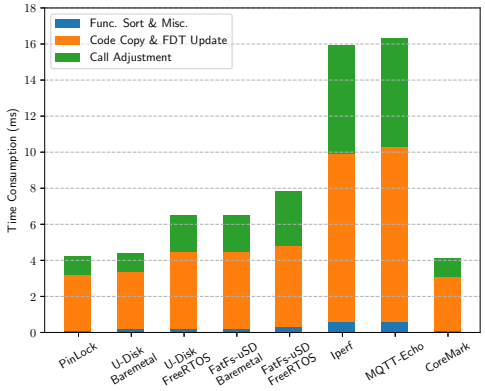


Figure 5. Overhead Breakdown in Each Randomization

applications, the performance of *SRAM* is similar to the `baseline`.

For BEEBS, we picked 19 programs that implement a wide range of algorithms involving integer/floating-point computations. For each program, we measured the execution time of running 4,096 iterations. The baseline was measured on SRAM. As shown in Table 4, the geometric means of the overhead were 25%, 26%, 27% and 28%, for each randomization frequency. However, we observed much higher overhead on `nettle-aes` and `picojpeg`. By investigating the source code, we found these programs continuously call simple functions in long loops, which incurs significant overhead since we instrument on every functions return.

Overhead Breakdown. We now breakdown the real overhead imposed by `HARM`. Comparing *SRAM* with *randomize once*, the difference directly reflects the actual overhead due to instrumentation and secure services because they both run on SRAM. The overhead of each firmware is 2.31%, 1.41%, 55.2%, 1.30%, 1.04% and 0% respectively. We did not observe huge gaps except for U-Disk-FreeRTOS. We attribute this to the frequent function return operations. In each function return, a secure service is needed to decode the code pointer. Comparing *randomize once* with *50/100/200 ms*, the difference directly reflects the overhead due to randomization. The overhead under

200 ms of each firmware is 0%, 1.92%, 1.77%, -1.12%, 0% and 0% respectively. As the frequency increases, we observed slow performance degradation. When the frequency reaches 20 Hz (the period is 50 ms), the maximum overhead we observed is 21% (U-Disk FreeRTOS).

We further zoom in on the overhead of randomization itself in Figure 5. The overhead mainly comes from 1) copying code from flash to sandbox and updating the function dispatch table; and 2) adjusting direct call offsets. The former overhead depends on the code size and the latter depends on the number of direct calls in the firmware.

Finally, we measured the time spent on world switching, which happens frequently when the firmware invokes secure services. As mentioned before, TrustZone-M highly optimizes the world switching [67]. Switching to the secure world can be achieved with a single `SG` instruction, which takes 3 CPU cycles based on chip manual (cf. §2.2 of [44]). To confirm this, we wrote a micro-benchmark that measures the time spent on a normal function call and a secure call. Both functions were empty and written in assembly to avoid any unexpected overhead. We used the `SysTick` counter to measure the CPU cycles. On average, the normal function takes 16 cycles while the secure function takes 19 cycles. This result agrees with the chip manual, indicating negligible overhead.

8.4. Energy Consumption

IoT applications are sensitive to energy consumption. In this experiment, we used a digital multimeter to profile energy consumption. We followed the instructions on the board manual [68] to collect instant current via pin P13 on the board. We recorded a reading every 500 ms and report the average in Table 5. The overall energy consumption does not vary a lot. Comparing *baseline* and *SRAM*, we observed slightly increased energy consumption. This indicates that SRAM consumes more power than flash when in the high-speed mode [69]. As re-randomization frequency increases, more time was spent by the secure runtime to do randomization. Since the secure runtime runs in the flash and flash uses less power, we observed slightly decreased energy consumption.

We further used the MicroTick timer demo provided by the SDK to evaluate the power saving mode support. The MCU entered into deep sleep mode (idle) and would be woken up (active) by a micro-tick interrupt every few seconds. When the power-saving mode was enabled, we observed that the power consumption was the same with the baseline when the firmware is either active (2.43 mA) or idle (1.85 mA). In contrast, when the power-saving mode was disabled, we observed high power consumption (2.43 mA) even if the firmware was supposed to sleep (i.e., 1.85 mA is expected), since the secure `SysTick` exception frequently woke up the MCU in a micro-tick cycle.

9. Discussion and Limitations

Due to the lack of code diversity mechanisms, CRAs including ROP and JIT-ROP like attacks impose an acute threat to the security of MCU devices. `HARM` provides an effective mitigation mechanism for CRAs by doing randomization periodically, at the cost of sacrificing some SRAM space to store code. The unique characteristics of MCU make CRA particularly easier to be conducted. In MCU devices, the memory is mapped to fixed locations and has smaller sizes. This means attackers can simply dump the code from fixed locations and then do gadget compilation. The good news for defenders is that in MCU devices, the attacker cannot search and compile the gadgets locally, because MCU firmware does not have the needed scripting environment. Since the leaked memory has to be sent out over the network, it takes much longer to conduct a full JIT-ROP like attack (~600 ms). Within such a long time window, `HARM` can complete a randomization cycle without significantly influencing the run-time performance. Leaking the functions indexes allows attackers to jump to another valid function, regardless of the code randomization. To deal with persistent attackers who want to brute-force such mapping information, `HARM` performs function index re-randomization on each device reboot.

`HARM` provides comprehensive randomization protection to all the executable code. First, `HARM` uses MPU to make sure no static code can be executed, including that in the flash memory. Second, the secure runtime module performs randomization continuously for all the functions in the sandbox, including the trampoline functions and interrupt handlers. Third, access to MPU is locked by TrustZone so DEP cannot be disabled.

Limitations: First, `HARM` currently focuses on control-flow protection and therefore cannot defend against data-only attacks such as DOP [70] and data-flow stitching [71]. Second, the additional instrumentation and periodical re-randomization bring about non-determinism which may violate the hard real-time constraints in some scenarios such as industrial control. Therefore, `HARM` is more suitable for applications without strict timing constraints, such as PinLock which we have evaluated. Third, `HARM` holds the code in SRAM to facilitate randomization, which implicitly doubles the RAM consumption for the code segment. However, there is no overhead on data segments. More memory overhead is imposed by having a secure runtime in the secure world. Fourth, our current strategy reboots the device whenever a crash is detected. Under persistent DoS attacks, the flash would wear quicker due to excessive re-programming. This problem can be alleviated by switching to a fail-safe operating mode instead of constant rebooting as mentioned before. Lastly, the soundness of our binary rewriting tool is challenged in the presence of many compilation options. We discuss several corner cases in Appendix C

10. Related Work

10.1. Code Re-randomization

Code re-randomization has been studied for a while. TASR [38] performs on-demand code re-randomization based on the monitoring of I/O events. The observation is that I/O operations are highly likely to be related to a JIT-ROP attack – output operations might be used to leak code, whereas input operations might be used to inject payloads. MARDU [37] is similar, but relies on abnormal program behaviors, e.g., process crash, to trigger re-randomization. RUNTIMEASLR [36] invokes

Table 5. AVERAGE ENERGY CONSUMPTION UNDER DIFFERENT RE-RANDOMIZATION FREQUENCIES (THE LOWER, THE BETTER).

	Baseline (mA)	**SRAM** (mA)	**Rand. Once** (mA)	**200 ms** (mA)	**100 ms** (mA)	**50 ms** (mA)
PinLock	8.70	8.70	8.70	8.70	8.67	8.69
U-Disk Bare-metal	15.05	15.34	15.73	17.70	15.68	15.65
U-Disk FreeRTOS	15.14	15.23	15.31	15.28	15.27	15.22
FatFs-uSD Bare-metal	14.11	15.14	15.11	15.10	15.08	15.05
FatFs-uSD FreeRTOS	13.76	13.80	14.41	14.37	14.36	14.34
Iperf	9.15	9.87	9.67	9.62	9.55	9.48
MQTT-Echo	9.13	9.11	9.31	9.20	9.16	9.08
CoreMark	8.01	8.67	8.79	8.77	8.77	8.73

a re-randomization when a child process is forked. Remix [41] performs live randomization at basic block level, minimizing the complexity of migrating stale code pointers. RERANZ [39] leverages a dedicated "shuffling process" to continuously replace old code with a fine-grained randomized code variant. Shuffler [40] implements asynchronous and continuous function-level code re-randomization on the order of milliseconds. These solutions work at different granularities (basic block vs. function vs. module) with different triggers (I/O events vs. program status vs. timer). However, they all share one similarity – they are designed for commodity platforms and depend on the rich runtime environment on these platforms, which is unavailable in low-end MCUs devices. HARM is the first re-randomization work for MCU. It does not rely on any runtime environment rather than the TrustZone security extension.

10.2. MCU Security

CFI: Apart from randomization, *Control-Flow Integrity* (CFI) [72] is another popular way to prevent control flow hijacking attacks. μRAI [22] is a compiler-based solution that enforces return address integrity by removing the need to spill return addresses to the stack. This is achieved by using direct jump instructions and a reserved register to determine the correct return location. Silhouette [45] also targets backward CFI, but introduces an incorruptible shadow stack and store hardening. CFI CaRE [73] leverages TrustZone to implement shadow stack mechanism. TZmCFI [74] extends it to further cover exception handlers. BARRA [75] is a variant of shadow stack which works with randomization. It protects the return addresses by replacing them with IDs that are re-randomized when the mapping table is leaked. BARRA shares the same idea of using an ID to encode return addresses, but it only covers backward-edge CFI. Forward-edge CFI has not been discussed a lot, partially due to its high performance overhead and false negatives [76]. C-FLAT [77] and OAT [51] provide remote attestation of control-flow for embedded systems. However, they are currently restricted to simple programs and require a remote verifier.

XOM: *eXecutable-Only-Memory* (XOM) is commonly used in conjunction with randomization solutions to defeat JIT-ROP attacks. It prevents code disclosure and therefore can defeat gadget collections. Many hardware XOM IPs have been used in different real devices [78]. uXOM [33] implements a software-based XOM for ARM MCUs leveraging special architectural features in Cortex-M. PicoXOM [34] leverages the debug unit of MCU to implement a lightweight XOM. XOM cannot prevent indirect JIT-ROP attack [31], [35] since gadgets can be inferred from code pointers in data sections. HARM covers both direct and indirect JIT-ROP via continuous randomization and code pointer encoding.

Others: MPU has been used a lot in literature to address security problems. Minion [1] identifies the reachable memory regions for each thread based on off-line static analysis and then uses MPU to enforce run-time memory access control. ACES [79] presents an LLVM-based compiler, which can infer and enforce memory compartmentalization automatically on bare-metal systems. Both Minion and ACES configure the MPU dynamically at run-time to switch to the needed memory access control setting to meet various demands and scenarios. HARM leverages MPU to enforce DEP, a basic assumption in randomization works. EPOXY [29] presents privilege overlaying to automatically identify operations requiring privilege and only allow them to run in privileged mode. The authors showcased its usage in many security mechanisms, including code integrity, CFI, and fine-grained randomization. The proposed randomization is performed per device-booting, making it vulnerable to JIT-ROP attacks. RevARM [58] is a fine-grained binary rewriting technique for ARM. It is for general-purpose binary instrumentation and can insert any logic at arbitrary location. Our implementation of internal representation (IR) is inspired by RevARM.

11. Conclusions

We propose HARM, the first continuous code re-randomization solution for MCUs. Our work relies on the ARM TrustZone extension to build a special runtime system for re-randomization. It periodically invokes non-bypassable randomization requests to the firmware execution. The firmware image is refined by the proposed binary analysis and rewriting tool to minimize the workload required to track and update code/data references. HARM is OS-agnostic. Therefore, it can be used with both bare-metal firmware and RTOS-based firmware. HARM supports multi-party development environment since no source code is needed. We implemented a HARM prototype for a real development board. With extensive evaluation, we conclude HARM provides mitigations to CRAs with moderate overhead.

Acknowledgment

We thank Dr. Kang Li from Baidu Security and Dr. Kyu Hyung Lee from UGA for their insightful comments. This work was supported in part by a grant from the University of Georgia Research Foundation, Inc.

References

[1] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching," in *NDSS*, 2018.

[2] O. Karliner, "FreeRTOS TCP/IP Stack Vulnerabilities – The Details," https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/, December 2018 (last accessed on: 8th March 2022).

[3] B. Seri, G. Vishnepolsky, and D. Zusman, "Critical vulnerabilities to remotely compromise VxWorks, the most popular RTOS," https://go.armis.com/hubfs/White-papers/Urgent11%20Technical%20White%20Paper.pdf, ARMIS, INC., Tech. Rep., 2019 (last accessed on: 8th March 2022).

[4] G. Beniamini, "Over The Air: Exploiting Broadcom's Wi-Fi Stack," https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017 (last accessed on: 8th March 2022).

[5] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 19–36.

[6] M. E. Garbelini, C. Wang, S. Chattopadhyay, S. Sumei, and E. Kurniawan, "SweynTooth: Unleashing Mayhem over Bluetooth Low Energy," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 911–925.

[7] Microsoft, "Data Execution Prevention (DEP)," http://support.microsoft.com/kb/875352/EN-US/, 2006 (last accessed on: 8th March 2022).

[8] M. Polychronakis and A. D. Keromytis, "ROP payload detection using speculative code execution," in *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, 2011, pp. 58–65.

[9] M. Prandini and M. Ramilli, "Return-oriented programming," *IEEE Security & Privacy*, vol. 10, no. 6, pp. 84–87, 2012.

[10] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," 2014.

[11] R. Qiao, M. Zhang, and R. Sekar, "A principled approach for rop defense," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 101–110.

[12] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-Oriented Programming on a Cortex-M Processor," in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 823–832.

[13] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou, "Detecting ROP with Statistical Learning of Program Characteristics," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ser. CODASPY '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 219–226.

[14] L. Luo, Y. Zhang, C. Zou, X. Shao, Z. Ling, and X. Fu, "On Runtime Software Security of TrustZone-M Based IoT Devices," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–7.

[15] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, Nov. 2009.

[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318.

[17] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[18] Alan Mujumdar, "Armv8.1-M Pointer Authentication and Branch Target Identification Extension," https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension, 2021 (last accessed on: 8th March 2022).

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "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.

[20] Brandon Azad, "Examining Pointer Authentication on the iPhone XS," https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019 (last accessed on: 8th March 2022).

[21] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 31–46.

[22] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing Embedded Systems with Return Address Integrity," in *Network and Distributed Systems Security (NDSS) Symposium*, 2020.

[23] P. Team, "PaX address space layout randomization (ASLR)," 2003.

[24] S. Priyadarshan, H. Nguyen, and R. Sekar, "Practical fine-grained binary code randomization," in *Annual Computer Security Applications Conference*, 2020, pp. 401–414.

[25] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-Assisted Code Randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 461–477.

[26] A. Gupta, J. Habibi, M. S. Kirkpatrick, and E. Bertino, "Marlin: Mitigating Code Reuse Attacks Using Code Randomization," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 3, pp. 326–337, 2015.

[27] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[28] H. Koo and M. Polychronakis, "Juggling the Gadgets: Binary-Level Code Randomization Using Instruction Displacement," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 23–34.

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

[30] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 574–588.

[31] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming," in *NDSS*, 2015.

[32] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. USA: IEEE Computer Society, 2014, p. 227–242.

[33] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient eXecute-Only Memory on ARM Cortex-M," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 231–247.

[34] Z. Shen, K. Dharsee, and J. Criswell, "Fast Execute-Only Memory for Embedded Systems," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 7–14.

[35] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 952–963.

[36] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make aslr win the clone wars: Runtime re-randomization," in *NDSS*, 2016.

[37] C. Jelesnianski, J. Yom, C. Min, and Y. Jang, "MARDU: Efficient and Scalable Code Re-randomization," in *Proceedings of the 13th ACM International Systems and Storage Conference*, 2020, pp. 49–60.

[38] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 268–279.

[39] Z. Wang, C. Wu, J. Li, Y. Lai, X. Zhang, W.-C. Hsu, and Y. Cheng, "Reranz: A light-weight virtual machine to mitigate memory disclosure attacks," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2017, pp. 143–156.

[40] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Nov. 2016, pp. 367–382.

[41] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the sixth ACM conference on data and application security and privacy*, 2016, pp. 50–61.

[42] Arm Holdings, "TrustZone technology for Armv8-M Architecture," https://developer.arm.com/documentation/100690/latest/, 2018 (last accessed on: 8th March 2022).

[43] S. LABS, "Silicon Labs 32-bit ARM Microcontroller Family," https://www.silabs.com/products/mcu/32-bit/arm-32-bit-microcontroller, 2021 (last accessed on: 8th March 2022).

[44] Arm Holdings, "ARM Cortex-M23 Processor Technical Reference Manual," https://developer.arm.com/documentation/ddi0550/c/, 2016 (last accessed on: 8th March 2022).

[45] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient Protected Shadow Stacks for Embedded Systems," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1219–1236.

[46] Z. Shen, K. Dharsee, and J. Criswell, "Fast execute-only memory for embedded systems," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 7–14.

[47] A. Khan, H. Kim, B. Lee, D. Xu, A. Bianchi, and D. J. Tian, "M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[48] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1416–1432.

[49] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 488–501.

[50] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "SeCloak: ARM Trustzone-Based Mobile Peripheral Control," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–13.

[51] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting Operation Integrity of Embedded Devices," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1433–1449.

[52] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *19th USENIX Security Symposium (USENIX Security 10)*. Washington, DC: USENIX Association, Aug. 2010.

[53] NXP, "7.20 AHB peripherals," https://www.nxp.com/docs/en/data-sheet/LPC55S6x.pdf, (last accessed on: 8th March 2022).

[54] STMicroelectronics, "12.3.4 SYSCFG CPU non-secure lock register," https://www.st.com/resource/en/reference_manual/dm00346336-stm32l552xx-and-stm32l562xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf, (last accessed on: 8th March 2022).

[55] NXP, "LPC55S69-EVK: LPCXpresso55s69 development board," https://www.nxp.com/design/development-boards/lpcxpresso-boards/lpcxpresso55s69-development-board:LPC55S69-EVK, 2019 (last accessed on: 8th March 2022).

[56] E. Bendersky, "pyelftools," https://github.com/eliben/pyelftools, 2021.

[57] N. A. Quynh, "Capstone engine," https://github.com/aquynh/capstone, 2021.

[58] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "RevARM: A platform-agnostic ARM binary rewriter for security applications," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp. 412–424.

[59] N. A. Quynh, "Keystone engine," https://github.com/keystone-engine/keystone, 2021.

[60] NXP, "Chapter 9: LPC55S6x/LPC55S2x/LPC552x Flash API," https://www.mouser.com/pdfDocs/NXP_LPC55S6x_UM.pdf, 2019 (last accessed on: 8th March 2022).

[61] ——, "MCUXpresso SDK Builder," https://mcuxpresso.nxp.com/en/welcome, 2021 (last accessed on: 8th March 2022).

[62] L. Bindle, "MQTT-C," https://github.com/LiamBindle/MQTT-C, 2021.

[63] "iPerf – The ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/, 2016.

[64] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, 2013.

[65] EEMBC, "Coremark," https://www.eembc.org/coremark/, 2021.

[66] J. Salwan, "ROPGadget," https://github.com/JonathanSalwan/ROPgadget, 2021.

[67] Arm Holdings, "Switching between Secure and Non-secure states," https://developer.arm.com/documentation/100690/0201/Switching-between-Secure-and-Non-secure-states, 2019 (last accessed on: 8th March 2022).

[68] NXP, "LPCXpresso55S69/55S28 Development Boards User Manual," https://www.mouser.com/pdfDocs/NXP_LPCXpresso55S69_LPCXpresso55S28_UM.pdf, 2019 (last accessed on: 8th March 2022).

[69] H. M. D. Kabir and M. Chan, "SRAM precharge system for reducing write power," *HKIE transactions*, vol. 22, no. 1, pp. 1–8, 2015.

[70] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 969–986.

[71] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of Data-Oriented exploits," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 177–192. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu

[72] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 559–573.

[73] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017, pp. 259–284.

[74] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M," *International Journal of Parallel Programming*, pp. 1–21, 2020.

[75] C. Zou and J. Xue, "Burn after reading: A shadow stack with microsecond-level runtime rerandomization for protecting return addresses**thanks to all the reviewers for their valuable comments. this research is supported by an australian research council grant (dp180104069)." in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 258–270.

[76] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 901–913.

[77] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 743–754.

[78] M. Schink and J. Obermaier, "Taking a Look into Execute-Only Memory," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019.

[79] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic Compartments for Embedded Systems," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 65–82.

[80] Arm Holdings, "Bare-metal Position Independent Executables," https://developer.arm.com/documentation/100748/0612/mapping-code-and-data-to-the-target/bare-metal-position-independent-executables, 2019 (last accessed on: 8th March 2022).

Appendix A. Table Branch Instruction Example

```
10050: tbb [pc, r3]
10054: .byte 0x4        ; branch to case 0
10055: .byte 0x6        ; branch to case 1
10056: .byte 0x8        ; branch to default case
10057: .byte 0x0        ; padding
10058: ldr r4, [r0]     ; case 0
1005A: b 0x1005E
1005C: ldr r4, [r0, #4] ; case 1
1005E: ...
```

Listing 5. Table Branch Byte (TBB) Instruction

Appendix B. Alternative Design Choices

Transparent Code Pointer Decoding: In the current design, we instrument indirect calls and function returns to explicitly invoke secure services that decode the code pointers. This instrumentation can be eliminated. Specifically, we can reserve and set a bit in the code pointer so that the resulting address always falls within the region of secure memory. When branching to this specially encoded pointer, a *SecureFault* exception will happen. It automatically traps into the secure runtime and we can perform the decoding in the handler. Unfortunately, in our experiments with this idea, the performance overhead becomes much higher than the current design. This is because exception handling involves more costly context switching operations, whereas our optimized implementation in assembly avoids saving and restoring the processor context. Essentially, we sacrifice storage for performance.

Rewriting Direct Calls as Indirect Calls: If the current design, after a randomization is performed, the secure runtime directly adjusts the offset in the direct call instructions at run-time. This can be avoided if we replace direct calls with indirect calls and use the function dispatch table to find the real target. However, since direct calls are frequently used in programs, the added overhead of trapping to the secure runtime actually outweighs the time required to adjust the direct call offsets. This has been verified in our experiments.

PIC: The *position-independent code* (PIC) [80] allows code to be placed at any locations using PC-relative addressing. Therefore, it is widely adopted in shared libraries and to enable ASLR in commodity OSs. While PIC makes randomization easier, it needs the library vendor to recompile the code. More importantly, it is unsuitable for MCUs for the following reasons. First, PIC introduces more overhead because references to global objects are redirected via the *Global Offset Table* (GOT). This requires additional redirection instructions (PLT) as well as more memory to store the GOT. That is why PIC is rarely used in MCU firmware. Second, direct calls within a library are not redirected in PIC. This means doing function-level randomization will still break the direct call references.

Appendix C. Special Cases of Binary Rewriting

During development, we encountered some special cases that need special treatment. Two of them are notable: *zero-sized symbols* and *fall-through symbols*. These special cases mainly occur in libc (e.g., `newlibc`) or compiler run-time (e.g., `libgcc`). For the zero-sized symbols, we infer its size by calculating the offset from itself to the following symbol. By fall-through symbols, we mean a function does not end with a return instruction but falls through the range of the following function. Doing a randomization would break their connection. To solve this problem, we insert a branch instruction `B` at the end of the fall-through function to explicitly jump to the intended instruction – the start of the following function.

Appendix D. Transparent Support of MCU OS

MCU OSs, such as FreeRTOS, typically leverage the hardware-generated exception frame as part of the task context. When a new task is created, the kernel handcrafts an exception frame in which the `PC` slot is assigned with the pointer to the task entry. Note that this entry has been encoded since it is a normal function pointer. When the new task is scheduled to run, the kernel executes an exception return by branching to `EXC_RETURN`. We have replaced this instruction to call `secure_return_veneer`. It decodes the code pointer on the exception stack and then invokes the real exception return. This leads to the new task beginning execution.

Task switching is triggered by the `PendSV` interrupt. Since `HARM` properly handles interrupts by encoding the exception return addresses at entries and decoding them at exits, task switching works smoothly with rerandomization. Note that `HARM` does not incorporate any OS-specific instrumentation. By this OS-agnostic design, `HARM` is compatible with any OS in theory. We have verified this claim by testing it with FreeRTOS in our evaluation.