Randezvous: Making Randomization Effective on MCUs

Anonymous Author(s)

ABSTRACT

Internet-of-Things devices such as autonomous vehicular sensors, medical devices, and industrial cyber-physical systems commonly rely on small, resource-constrained microcontrollers (MCUs). MCU software is typically written in C and is prone to memory safety vulnerabilities that are exploitable by remote attackers to launch code reuse attacks and code/control data disclosure attacks.

We present *Randezvous*, a software diversification-based system that achieves a highly performant mitigation to such attacks and their brute force variants on ARM MCUs. Atop code/data layout randomization and an efficient execute-only code approach, Randezvous creates *decoy pointers* to camouflage control data in memory; code pointers residing in the stack are then protected by a *diversified shadow stack*, *local-to-global variable promotion*, and *return address nullification*. Furthermore, Randezvous adds a novel *delayed reboot* mechanism to slow down brute force attacks and mitigates control data spraying attacks via *global guards*. We demonstrate Randezvous's security by statistically modeling memory-disclosure-equipped brute force attacks under Randezvous, crafting a proof-of-concept exploit that shows Randezvous's efficacy, and studying a real-world CVE. Our evaluation of Randezvous shows low overhead on three benchmark suites and two applications.

CCS CONCEPTS

• **Security and privacy → Systems security**; **Embedded systems security**.

KEYWORDS

microcontrollers, control data protection, entropy reinforcements, randomization

ACM Reference Format:

Anonymous Author(s). 2022. Randezvous: Making Randomization Effective on MCUs. In *Proceedings of Annual Computer Security Applications Conference (ACSAC '22).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXX.XXXXXXX

1 INTRODUCTION

The increasing prevalence of Internet-of-Things devices continues to drive the need for security in embedded microcontroller (MCU) systems. Software for such systems is commonly written in C and, consequently, suffers from memory safety vulnerabilities. These vulnerabilities can be exploited by attackers to launch control-flow hijacking attacks [15, 19, 20, 31, 40, 71, 83] which corrupt control data (e.g., return addresses and function pointers) so that control flow is diverted to existing code within the program. Worse yet, application software on such systems is typically executed in the processor's privileged mode alongside or without an operating system kernel. Successful exploitation of MCU software means that the attacker controls the *entire* system.

To mitigate control-flow hijacking attacks on MCUs, previous work [5, 43, 60, 84, 88, 91] has explored control-flow integrity (CFI) [1] which protects or checks the integrity of control data used in indirect control-flow transfers. However, CFI is vulnerable to advanced attacks [19, 37] even with a fully precise static control-flow graph (CFG) and a protected shadow stack. Also, CFI implementations on MCUs incur high runtime overhead (8.1%–513% [5, 60, 91]), leaving them less likely to be deployed in practice.[1]

Randomization [65] with execute-only memory (XOM) [47, 75] is another potential solution: by randomizing the location of code and preventing buffer overreads [82] from reading the code segment, attackers no longer know where reusable code is located and therefore cannot divert control flow to the chosen code. However, such approaches have two key limitations on MCUs. First, the address space on MCUs is limited: there is no virtual memory [7, 8], so the entropy of randomization is limited by the physical memory size (typically on the order of kilobytes to megabytes, with a maximum of 64 MB [59] we have ever seen). Brute force attacks [74] which simply guess the location of reusable code can therefore succeed in short amounts of time. Second, and worse yet, previous solutions [2, 25, 41, 64] do not mitigate control data leakage attacks in which a buffer overread [82] leaks control data to learn the location of reusable code [27, 30, 66, 72]. In fact, a proof-of-concept exploit we built shows that even a large-sized MCU protected with randomization and XOM can be breached in less than an hour with the help of control data disclosure. On the other hand, control data leakage defenses [16, 27, 53, 67] do exist in general-purpose systems, which hide control data using indirection or encryption. However, they still leave control data identifiable and usable by attackers.

This paper presents *Randezvous*, a system that mitigates control-flow hijacking attacks against ARMv7-M and ARMv8-M MCUs that utilize brute force attacks and attacks which leak control data. Built on top of previous work that randomizes code and global data layouts [25] and enforces XOM [75], Randezvous protects control data with a set of novel techniques we developed. At the center of Randezvous is a new concept called a *decoy pointer*, which is a code pointer that points to a random unused trap instruction; by filling unused data memory with decoy pointers, real code pointers are camouflaged and thus protected. Leveraging decoy pointers in global data segments, Randezvous moves return addresses into a *diversified shadow stack* and *promotes local variables containing function pointers* into globals to protect them. To further reduce the danger of return address leakage, Randezvous introduces *return address nullification* which overwrites stale return addresses with

ACSAC '22, December 5–9, 2022, Austin, TX

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] Silhouette [91] incurs 11.2%–12.1% runtime overhead on our board, much higher than the reported 1.3%–3.4%. §8 discusses the difference in more detail.

decoy pointers so that leakage is limited to return addresses of currently executing functions. In addition to control data protection, Randezvous reinforces the limited entropy on MCUs against attacks with *delayed reboot* and *global guards*. Delayed reboot adds an artificial reboot delay to slow down brute force attacks when an attack attempt is detected. Global guards are an adaptation of previous memory guards [26] to mitigate spraying attacks that massively corrupt a memory region in order to guarantee control-flow hijacking. Collectively, Randezvous builds a holistic probabilistic (but measurably strong) defense against control-flow hijacking attacks on MCUs, which is, to the best of our knowledge, the first to mitigate both control data leakage and lack of entropy on MCUs. Compared to randomization with XOM alone, Randezvous significantly enhances the protection of control data in readable memory and improves the entropy against attacks.

We built a prototype of Randezvous for ARMv8-M MCUs at https://anonymous.4open.science/r/Randezvous, building off the LLVM/Clang compiler [48]. We evaluated Randezvous's security by statistically modeling brute force attacks with control data leakage, building a real exploit that demonstrates the necessity of Randezvous's security features, and analyzing how Randezvous can stop exploitations of a real-world CVE. We also evaluated Randezvous's overhead on three benchmark suites and two real-world applications. On average, Randezvous incurred 5.9% performance overhead, 15.4% code size overhead, and 22.0% data size overhead.

To summarize, our contributions are as follows:

- We developed a set of novel control data protection techniques for MCUs that strengthen randomization and XOM, centered on decoy pointers and including a diversified shadow stack, return address nullification, and local-to-global variable promotion.
- We devised delayed reboot, a mechanism that mitigates brute force attacks exploiting the limited entropy on MCUs.
- We designed and implemented Randezvous, a strong holistic software diversity approach to securing MCUs against control-flow hijacking attacks.
- We built the first mathematical model of brute force attacks with control data leakage on MCUs to evaluate the strength of Randezvous's defenses and demonstrated the efficacy of Randezvous with a proof-of-concept exploit and a study on a real-world CVE.
- We evaluated the performance of Randezvous and found that it incurs, on average, 5.9% performance overhead, 15.4% code size overhead, and 22.0% data size overhead on our benchmarks and applications.

2 BACKGROUND

Randezvous targets MCUs of the ARMv7-M [7] and ARMv8-M [8] architectures. These systems have unique features that present challenges not found on general-purpose platforms.

Architecture and Address Space Layout. Software for MCUs is commonly compiled into a single native code executable that contains all application, library, and/or operating system kernel code. The ARMv7-M [7] and ARMv8-M [8] architectures support unprivileged and privileged execution modes with system call, trap, and interrupt mechanisms that switch between the two modes. However, it is common for all code to be executed in the privileged mode to avoid the latency of privilege mode switching.

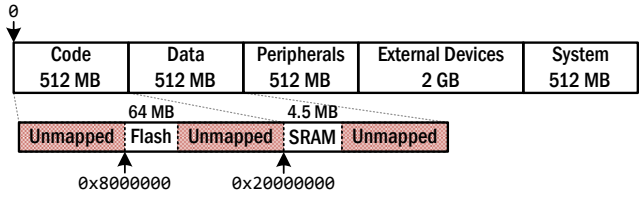

Figure 1: NXP MIMXRT685-EVK (ARMv7/8-M) Address Space (Multiple Memory Mappings Excluded)

Figure 1 depicts the general address space of the two architectures [7, 8], as well as all usable memory mapped on an NXP MIMXRT685-EVK board [58] that we use. Though code and data regions are up to 0.5 GB each, there is only 64 MB flash memory for code and 4.5 MB SRAM for data on our board. The System and Peripheral regions contain memory-mapped CPU and I/O device registers, respectively. As neither architecture supports virtual memory [7, 8], code/data layout randomization techniques are limited to moving code/data within the limited physical address space and have much lower entropy than general-purpose machines with a 64-bit virtual address space and gigabytes of RAM.

Memory Protections. Both architectures optionally support a memory protection unit (MPU) [7, 8], allowing privileged software to configure up to a fixed number of regions across the address space for memory access control. Each region specifies the start address and length of the region as well as the region's read, write, and execute access restrictions for privileged and unprivileged code. To prevent code injection [61], typically code is placed in a non-writable and executable region, and data is placed in writable and non-executable regions. Not all combinations of access permissions are available: neither architecture natively supports XOM [7, 8]. However, efficient software-based XOM solutions [47, 75] exist.

3 THREAT MODEL

We assume a benign but potentially buggy MCU application with memory safety vulnerabilities that allow a remote attacker to write (and optionally read) arbitrary memory locations. We assume the attacker wants to launch a control-flow hijacking attack, such as return-into-libc [74, 83] or return-oriented programming (ROP) [71, 73], against the system. Memory safety attacks that do not corrupt control data (such as non-control data attacks [22]) are out of scope. We further assume that the attacker has a copy of the source code and can generate native code of the same instruction set as used on the system (though layouts may be different due to randomization). The attacker can therefore locate exploitable vulnerabilities and find reusable code in the program's code segment for the aforementioned attacks.

As Randezvous uses randomization to thwart code reuse attacks, it faces several threats that may undermine its defenses:

Threat 1. *An attacker may attempt to use a buffer overread [82] to read the code segment and locate reusable code.*

Threat 2. *An attacker may attempt to use a buffer overread [82] to read control data (pointers to code like return addresses and function pointers) out of memory to locate reusable code.*

Threat 3. *An attacker may attempt to guess the location of reusable code or the location of a control data slot (a memory location containing control data) in a brute force attack.*

Threat 4. *An attacker may corrupt a control data slot to hijack the control flow.*

Threat 5. *An attacker may "spray" control data across a memory region [78] to corrupt all control data slots within that region.*

4 DESIGN

Randezvous is a compiler that transforms code installed on an MCU and a set of runtime support routines used by the MCU's boot code and exception handlers. Our design requires that the target MCU is an ARMv7/8-M system with MPU support, the set of debug registers needed by PicoXOM [75], and a hardware-based cryptographically secure pseudorandom number generator (CSPRNG). These hardware features are available on many real-world MCUs, from low-end (e.g., STM32L412R8 [79]) to high-end (e.g., MIMXRT685-EVK [58]) and across manufacturers (e.g., STMicroelectronics [81], Microchip [56], and Renesas [70]).

In principle, Randezvous protects control data by destroying it when possible and hiding it with improved entropy when destruction is infeasible. We break down Randezvous's design components into three categories: ① randomization and code protection, ② control data protection, and ③ entropy reinforcements. We first describe the randomization and code protection techniques that Randezvous employs, which were explored by previous work [25, 75]. We then explain how Randezvous's control data protection and entropy reinforcement techniques mitigate the additional threats described in §3.

4.1 Randomization and Code Protection

Traditional code reuse attacks [71, 83] require the attacker to know a priori the location of reusable code in memory. Randezvous therefore utilizes randomization and XOM to force the attacker to either use a buffer overread to leak control data [27, 30, 66, 72] or use brute force attacks that guess the location of reusable code.

Specifically, Randezvous performs the following randomized permutations of code at compile time: ① *Function layout reordering*: Randezvous places each function in the program at a random location within the code segment. ② *Basic block layout reordering*: Within each function, Randezvous shuffles the order of basic blocks. If a basic block can fall through to a successor, they are kept contiguous in memory to avoid adding extra branch instructions to the code. ③ *Trap instruction insertion*: Randezvous fills *unused* code segment memory (between functions and between basic blocks that do not fall through) with trap instructions. These instructions are never executed during benign executions and only detect attack probes that jump to unused code. When that happens, Randezvous's trap handler responds by rebooting the system and optionally alerting a system administrator that a potential attack attempt has been thwarted. Randezvous also randomizes the layout of global data segments (i.e., `.rodata`, `.data`, and `.bss`) at compile time by placing each memory object at a random location within its segment. The reason to use compile-time randomization rather than runtime rerandomization is that, compared to the former, the latter requires significantly more resources on MCUs (e.g., separate memory for storing the original program to be randomized) while only adding one extra bit of entropy against brute force attacks [74]. Though it is an orthogonal issue to those this paper addresses, §A discusses how binaries diversified at compile time can be deployed in practice for interested readers.

To mitigate Threat 1, Randezvous employs XOM on the code segment. As the ARMv7/8-M MPU does not support XOM [7, 8], Randezvous employs a software alternative named PicoXOM. PicoXOM [75] configures the ARM debug registers, called Data Watchpoint and Trace (DWT) comparators [7, 8], to generate a trap if a read is performed from the code segment. Furthermore, since the debug registers are memory-mapped [7, 8] i.e., they are reconfigurable by writing to special memory locations, PicoXOM uses additional DWT comparators to ensure that XOM cannot be disabled by reconfiguring the debug registers.

4.2 Control Data Protection

Randomization plus XOM defeats Threat 1. However, an attacker can attack the system by leaking control data (Threat 2) or by guessing the location of code (Threat 3) and then corrupting control data in memory (Threats 4 and 5). We now describe how Randezvous protects the confidentiality and integrity of control data.

4.2.1 Decoy Pointers. To mitigate Threats 2 and 4, we developed *decoy pointers*, which are code pointers that point to random trap instructions and are used to fill unused data memory. Unlike other techniques used in code/data layout randomization, decoy pointers are novel as they, when combined with randomization and XOM, can camouflage genuine control data (slots): attackers leaking data via a buffer overread [82] cannot distinguish actual control data from decoy pointers; neither can they distinguish control data slots from unused data memory. Even if leaked, decoy pointers are lethal and using them in control-flow hijacking risks trapping the system.

By default, Randezvous only fills unused memory in the global data segments with decoy pointers. §4.2.2 and §4.2.3 explain how Randezvous protects control data on the stack by moving it to the global data segments. As many MCU heap implementations simply manage a statically allocated chunk of memory in a global data segment as the heap, such a heap as a whole benefits from decoy pointers placed around it. A more overhead-tolerant implementation could camouflage in-heap control data by providing a custom `free` that refills freed memory with decoy pointers.

4.2.2 Diversified Shadow Stack. Return addresses on the stack pose two challenges under our threat model. First, return addresses are the most common target to corrupt in code reuse attacks (Threat 4). Second, return addresses are relatively easy to leak (Threat 2) via buffer overreads [82]. Randezvous must protect return addresses to mitigate these threats.

Randezvous protects return addresses by using a *diversified shadow stack*, which is a compact shadow stack [17] with random per-function strides. Randezvous employs four methods of randomizing the shadow stack. First, Randezvous places the shadow stack within the `.data` segment so that its location is randomized at compile time. Second, Randezvous initializes the shadow stack with decoy pointers, camouflaging real return addresses. Third, Randezvous selects a static random stride value for each function

at compile time. Fourth, Randezvous selects a dynamic global random stride value at boot time from the CSPRNG. For each non-leaf function, both the static and dynamic stride values are added to a shadow stack pointer in its prologue to determine the location for saving the return address for the next function call. Likewise, the stride values are subtracted from the shadow stack pointer in its epilogue before loading its own return address from the shadow stack. Since the dynamic stride value is selected at boot time, the memory locations to which return addresses are stored get rerandomized for each reboot. Randezvous further encodes the static stride values in code and keeps the shadow stack pointer and dynamic stride value in reserved registers to prevent leakage and corruption.

4.2.3 Local-to-Global Variable Promotion. Local variables that hold function pointers are susceptible to leakage (Threat 2) or corruption (Threat 4) as they are stored on the regular stack. An attacker can use a buffer overflow to corrupt them with addresses of reusable code. The attacker can also use a buffer overread [82] to leak them and use them to learn the location of reusable code. To mitigate such threats, we developed a simple *local-to-global variable promotion* transformation in the Randezvous compiler. This transformation converts local variables that may contain function pointers into global variables, enabling global data layout randomization to randomize their locations and decoy pointers to camouflage them.

The transformation is safe so long as the function containing the promoted variable is not called recursively (i.e., the function is not part of a strongly connected component of two or more nodes in the call graph). To support local function pointers in recursive functions, Randezvous promotes each of such function pointers to an array and requires developers to specify a maximum recursion depth as the array length. Randezvous then instruments the function to use one copy of the function pointer for each recursion.

4.2.4 Return Address Nullification. Our diversified shadow stack in §4.2.2 mitigates return address leakage. However, a buffer overread [82] may still allow an attacker to leak large amounts of the shadow stack at a time. To further reduce the danger of such leakage, we developed a new compiler transformation called *return address nullification* which overwrites the stale return address on the shadow stack with a null value before a function returns. This transformation ensures that a single buffer overread can only leak the return addresses of actively executing functions, limiting the number of return addresses a particular buffer overread can disclose.

When nullifying a return address, instead of zeroing it out, Randezvous overwrites it with a distinct decoy pointer statically chosen and encoded in code for each nullification site. In this way, Randezvous ensures that memory used for return addresses always appears to be decoy pointers, sustaining its initial state.

4.3 Entropy Reinforcements

Despite Randezvous's randomization and control data protection schemes, the entropy they provide on MCUs with small memory size may not effectively resist brute force and control data spraying attacks (as §6 will discuss). This section discusses how Randezvous reinforces the limited entropy on MCUs to mitigate such attacks.

4.3.1 Delayed Reboot. Randezvous's code reuse defenses are probabilistic: each time an attacker tries to attack the system by guessing where reusable code is located or by guessing which chunk of memory contains control data, there is a small chance that the attacker will guess correctly. Consequently, if the attacker repeatedly tries different values and has no bound on the number of attack attempts (Threat 3), there is an amount of time by which the attacker is expected to guess correctly and succeed. It is then natural to ask how long a system is expected to resist such brute force attacks. If the time is sufficiently long, then probabilistic defenses suffice.

Our security analysis in §6 models such attacks and computes the time by which we expect an attacker to succeed. Our analysis shows that the entropy provided by the aforementioned Randezvous defenses alone may not effectively resist such brute force attacks for a reasonable length of time for all MCUs; MCUs with less memory simply have too few places in which to hide code and/or camouflage control data.

As the number of possible locations of a single piece of reusable code or control data is too small, the only other recourse is to make each failed attack attempt take longer. Hence, we developed a technique called *delayed reboot* which artificially delays a system's reboot time. Whenever it detects a trap caused by a violation of Randezvous's security policies, Randezvous reboots the system. Successive reboots are incrementally slowed, artificially reducing the number of failed attempts an attacker can feasibly perpetrate in a given amount of time. For the i-th successive system reboot caused by a violation, an artificial delay in time D_i is added to the boot sequence. D_i monotonically increases as i increases until the number of such reboots reaches a predetermined value R, after which D_i remains constant.

Delayed reboot exchanges availability for confidentiality and integrity through configuration of the parameter R and the delay function incrementing D_i: a smaller value of R and larger values of D_i provide more integrity and confidentiality at the expense of availability. Our security evaluation in §6 quantifies the security gain and availability loss of using delayed reboot, and we use our analysis results to inform concrete configurations to meet specific system requirements. However, we note that delayed reboot may not be appropriate for systems with hard real-time requirements or that cannot tolerate service disruptions. For example, a car's engine control unit may not be suitable for delayed reboot due to real-time constraints while, in contrast, a network of monitoring sensor devices can tolerate delayed reboot, especially if multiple devices monitor overlapping areas to provide redundancy. Systems that cannot use delayed reboot will need to use more memory to gain the entropy needed to stay secure.

4.3.2 Global Guards. Randomizing global data and camouflaging control data with decoy pointers together hinder an attacker from corrupting control data in the global data segments. However, the attacker can still corrupt control data in these regions via spraying attacks (Threat 5). If corrupting non-control data does not crash the program, a buffer overflow that writes to the whole `.data` segment is *guaranteed* to corrupt control data in the `.data` segment, neutralizing the already limited entropy of randomization.

To mitigate this threat, we repurposed guard memory [26], which was originally meant to detect stack smashing only. At boot time, Randezvous uses the CSPRNG to randomly select one or more randomly sized pieces of unused data memory as *global guards* and

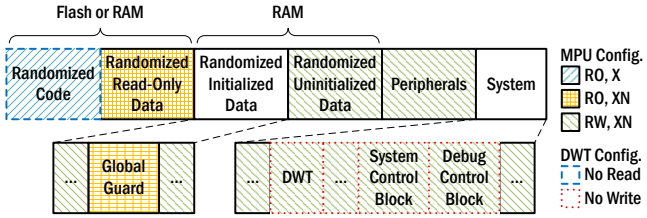

Figure 2: Memory Protection of Randezvous Prototype

configures the MPU to disallow writes to them (the specific number depends on how many regions the MPU can support). Attempted writes to them cause a trap and trigger a reboot.

Global guards establish the entropy against spraying attacks; successful attacks must avoid writing to any global guards, which becomes much less probable. Since global guards are randomly selected at boot time, their location information from previous failed attack attempts cannot be used to inform future attacks.

5 IMPLEMENTATION

We built a prototype of Randezvous for the ARMv8-M architecture with all of our design components except delayed reboot; we opted not to implement it as it does not impact our evaluation. We added four compiler passes to LLVM/Clang 11.0.1's ARM code generator [48], totaling 4,336 source lines of code using Tokei 12.1.2 [90]. We now describe our prototype's memory layout and then describe implementation details of our compiler passes.

PicoXOM Enhancements and Memory Configuration. PicoXOM, Randezvous's XOM component, was only implemented for ARMv7-M [75]. We extended its MPU and DWT configuration code to support ARMv8-M. Unlike the previous implementation on ARMv7-M, which can only protect code segments up to 128 KB, we have verified that PicoXOM on ARMv8-M can support an arbitrary code size. This allows our prototype to support larger code bases. It also increases the entropy for code layout randomization if the usable memory for code is larger than 128 KB. Our extended PicoXOM implementation contains 361 source lines of C code.

Figure 2 shows our Randezvous prototype's memory protection configuration. It requires five MPU regions to cover code, read-only data, RAM, a single global guard, and peripherals. Note that the System region requires no separate MPU region because it is always readable, writable, and execute-never for privileged code regardless of the MPU configuration [8]. ARMv8-M DWT comparators must be used in pairs to monitor memory address ranges [8], so our prototype uses four DWT comparators (two pairs) to read-protect the code segment and write-protect critical memory-mapped system registers.

Code Layout Randomization. The code layout randomization pass, as §4.1 describes, randomizes the code layout by shuffling the order of functions and basic blocks and inserting trap instructions between them. It takes a size option for developers to specify the maximum code size and a seed option to be able to generate different code layouts. We used LLVM's `RandomNumberGenerator` [51] to make our experimental results more reproducible.

```
push {r4,lr}
...
pop  {r4,pc}
```
(a) Original Code

```
str  lr,[r8],#32
add  r8,r8,r9
push {r4}
...
pop  {r4}
sub  r8,r8,r9
ldr  pc,[r8,#32]!
```
(b) Diversified Shadow Stack w/ Static Stride of 32

```
str  lr,[r8],#32
add  r8,r8,r9
push {r4}
...
pop  {r4}
sub  r8,r8,r9
ldr  lr,[r8,#32]!
movw ip,#decoy-lo16
movt ip,#decoy-hi16
str  ip,[r8]
bx   lr
```
(c) Return Address Nullification

Figure 3: Example of Prologue/Epilogue Transformations

Global Data Layout Randomization. The global data layout randomization pass randomizes the layout of global data segments by shuffling the order of global variables and inserting an unused memory object (called a garbage object) of random size between each two of them. Similarly, it takes three size options for the maximum size of the three global data segments and also comes with a seed option. We opted to implement decoy pointers and global guards in this pass as well, as they reside in the global data segments. The implementation of the former is by initializing garbage objects in the `.rodata` and `.data` segments with addresses of random trap instructions. For the latter, we opted to implement support for a single global guard by randomly picking a developer-specified number of garbage objects in the `.data` segment and encoding their addresses and sizes in a runtime function. This function randomly picks a garbage object from those encoded ones as the global guard and returns its address and size; the source of randomness used in the function is the CSPRNG whose address is specified by developers. Randezvous's MPU configuration code calls this function and sets up a read-only MPU region for the global guard.

Diversified Shadow Stack. The diversified shadow stack pass transforms function prologues and epilogues so that they access the shadow stack for their return addresses. The pass also takes a seed option. The shadow stack itself is created as a global variable whose size can be specified by developers. Its location in the `.data` segment is randomized by the global data layout randomization pass. To improve performance and avoid leakage of the shadow stack address and stride, we reserve two callee-saved registers (`r8` and `r9`) to hold the shadow stack pointer and stride value, respectively. Our pass generates a runtime function to initialize the two registers. This function sets `r8` to point to the shadow stack, loads a random number to `r9` from the CSPRNG, and clears a developer-specified number of high bits in `r9` to limit the stride length. The system's boot code calls this function before making any other function calls. During transformation, our pass generates a random static stride for each function, whose length is also limited by the same number of bits. As the pass already finds and transforms instructions in function epilogues, we opted to implement return address nullification in the pass as well. For each instrumented function epilogue, our pass randomly picks a trap instruction and generates code that writes its address back to the shadow stack. Figure 3 demonstrates the diversified shadow stack and return address nullification transformations performed on a function.

Local-to-Global Variable Promotion. The local-to-global variable promotion pass promotes local variables whose type contains function pointer types to globals, as §4.2.3 describes. The pass operates on LLVM's intermediate representation (IR) bitcode before it is lowered to machine code. As none of our benchmarks and applications uses local function pointers in recursive functions, we elided implementing support for it.

6 SECURITY EVALUATION

We now evaluate Randezvous's security by measuring the entropy it adds to three different-sized MCUs and computing the amount of reboot delay needed to protect these systems from attacks for a given amount of time. We then provide a proof-of-concept exploit that experimentally demonstrates the security of Randezvous and a study on how Randezvous could mitigate attacks exploiting a real-world CVE. Table 8 in §B lists all mathematical symbols used in this section for quick reference. §C details the derivation of each numbered equation in this section for interested readers.

6.1 Attack Procedure

We model a return-into-libc [74, 83] control-flow hijacking attack as it is the simplest. Other types of attacks (e.g., ROP [71, 73] and JIT-ROP [77]) require locating additional reusable code and therefore require leaking or guessing more code locations. Consequently, if Randezvous can resist return-into-libc attacks, it should be able to resist these more sophisticated attacks as well. In the return-into-libc attack, the attacker follows the two steps below:

(1) Locate the control flow target to which to jump. For a return-into-libc attack, this is the address of a function.
(2) Find a control data slot and corrupt it with the address of the control flow target acquired in Step 1. This is usually where a return address or a function pointer is stored, which will be used in a future control flow transfer.

On an unprotected system, Step 1 can be skipped because the attacker has a priori knowledge of the code layout. However, as Randezvous randomizes the code and data layouts and forbids code reads via PicoXOM [75] (§4.1), the attacker is forced to

1a) guess the location of the control flow target, or
1b) try leaking a return address using a buffer overread, or
1c) try leaking a function pointer (if any) using a buffer overread.

Similarly, in Step 2, finding the address of a control data slot is no longer straightforward for the attacker; in Randezvous, control data is stored in the `.data` segment (§4.2.2 and §4.2.3), randomized to unknown locations (§4.1), camouflaged among numerous decoy pointers (§4.2.1), and protected by randomly-picked non-writable global guards (§4.3.2). As a result, the attacker must either

2a) guess the location of a control data slot to corrupt, or
2b) massively corrupt part of the `.data` segment, aiming to corrupt a desired control data slot while hitting none of the global guards, i.e., a control data spraying attack [78].

6.2 Attack Probe Analysis

Our analysis assumes that the attacker knows the boundaries of randomized memory regions and makes no out-of-bounds guesses in Steps 1 and 2. While not always true in practice, this assumption biases the analysis in the attacker's favor and simplifies our analysis.

We first analyze the expected number of attempts the attacker needs for each strategy to complete Step 1. For Strategy 1a, let S_C be the size of the randomized code segment, S_{C_O} be the size of the original application code, and S_T be the size of the control flow target. Assuming the control flow target is 2-byte aligned (typical for Thumb instructions [7, 8]) and has an equal chance to appear in each eligible location, then the probability of success is $p_{S,1a} = \frac{2}{S_C - S_T + 2}$; the chance of finding a trap instruction can be approximated as $p_{T,1a} = \frac{S_C - S_{C_O}}{S_C} \cdot \frac{S_C - S_T}{S_C}$. For a brute force attack, an attacker can simply retry the attack repeatedly with different values for the control flow target until the attack works, excluding previously guessed values each time a new guess is made. If P_x is a random variable representing the number of guesses for a success in Strategy x ($x \in \{1a, 1b, 1c, 2a, 2b\}$), then the expected number of guesses for completing Step 1 with Strategy 1a is

$$E(P_{1a}) = \frac{S_C - S_T + 4}{4}. \tag{1}$$

For Strategies 1b and 1c, our analysis assumes the attacker's best case scenario: a function pointer or return address pointing into the desired function exists in a single memory location; if leaked, the attacker can locate the function. In this scenario, the attacker first uses a buffer overread [82] to leak the entire contents of the `.data` segment and then examines it for the desired control data. Even so, the attacker can at most eliminate values that do not look like control data and still must guess which of the remaining ones can be used. This is because Randezvous randomizes the `.data` segment layout and camouflages control data with decoy pointers. Let S_D be the size of the randomized `.data` segment, $S_{D'}$ be the size of memory in the `.data` segment that does not look like control data, and N be the number of control data slots in the `.data` segment. N can be approximated by the current call chain depth (due to return address nullification in §4.2.4) plus the number of function pointers in the program. Assuming the desired control data has an equal chance to appear in each possible location, the attacker must try every memory location that appears to contain control data. The probability of success is $p_{S,1b} = p_{S,1c} = \frac{4}{S_D - S_{D'}}$, and the probability of finding a decoy pointer is approximately $p_{T,1b} = p_{T,1c} = \frac{S_D - S_{D'} - 4N}{S_D - S_{D'}}$. The difference between Strategies 1b and 1c is whether the attacker can exclude a previously incorrect guess: return addresses might be stored in different memory locations as the dynamic shadow stack stride is randomized on each boot, while function pointers always reside in the same address across reboots as the `.data` segment is randomized once at compile time. This leads to a difference in the expected number of guesses, shown in Equations 2 and 3, respectively:

$$E(P_{1b}) = \frac{S_D - S_{D'}}{4} \tag{2}$$

$$E(P_{1c}) = \frac{S_D - S_{D'} + 4}{8} \tag{3}$$

We now consider the expected number of attempts needed to complete Step 2. For Strategy 2a, the attacker can also leverage a buffer overread [82] on the `.data` segment to filter out memory that does not resemble control data except for zeroed memory, which might be uninitialized control data slots. Let S_{D_0} be the size of zeroed memory in the `.data` segment and S_G be the total size of all

Table 1: Common Values for Time Analysis

	Small	Medium	Large		All Systems
S_C	32 KB	1 MB	16 MB	S_G	32 bytes
S_{C_O}	16 KB	128 KB	1 MB	S_T	16 bytes
S_D	32 KB	256 KB	4 MB	S_W	128 bytes
$S_{D'}$	1 KB	4 KB	32 KB	t_B	1 second
S_{D_0}	128 bytes	512 bytes	1 KB	t_N	0.6 seconds
N	8	32	64	T_{min}	3 days

Table 2: Time Analysis Results (Best/Worst for the Attacker)

System	Case	Strategies	$E(P)$	p_T	T_n
Small	Worst	{1a, 2a}	8,155,248.0	0.151%	56.8 days
Small	Best	{1c, 2b}	132,651.0	6.073%	1.0 days
Medium	Worst	{1a, 2a}	529,522,800.0	0.557%	10.1 years
Medium	Best	{1c, 2b}	2,087,482.7	1.934%	15.0 days
Large	Worst	{1a, 2a}	68,199,318,000.0	0.007%	1,297.7 years
Large	Best	{1c, 2b}	266,649,737.1	0.220%	5.1 years

global guards. The chance of success is $p_{S,2a} = \frac{4N}{S_D - S_{D'} + S_{D_0}}$, and the chance of hitting any of the global guards is $p_{T,2a} = \frac{S_G}{S_D - S_{D'} + S_{D_0}}$. Similar to Strategy 1b, the attacker cannot exclude previously incorrect guesses due to both the dynamic shadow stack stride and the global guards, so the expected number of guesses is

$$E(P_{2a}) = \frac{S_D - S_{D'}}{4N}. \tag{4}$$

For Strategy 2b, let S_W be the size of memory in the `.data` segment that the attacker chooses to corrupt. Equations 5 and 6 give the probability of success and of hitting a global guard, respectively. Equation 7 computes the expected number of attempts:

$$p_{S,2b} = \frac{\sum_{i=1}^{\min(N, \frac{S_W}{4})} C(N,i) C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4} - i)}{C(\frac{S_D}{4}, \frac{S_W}{4})} \tag{5}$$

$$p_{T,2b} = 1 - p_{S,2b} - \frac{C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4})}{C(\frac{S_D}{4}, \frac{S_W}{4})} \tag{6}$$

$$E(P_{2b}) = \frac{1}{p_{S,2b}} \tag{7}$$

Combining the two steps, there are three outcomes: ① success, only when an attacker makes a correct guess in both steps; ② nothing happening, due to an incorrect guess in Step 2 that hits none of the global guards; ③ trap, which can be caused by an incorrect guess in either Step 1 (finding a decoy pointer) or Step 2 (hitting any of the global guards). As none of the two unsuccessful outcomes gives information about which control data (slot) is (in)correct, the attacker can only guess blindly in both steps. Let P be a random variable of the number of brute force attacks for a success. Since the two steps are independent of each other, we have the chance of success $p_S = p_{S,x} \cdot p_{S,y}$, the chance of trapping the system $p_T = p_{T,x} \cdot p_{S,y} + p_{T,y}$, and *the expected number of brute force attacks for a success* $E(P) = E(P_x) \cdot E(P_y)$ if the attacker adopts Strategies x and y ($x \in \{1a, 1b, 1c\}$ and $y \in \{2a, 2b\}$).

6.3 Time Analysis

Entropy measures a system's randomness, but it fails to measure the system's strength against brute force attacks as it fails to consider the frequency at which attacks are launched. We therefore analyze how long a Randezvous-protected system, with different sizes, can resist brute force attacks. This analysis informs the configuration of delayed reboot and thus controls the security/availability trade-off.

Let t_B be the time from system booting to reaching a vulnerability that an attacker can exploit and t_N be the time for the attacker to send an attack payload and receive its execution result over the network. Without Randezvous's delayed reboot, we can expect the system to withstand brute force attacks by an amount of time $T_n = (p_T \cdot t_B + t_N) \cdot E(P)$. With delayed reboot providing a total delay of T_d, we wish the whole system to resist brute force attacks for at least an amount of time T_{min} before the attacker finishes the expected number of attack payloads to succeed. So we have $T_d \leq \sum_{i=1}^{R} D_i$, $T_n + T_d = T_{min}$, and $R \leq p_T \cdot E(P)$, where R is the number of reboots after which the delay stops increasing and $\{D_i\}_{i=1}^{R}$ is the sequence of the delay Randezvous adds to the i-th reboot, as §4.3.1 describes.

We aim to protect the system from brute force attacks for three or more days. Three days give the system time to alert an administrator about the attack and for the administrator to respond, even if the attack commences during a short period in which the administrator is unavailable (e.g., a weekend). Table 1 lists three sets of common values representing three MCUs of different sizes (STM32L412R8 [79], STM32F469NIH6 [80], and MIMXRT685-EVK [58, 59]) and values we pick to evaluate attacks (latencies are based on a wireless network [76]) and Randezvous's protections. By substituting all variables with their corresponding values in each set, we can estimate whether delayed reboot is needed (i.e., whether $T_n < T_{min}$) and, if so, how much delay can be scattered throughout all R reboots. Our results, summarized in Table 2, show that the medium- and large-sized systems do not need delayed reboot; Randezvous's other protections can mitigate all the modeled attacks for at least half a month. The small system, however, requires an average per-reboot delay of 21.3 seconds to keep it probabilistically secure for three days against all possible attack strategies that we evaluated.

While our results necessitate delayed reboot for certain systems, we note that using an exponentially-growing delay will still provide reasonable availability when an attack commences while maintaining our target of three days worth of resilience. For example, our best case for the attacker expects the system to trap 8055.3 times; Randezvous could be configured with $\{D_i\}_{i=1}^{R}$ as an exponential sequence with $D_1 = 100$ ms, $R = 8055$, and a ratio of 1.001. Even at the 2000-th reboot (at which point a system administrator should have been notified), the delay on a single boot is just around 738 ms.

6.4 Exploit Analysis

Proof-of-Concept Exploit. We built a proof-of-concept exploit to showcase Randezvous's security. The exploit consists of a script representing an attacker and a vulnerable application that can run on an NXP MIMXRT685-EVK board [58]. The application contains both arbitrary memory read and write vulnerabilities, matching our threat model in §3. To favor the attacker, it also contains a global function pointer pointing to the attacker's desired function. We compiled the application with three different configurations: one unprotected, one protected with only randomization and PicoXOM (as in §4.1), and one protected with full Randezvous. For the two

protected configurations, we further configured the application to match each of the three different-sized MCUs in Table 1 as closely as possible. When running the application, the script communicates with the board via a serial port and sends attack payloads generated from the best strategies for the attacker for each configuration: direct return-into-libc for the unprotected, Strategy 1c with return address corruption for randomization plus PicoXOM, and Strategies 1c and 2b for Randezvous.

Our exploit succeeded immediately for the unprotected system as no guessing is needed for Step 1 or 2. With randomization plus PicoXOM, the exploit succeeded in 15 seconds, 68 seconds, and 2,821 seconds for the small-, medium-, and large-sized systems, respectively. Most of time was spent trying out leaked values that resemble control data. In contrast, the exploit failed in all three Randezvous-protected systems even after continuously sending attack payloads for three days.

Real-World CVE. To demonstrate its efficacy against real-world exploits, we analyzed how Randezvous could stop attacks exploiting CVE-2021-27421 [29]. We picked this CVE because it can both read from and write to arbitrary heap locations, because it affects applications using the NXP MCUXpresso SDK library, and because we can exploit it on our NXP MIMXRT685-EVK board [58].

CVE-2021-27421 [29] overflows a heap buffer. We built a demonstrative application with the CVE for the small-sized system and compiled it with similar configurations to those used in our proof-of-concept exploit. We then launched a return-into-libc attack on our board, which exploits the CVE to corrupt a pointer in the heap to point to a memory location of our choosing which the application subsequently dereferences. Our attack utilizes attack strategies that use no buffer overread and spraying (Strategy 1a with return address corruption for randomization plus PicoXOM and Strategies 1a and 2a for Randezvous) because the application stores no return address or function pointer to the attacker's desired control flow target in memory and because the exploit only corrupts four bytes of memory. Our attack exploited the unprotected system immediately and the system protected by randomization plus PicoXOM in 23.6 hours. In contrast, the attack failed on the Randezvous-protected system after running for 26.5 hours; we therefore expect Randezvous to resist the attack for more than three days for both the small- and larger-sized systems.

7 PERFORMANCE EVALUATION

We evaluated Randezvous's performance on an NXP MIMXRT685-EVK board which has an ARM Cortex-M33 processor implementing the ARMv8-M Mainline architecture that can run up to 300 MHz [58]. It comes with 4.5 MB of SRAM, 64 MB of flash memory, a true random number generator (TRNG) that fulfills Randezvous's CSPRNG requirement, and an SD card slot [58, 59].

We used three benchmark suites and two real-world applications to evaluate Randezvous. **BEEBS** [62] is a collection of small benchmarks to measure energy consumption on embedded systems. It includes a wide range of workloads commonly seen on MCUs such as matrix multiplication, sorting, and hashing. As many BEEBS programs are too small or perform too little computation, we picked 54 of its 80 programs that, for a total of 10,240 iterations, run longer than 0.1 seconds on our board. **CoreMark-Pro** [35] is a processor benchmark suite that includes and enhances CoreMark [34], an industry standard benchmark for embedded processors, with more CPU- and memory-intensive programs. It consists of five integer benchmarks and four floating-point benchmarks that together characterize processor performance. **MbedTLS-Benchmark** [55] is a test program for the Mbed TLS library [6] that benchmarks both the latency and throughput of various cryptographic algorithms (e.g., MD5, SHA, AES, and RSA). **PinLock** [36] is an application that emulates a password-based lock. It reads a 4-digit passphrase from a serial port, computes a SHA-256 hash of the input, and activates an LED if the hash matches the stored passphrase hash. **FatFs-SD** is an application from the board manufacturer; it operates a FAT file system on an SD card with file system creation, mounting, and file I/O. Previous work [5, 25, 75, 76] used PinLock and FatFs-SD.

We compiled each program into an ELF executable and loaded its code into the SRAM for execution, using two configurations: Baseline and Randezvous. In Baseline, we used the LLVM/Clang compiler [48] to compile programs with all Randezvous passes and runtime components disabled. In Randezvous, we enabled everything; all Randezvous's randomization seeds are set to zero, and all memory size options for Randezvous's passes are set appropriately to allow execution in the SRAM while still adding entropy to the program. In particular, the shadow stack size and stride length were tailored to add one bit of entropy. Both configurations use the `-Os` optimization (a common practice in MCU software development) plus the `-fomit-frame-pointer` option and perform link-time optimization (LTO) via `-flto` and `-fuse-ld=lld` options.

7.1 Performance Overhead

To measure Randezvous's performance overhead, we configured each BEEBS benchmark to execute for 10,240 iterations of its workload and print out its execution time in milliseconds. Each benchmark in CoreMark-Pro was configured to execute for a minimal number of iterations that is a power of 10 and yields an execution time of at least 10 seconds. MbedTLS-Benchmark measures latency and throughput with 1,024 iterations and 1 or 3 seconds, respectively. All benchmarks produced identical numbers over multiple runs, yielding zero standard deviations. As PinLock and FatFs-SD access slow peripherals, we ran each of them 10 times and report the average execution time with a standard deviation.

Tables 3, 4, 5, and 6 present Baseline performance in absolute numbers as well as the overhead Randezvous incurs relative to Baseline on CoreMark-Pro, MbedTLS-Benchmark, and the applications, respectively. Due to space, we only summarize the numbers for BEEBS. Overall, Randezvous incurs minor performance overhead of 5.9%: 6.9% in BEEBS (from -1.6% to 26.2%), 7.0% in

Table 3: CoreMark-Pro Execution Time (Lower is Better)

	Baseline (ms)	Randezvous (×)		Baseline (ms)	Randezvous (×)
cjpeg-rose7-...	21,172	1.023	parser-125k	41,700	1.069
core	33,813	1.112	radix2-big-64k	15,363	1.177
linear_alg-...	45,177	1.001	sha-test	17,220	1.046
loops-all-...	73,085	1.010	zip-test	37,097	1.014
nnet_test	183,048	1.195			
Geomean (×)					1.070

Table 4: MbedTLS-Benchmark Throughput (Higher is Better)

	Baseline	Randezvous (×)		Baseline	Randezvous (×)		Baseline	Randezvous (×)
MD5 (KB/s)	14,630.25	0.981	AES-CCM-192 (KB/s)	3,198.46	0.924	ECDSA-secp384r1 (sign/s)	13.05	0.968
SHA-1 (KB/s)	56,491.74	0.958	AES-CCM-256 (KB/s)	3,085.67	0.934	ECDSA-secp256r1 (sign/s)	28.39	0.960
SHA-256 (KB/s)	58,746.89	0.956	CTR_DRBG (NOPR) (KB/s)	8,699.62	0.913	ECDSA-secp521r1 (verify/s)	5.66	1.011
SHA-512 (KB/s)	2,216.21	0.978	CTR_DRBG (PR) (KB/s)	5,092.05	0.932	ECDSA-secp384r1 (verify/s)	12.21	0.965
3DES (KB/s)	816.31	0.874	HMAC.. SHA-1 (NOPR) (KB/s)	1,339.78	0.962	ECDSA-secp256r1 (verify/s)	27.09	0.957
DES (KB/s)	2,067.83	0.873	HMAC.. SHA-1 (PR) (KB/s)	1,215.91	0.961	ECDHE-secp521r1 (handshake/s)	4.46	0.991
AES-CBC-128 (KB/s)	61,610.23	0.950	HMAC.. SHA-256 (NOPR) (KB/s)	1,585.92	0.967	ECDHE-secp384r1 (handshake/s)	7.52	0.975
AES-CBC-192 (KB/s)	56,954.36	0.954	HMAC.. SHA-256 (PR) (KB/s)	1,585.94	0.967	ECDHE-secp256r1 (handshake/s)	16.44	0.970
AES-CBC-256 (KB/s)	50,861.98	0.958	RSA-1024 (public/s)	1,420.58	0.987	ECDH-secp521r1 (handshake/s)	8.62	0.991
AES-GCM-128 (KB/s)	2,192.97	0.919	RSA-1024 (private/s)	14.92	0.975	ECDH-secp384r1 (handshake/s)	14.71	0.975
AES-GCM-192 (KB/s)	2,165.85	0.921	DHE-2048 (handshake/s)	0.94	0.979	ECDH-secp256r1 (handshake/s)	32.55	0.971
AES-GCM-256 (KB/s)	2,139.37	0.922	DH-2048 (handshake/s)	1.18	0.975			
AES-CCM-128 (KB/s)	3,319.83	0.919	ECDSA-secp521r1 (sign/s)	7.69	0.986			
Geomean (×)								0.955

Table 5: MbedTLS-Benchmark Latency (Lower is Better)

	Baseline (cycle/byte)	Randezvous (×)		Baseline (cycle/byte)	Randezvous (×)
MD5	15.84	1.009	AES-GCM-256	113.37	1.084
SHA-1	3.47	1.009	AES-CCM-128	72.75	1.087
SHA-256	3.30	1.009	AES-CCM-192	75.54	1.081
SHA-512	109.40	1.022	AES-CCM-256	78.33	1.070
3DES	298.72	1.145	CTR_DRBG (NOPR)	27.22	1.092
DES	117.32	1.145	CTR_DRBG (PR)	47.12	1.071
AES-CBC-128	3.12	1.013	HMAC.. SHA-1 (NOPR)	181.58	1.039
AES-CBC-192	3.44	1.012	HMAC.. SHA-1 (PR)	200.18	1.040
AES-CBC-256	3.95	1.013	HMAC.. SHA-256 (NOPR)	153.25	1.034
AES-GCM-128	110.58	1.087	HMAC.. SHA-256 (PR)	153.25	1.034
AES-GCM-192	111.97	1.085			
Geomean (×)					1.055

CoreMark-Pro, 4.5% in MbedTLS-Benchmark's throughput, 5.5% in MbedTLS-Benchmark's latency, and 0.6% in the applications.

We studied the overhead by enabling only one of Randezvous's features at a time. We discovered that the diversified shadow stack and return address nullification transformations are the major sources of overhead in BEEBS and CoreMark-Pro. Specifically, the former reserves two registers and adds a few instructions in the prologue and epilogue(s) of every non-leaf function. The latter adds a few more instructions in those function epilogues. As a result, Randezvous incurred more overhead on benchmarks with higher register pressure and more frequent function calls. MbedTLS-Benchmark's latency overhead on each algorithm roughly matches its throughput overhead. The highest (in `DES` and `3DES`) also comes from these transformations. `ECDSA-secp521r1` saw a miniscule speedup in signature verification, likely caused by caching. Randezvous exhibits negligible runtime overhead in the applications. We believe this is due to I/O dominating the execution time.

Table 6: Application Execution Time (Lower is Better)

	Baseline (ms)	Stdev (ms)	Randezvous (×)	Stdev (×)
PinLock	46,429.5	108.8	1.009	0.001
FatFs-SD	14,965.3	47.6	1.003	0.003
Geomean	—	—	1.006	—

Table 7: CoreMark-Pro Memory Usage (Lower is Better)

	Baseline Code (bytes)	Baseline Data (bytes)	Randezvous Code (×)	Randezvous Data (×)
cjpeg-rose7-...	104,978	51,802	1.135	1.039
core	72,852	8,451	1.151	1.184
linear_alg-...	75,046	8,839	1.148	1.176
loops-all-...	84,440	12,624	1.146	1.125
nnet_test	75,218	48,734	1.147	1.033
parser-125k	80,808	7,266	1.137	3.855
radix2-big-64k	74,196	1,383,731	1.149	1.001
sha-test	77,368	5,887	1.139	1.264
zip-test	91,910	20,347	1.128	1.084
Geomean	—	—	1.142	1.275

7.2 Memory Overhead

Memory usage is critical for MCUs. We therefore measured how much memory Randezvous uses to provide its protections by calculating code and global data segment sizes (without unused memory) before and after its transformations during compilation.

Table 7 and Figure 4 show Randezvous's code and data size overhead on CoreMark-Pro, MbedTLS-Benchmark, and the two applications, respectively. Again, we summarize BEEBS results due to space. Overall, Randezvous incurs moderate overhead on both code and data sizes: a geometric mean of 15.8% on code size (from 13.3% to 16.2%) and 21.2% on data size (from 7.9% to 31.8%) in BEEBS, 14.2% and 27.5% in CoreMark-Pro, 10.8% and 11.9% in MbedTLS-Benchmark, and 13.6% and 24.5% in the applications. We note that

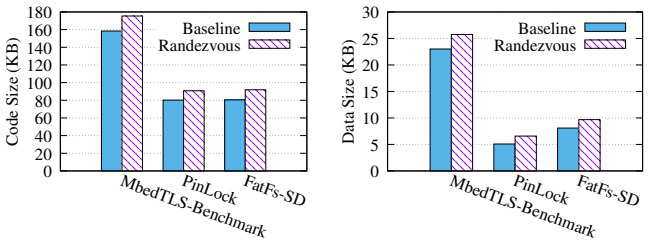

Figure 4: MbedTLS-Benchmark and Application Memory Usage (Lower is Better)

`parser-125k` in CoreMark-Pro exhibits the highest data size overhead because its shadow stack is more than twice the size of its original global data size to accommodate a function that calls itself over 2,000 times. Correspondingly, its stack usage decreases as none of its recursive stack frames contains a return address slot.

Breaking down the overhead, the code size overhead comes from PicoXOM (3.2%–5.5%), function prologue/epilogue transformations (4.6%–7.0%), and runtime components that set up the shadow stack and a single global guard (1,356 bytes). The data size overhead comes from string literals used in additional code (1,389 bytes), a diversified shadow stack (48–19,040 bytes), and promoted local variables containing function pointers (0–5.0%).

8 RELATED WORK

Randomization on General-Purpose Systems. Randomization on general-purpose systems is well studied. The original ASLR [11, 65] loads memory sections at random addresses and is widely deployed. Due to its coarse granularity and lack of entropy on 32-bit systems, researchers have focused on fine-grained code randomization at the level of pages [9], functions [13, 27, 39, 44], basic blocks [45, 87], instructions [32, 42, 63], register allocation [27, 63], execution paths [30], or tunable sizes [68]. Fine-grained data randomization has been explored as well, including global data object reordering [13], data representation encryption [12, 18], structure field randomization [21, 28, 39, 50], stack randomization [3, 13, 23, 49], and heap randomization [10, 57]. While most of these techniques can be used on MCUs, Randezvous leverages just a few of them with the best efficacy and the least performance impact.

Leakage-resistant randomization for general-purpose systems, such as Readactor [27], ASLR-Guard [53], LR^2 [16], and kR^X [67], hide code pointers via indirection or encryption. These systems are still susceptible to control data leakage; despite not knowing *where* code is located, attackers can identify indirect or encrypted code pointers from disclosed memory and reuse them to corrupt control data slots. Randezvous's decoy pointers, in contrast, prevent attackers from identifying real code pointers from decoy pointers; using a leaked pointer risks causing a trap.

Runtime rerandomization shortens the window for successful exploitation and can be done manually at run-time [24], periodically [4, 38, 39, 69, 89], at certain system calls [14, 52, 85], and when detecting suspicious probes [86]. Randezvous uses no runtime rerandomization as its additional resource consumption outweighs its security gain (as §4.1 describes).

Randomization on MCUs. Previous work has employed randomization for MCUs. μArmor [2] and EPOXY [25] employ compile-time code layout randomization; EPOXY [25] also randomizes data layout at compile time. AVRAND [64] and MAVR [41] proposed boot-time code layout randomization for AVR MCUs. Both solutions randomize code and reprogram the flash memory at every reboot, using a trusted bootloader reading metadata from EEPROM or a separate processor with extra flash memory. Compared to Randezvous, all of the above systems assume a weaker threat model and, therefore, do not mitigate information leakage. Consequently, attackers can still locate code and launch code reuse attacks on these systems using information leaked from code [77] or data [27, 30, 66, 72]. It is also unclear if these systems can resist brute force attacks effectively; they omitted modeling such attacks [2, 25, 64] or yielded an outrageously large number of guesses by incorrectly assuming that attackers have to guess the locations of all functions in the program before launching an attack [41]. HARM [76] implements function-level periodical code rerandomization using TrustZone-M on ARMv8-M [8], requiring more than twice the memory. fASLR [54] uses TrustZone-M to dynamically load functions to random addresses in RAM when being called and unload finished ones when out of RAM, thus reducing memory usage of rerandomization. Unlike HARM and fASLR, Randezvous requires no TrustZone-M and thus supports ARMv7-M systems. While runtime rerandomization reduces the window of code reuse attacks, a successful exploit equipping memory disclosure to learn the code layout is still possible, especially where rerandomization may not take place frequently (e.g., fASLR [54]).

As to performance, AVRAND [64] and MAVR [41] only present startup overhead in absolute numbers; comparing to Randezvous is impossible. EPOXY shows better performance in BEEBS (1.6% on average) than Randezvous because its safe stack [46] improves locality. For BEEBS programs that both Randezvous and HARM [76] evaluate (all 19 programs by HARM), Randezvous outperforms HARM (8.8% vs. 25%, on average). Similarly, in BEEBS programs shared between Randezvous and fASLR [54] (5 programs out of 9 by fASLR), Randezvous is slightly faster (2.3% vs. 3.7%, on average).

CFI on MCUs. An alternative to randomization is to use CFI [1] and/or protected shadow stacks [17]. To protect shadow stacks, CaRE [60] and TZmCFI [43] leverage TrustZone-M [8], RECFISH [84] utilizes privilege mode switching, and Silhouette [91] and Kage [33] utilize ARM's unprivileged store instructions. μRAI [5] encodes return addresses in a reserved register and uses system calls to extend the encoding space. All these solutions enforce return address integrity and use coarse-grained forward-edge CFI [1], while SCFP [88] extends a RISC-V MCU with a stateful instruction encryption scheme for fine-grained CFI. However, even with a fully precise static CFG and a protected shadow stack, CFI is still vulnerable to advanced forward-edge corruptions that adhere to the CFG [19, 37]. In contrast, Randezvous provides probabilistic guarantees but is not susceptible to such attacks without identifying both a control flow target and a control data slot.

Performance-wise, Randezvous outperforms all the above CFI implementations except Silhouette and Kage. We believe Silhouette's low overhead (3.4% on BEEBS and 1.3% on CoreMark-Pro) is due to the high latency of the SDRAM used in its evaluation [91]; we evaluated Silhouette on our board (which uses SRAM), and its overhead increases to 12.1% on BEEBS and 11.2% on CoreMark-Pro.

9 CONCLUSIONS

We presented Randezvous: a diversification-based control-flow hijacking defense enhanced with novel techniques that mitigate control data leakage attacks and strengthen the low entropy found on MCUs. Our work demonstrated Randezvous's efficacy and showed that Randezvous incurs low overhead on our benchmarks and applications. We will open-source Randezvous.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information Systems Security* 13, 1, Article 4 (Nov. 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. 2019. Challenges in Designing Exploit Mitigations for Deeply Embedded Systems. In *Proceedings of the 2019 IEEE European Symposium on Security and Privacy (EuroSP '19)*. IEEE Computer Society, Stockholm, Sweden, 31–46. https://doi.org/10.1109/EuroSP.2019.00013

[3] Misiker Tadesse Aga and Todd Austin. 2019. Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '19)*. IEEE Computer Society, Washington, DC, 26–36. https://doi.org/10.1109/CGO.2019.8661202

[4] Salman Ahmed, Ya Xiao, Kevin Z. Snow, Gang Tan, Fabian Monrose, and Danfeng (Daphne) Yao. 2020. Methodologies for Quantifying (Re-)Randomization Security and Timing under JIT-ROP. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM, Orlando, FL, 1803–1820. https://doi.org/10.1145/3372297.3417248

[5] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. 2020. μRAI: Securing Embedded Systems with Return Address Integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS '20)*. Internet Society, San Diego, CA, 18 pages. https://doi.org/10.14722/ndss.2020.24016

[6] Arm Holdings. 2008. *SSL Library Mbed TLS*. https://tls.mbed.org

[7] Arm Holdings 2018. *ARMv7-M Architecture Reference Manual*. Arm Holdings. DDI 0403E.d.

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

[9] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. USENIX Association, San Diego, CA, 433–447.

[10] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, Ottawa, ON, Canada, 158–168. https://doi.org/10.1145/1133981.1134000

[11] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. 2003. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium (Security '03)*. USENIX Association, Washington, DC, 105–120. https://www.usenix.org/conference/12th-usenix-security-symposium/address-obfuscation-efficient-approach-combat-broad-range

[12] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*. Springer-Verlag, Paris, France, 1–22. https://doi.org/10.1007/978-3-540-70542-0_1

[13] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium (Security '05)*. USENIX Association, Baltimore, MD, 255–270. https://www.usenix.org/conference/14th-usenix-security-symposium/efficient-techniques-comprehensive-protection-memory-error

[14] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, 268–279. https://doi.org/10.1145/2810103.2813691

[15] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Berkeley, CA, 227–242. https://doi.org/10.1109/SP.2014.22

[16] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. Internet Society, San Diego, CA, 15 pages. https://doi.org/10.14722/ndss.2016.23364

[17] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP '19)*. IEEE Computer Society, San Francisco, CA, 985–999. https://doi.org/10.1109/SP.2019.00076

[18] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Philippe Martin, and Miguel Castro. 2008. *Data Randomization*. Technical Report MSR-TR-2008-120. Microsoft Research.

[19] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (Security '15)*. USENIX Association, Washington, DC, 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[20] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. USENIX Association, San Diego, CA, 385–399. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

[21] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. 2015. A Practical Approach for Adaptive Data Structure Layout Randomization. In *Proceedings of the 20th European Symposium on Computer Security (ESORICS '15)*. Springer-Verlag, Vienna, Austria, 69–89. https://doi.org/10.1007/978-3-319-24174-6_4

[22] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium (Security '05)*. USENIX Association, Baltimore, MD, 177–191. https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats

[23] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS '15)*. Internet Society, San Diego, CA, 15 pages. https://doi.org/10.14722/ndss.2015.23248

[24] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-Demand Live Randomization. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. ACM, New Orleans, LA, 50–61. https://doi.org/10.1145/2857705.2857726

[25] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S. Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. 2017. Protecting Bare-Metal Embedded Systems with Privilege Overlays. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP '17)*. IEEE Computer Society, San Jose, CA, 289–303. https://doi.org/10.1109/SP.2017.37

[26] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (Security '98)*. USENIX Association, San Antonio, TX, 15 pages. https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention

[27] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, San Jose, CA, 763–780. https://doi.org/10.1109/SP.2015.52

[28] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, 243–255. https://doi.org/10.1145/2810103.2813682

[29] CVE 2021. *CVE-2021-27421*. https://www.cve.org/CVERecord?id=CVE-2021-27421

[30] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS '15)*. Internet Society, San Diego, CA, 15 pages. https://doi.org/10.14722/ndss.2015.23262

[31] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. USENIX Association, San Diego, CA, 401–416. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/davi

[32] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge Me If You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for x86 and ARM. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIACCS '13)*. ACM, Hangzhou, China, 299–310. https://doi.org/10.1145/2484313.2484351

[33] Yufei Du, Zhuojia Shen, Komail Dharsee, Jie Zhou, Robert J. Walls, and John Criswell. 2022. Holistic Control-Flow Protection on Real-Time Embedded Systems with Kage. In *Proceedings of the 31st USENIX Security Symposium (Security '22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/du

[34] EEMBC. 2018. *CoreMark: An EEMBC Benchmark*. https://www.eembc.org/coremark

[35] EEMBC. 2019. *CoreMark-Pro: An EEMBC Benchmark*. https://www.eembc.org/coremark-pro

[36] Embedded Security. 2018. *PinLock*. https://github.com/embedded-sec/ACES/tree/master/test_apps/pinlock

[37] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the

Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, 901–913. https://doi.org/10.1145/2810103.2813646

[38] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. 2019. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, Providence, RI, 469–484. https://doi.org/10.1145/3297858.3304037

[39] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium (Security '12)*. USENIX Association, Bellevue, WA, 475–490. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida

[40] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, San Jose, CA, 575–589. https://doi.org/10.1109/SP.2014.43

[41] Javid Habibi, Aditi Gupta, Stephen Carlsony, Ajay Panicker, and Elisa Bertino. 2015. MAVR: Code Reuse Stealthy Attacks and Mitigation on Unmanned Aerial Vehicles. In *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS '15)*. IEEE Computer Society, Columbus, OH, 642–652. https://doi.org/10.1109/ICDCS.2015.71

[42] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, San Francisco, CA, 571–585. https://doi.org/10.1109/SP.2012.39

[43] Tomoaki Kawada, Shinya Honda, Yutaka Matsubara, and Hiroaki Takada. 2021. TZmCFI: RTOS-Aware Control-Flow Integrity Using TrustZone for Armv8-M. *International Journal of Parallel Programming* 49 (April 2021), 216–236. https://doi.org/10.1007/s10766-020-00673-z

[44] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, Miami Beach, FL, 339–348. https://doi.org/10.1109/ACSAC.2006.9

[45] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-Assisted Code Randomization. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP '18)*. IEEE Computer Society, San Francisco, CA, 461–477. https://doi.org/10.1109/SP.2018.00029

[46] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX Association, Broomfield, CO, 147–163. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[47] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proceedings of the 28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, 231–247. https://www.usenix.org/conference/usenixsecurity19/presentation/kwon

[48] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Palo Alto, CA, 12 pages. https://doi.org/10.1109/CGO.2004.1281665

[49] Seongman Lee, Hyeonwoo Kang, Jinsoo Jang, and Brent Byunghoon Kang. 2022. SaVioR: Thwarting Stack-Based Memory Safety Violations by Randomizing Stack Layout. *IEEE Transactions on Dependable and Secure Computing* (July 2022), 2559–2575. https://doi.org/10.1109/TDSC.2021.3063843

[50] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. 2009. Polymorphing Software by Randomizing Data Structure Layout. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer-Verlag, Como, Italy, 107–126. https://doi.org/10.1007/978-3-642-02918-9_7

[51] LLVM 2014. *llvm::RandomNumberGenerator Class Reference*. https://llvm.org/doxygen/classllvm_1_1RandomNumberGenerator.html

[52] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16)*. Internet Society, San Diego, CA, 15 pages. https://doi.org/10.14722/ndss.2016.23173

[53] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, Denver, CO, 280–291. https://doi.org/10.1145/2810103.2813694

[54] Lan Luo, Xinhui Shao, Zhen Ling, Huaiyu Yan, Yumeng Wei, and Xinwen Fu. 2022. fASLR: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems. *IEEE Internet of Things Journal* 9, 18 (Sept. 2022), 17120–17135. https://doi.org/10.1109/JIOT.2022.3190374

[55] Mbed TLS Contributors. 2009. *Mbed TLS Benchmark Demonstration Program*. https://github.com/ARMmbed/mbedtls/blob/development/programs/test/benchmark.c

[56] Microchip 2020. *32-bit Microcontroller Families: Industry's Broadest and Most Innovative 32-bit MCU Portfolio*. Microchip. DS30009904V.

[57] Gene Novark and Emery D. Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, IL) *(CCS '10)*. ACM, 573–584. https://doi.org/10.1145/1866307.1866371

[58] NXP 2021. *UM11147 User Manual: RT6xx User Manual*. NXP. Rev. 1.4.

[59] NXP 2021. *UM11159 User Manual: i.MX RT685 Evaluation Board User Manual*. NXP. Rev. 2.

[60] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID '17)*. Springer-Verlag, Atlanta, GA, 259–284. https://doi.org/10.1007/978-3-319-66332-6_12

[61] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7 (Nov. 1996). Issue 49. http://www.phrack.org/issues/49/14.html

[62] James Pallister, Simon Hollis, and Jeremy Bennett. 2013. BEEBS: Open Benchmarks for Energy Measurements on Embedded Platforms. *arXiv preprint arXiv:1308.5174* (Aug. 2013). arXiv:1308.5174 [cs.PF] https://arxiv.org/abs/1308.5174

[63] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, San Francisco, CA, 601–615. https://doi.org/10.1109/SP.2012.41

[64] Sergio Pastrana, Juan Tapiador, Guillermo Suarez-Tangil, and Pedro Peris-López. 2016. AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*. Springer-Verlag, San Sebastián, Spain, 58–77. https://doi.org/10.1007/978-3-319-40667-1_4

[65] PaX Team. 2001. *Address Space Layout Randomization*. https://pax.grsecurity.net/docs/aslr.txt

[66] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. 2017. Breaking and Fixing Destructive Code Read Defenses. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*. ACM, Orlando, FL, 55–67. https://doi.org/10.1145/3134600.3134626

[67] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. ACM, Belgrade, Serbia, 420–436. https://doi.org/10.1145/3064176.3064216

[68] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. Practical Fine-Grained Binary Code Randomization. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC '20)*. ACM, Austin, TX, 401–414. https://doi.org/10.1145/3427228.3427292

[69] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS '20)*. ACM, Taipei, China, 494–505. https://doi.org/10.1145/3320269.3384757

[70] Renesas 2022. *RA Family Brochure*. Renesas. Document No. R01CP0035EJ0300.

[71] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[72] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS '17)*. Internet Society, San Diego, CA, 15 pages. https://doi.org/10.14722/ndss.2017.23477

[73] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, Alexandria, VA, 552–561. https://doi.org/10.1145/1315245.1315313

[74] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. ACM, Washington, DC, 298–307. https://doi.org/10.1145/1030083.1030124

[75] Zhuojia Shen, Komail Dharsee, and John Criswell. 2020. Fast Execute-Only Memory for Embedded Systems. In *Proceedings of the 2020 IEEE Secure Development Conference (SecDev '20)*. IEEE Computer Society, Atlanta, GA, 7–14. https://doi.org/10.1109/SecDev45635.2020.00017

[76] Jiameng Shi, Le Guan, Wenqiang Li, Dayou Zhang, Ping Chen, and Ping Chen. 2022. HARM: Hardware-assisted Continuous Re-randomization for Microcontrollers. In *Proceedings of the 2022 IEEE European Symposium on Security and Privacy (EuroSP '22)*. IEEE Computer Society, Genoa, Italy, 520–536. https://doi.org/10.1109/EuroSP53844.2022.00039

[77] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, San Francisco, CA, 574–588. https://doi.org/10.1109/SP.2013.45

[78] Alexander Sotirov. 2007. Heap Feng Shui in JavaScript. In *Black Hat Europe*.

[79] STMicroelectronics 2020. *DS12469 Datasheet: STM32L412xx*. STMicroelectronics. DS12469 Rev 8.

[80] STMicroelectronics 2021. *DS11189 Datasheet: STM32F469xx*. STMicroelectronics. DS11189 Rev 7.

[81] STMicroelectronics 2022. *AN4230 Application Note: STM32 Microcontroller Random Number Generation Validation Using the NIST Statistical Test Suite*. STMicroelectronics. Rev 7.

[82] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking the Memory Secrecy Assumption. In *Proceedings of the 2nd European Workshop on System Security (EuroSec '09)*. ACM, Nuremburg, Germany, 1–8. https://doi.org/10.1145/1519144.1519145

[83] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID '11)*. Springer-Verlag, Menlo Park, CA, 121–141. https://doi.org/10.1007/978-3-642-23644-0_7

[84] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi, and Bryan C. Ward. 2019. Control-Flow Integrity for Real-Time Embedded Systems. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS '19)*. Schloss Dagstuhl–Leibniz-Zentrum füer Informatik, Stuttgart, Germany, 2:1–2:24. https://doi.org/10.4230/LIPIcs.ECRTS.2019.2

[85] Zhe Wang, Chenggang Wu, Jianjun Li, Yuanming Lai, Xiangyu Zhang, Wei-Chung Hsu, and Yueqiang Cheng. 2017. 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 (VEE '17)*. ACM, Xi'an, China, 143–156. https://doi.org/10.1145/3050748.3050752

[86] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. 2019. SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-Randomization. In *Proceedings of the 28th USENIX Security Symposium (Security '19)*. USENIX Association, Santa Clara, CA, 1239–1256. https://www.usenix.org/conference/usenixsecurity19/presentation/wang

[87] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, Raleigh, NC, 157–168. https://doi.org/10.1145/2382196.2382216

[88] Mario Werner, Thomas Unterluggauer, David Schaffenrath, and Stefan Mangard. 2018. Sponge-Based Control-Flow Protection for IoT Devices. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroSP '18)*. IEEE Computer Society, London, United Kingdom, 214–226. https://doi.org/10.1109/EuroSP.2018.00023

[89] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 367–382. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/williams-king

[90] XAMPPRocky and contributors. 2015. *Tokei: Count your code, quickly.* https://github.com/XAMPPRocky/tokei

[91] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. 2020. Silhouette: Efficient Protected Shadow Stacks for Embedded Systems. In *Proceedings of the 29th USENIX Security Symposium (Security '20)*. USENIX Association, Boston, MA, 1219–1236. https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie

A DIVERSIFIED BINARY DEPLOYMENT

Deploying and updating diversified MCU application binaries provides challenges to software developers. However, we believe that such challenges can be readily addressed. When a device manufacturer releases a new version of software for an MCU, they can first translate all compilation units to LLVM IR and link the files into a single LLVM IR file containing all the code using LLVM's LTO features [48]. They can then, for each device, generate random seeds using a CSPRNG or TRNG, have the compiler's code generator translate the LLVM IR into a randomized binary using those seeds, and then record in a database the hash of the generated binary and the seeds that were used to create it.

When a customer submits a crash dump or requests a service from the device manufacturer that requires knowing which diversified binary the customer is using, the customer can simply supply the hash of their binary file. The device manufacturer can then feed the corresponding random seeds from the database into the code generator and regenerate the randomized binary. In this way, the device manufacturer can always re-create the randomized binary without having to store a copy of each binary given to a customer.

Table 8: Definitions of Mathematical Symbols Used in the Security Evaluation

Symbol	Definition
S_C	Size of randomized code segment
S_{C_O}	Size of original application code
S_T	Size of control flow target
S_D	Size of randomized `.data` segment
$S_{D'}$	Size of memory in `.data` that does not resemble control data
S_{D_0}	Size of zeroed memory in `.data`
S_G	Total size of all global guards
S_W	Size of memory in `.data` that attacker chooses to corrupt
N	Number of control data slots in `.data`
$p_{S,x}$	Probability of success w/ Strategy x
$p_{T,x}$	Probability of finding/hitting a trap w/ Strategy x
p_S	Probability of success
p_T	Probability of trapping the system
P_x	Number of guesses for a success in Strategy x
P	Number of brute force attacks for a success
$E(X)$	Expected value of random variable X
t_B	Time from booting to reaching an exploitable vulnerability
t_N	Time for attacker to send and receive data over network
T_n	Expected time to resist brute force attacks w/o delayed reboot
T_d	Total time of delay provided by delayed reboot
T_{min}	Expected time to resist brute force attacks w/ delayed reboot
D_i	Time of delay at i-th reboot caused by security violation
R	Number of reboots after which reboot delay stops increasing

B SYMBOL DEFINITIONS

See Table 8.

C EQUATION DERIVATION

Derivation of Equation 1. To derive Equation 1, let F_{1a} be the search space size of Strategy 1a. We have

$$F_{1a} = \frac{S_C - S_T + 2}{2}.$$

We hereafter use p_i to represent the probability of success at i-th guess regardless of the strategy. In Strategy 1a, we have

$$p_i = \frac{\cancel{F_{1a} - 1}}{F_{1a}} \cdot \frac{\cancel{F_{1a} - 2}}{\cancel{F_{1a} - 1}} \cdots \frac{\cancel{F_{1a} - i + 1}}{\cancel{F_{1a} - i + 2}} \cdot \frac{1}{\cancel{F_{1a} - i + 1}} = \frac{1}{F_{1a}}.$$

So the expected number of guesses for a success in Strategy 1a, $E(P_{1a})$, can be expressed by:

$$E(P_{1a}) = \sum_{i=1}^{F_{1a}} i \cdot p_i = \sum_{i=1}^{F_{1a}} i \cdot \frac{1}{F_{1a}} = \frac{S_C - S_T + 4}{4}.$$

Derivation of Equation 2. To derive Equation 2, we first have

$$p_i = (1 - p_{S,1b})^{i-1} p_{S,1b}.$$

So

$$E(P_{1b}) = \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i(1 - p_{S,1b})^{i-1} p_{S,1b}.$$

According to geometric distribution,

$$E(P_{1b}) = \frac{1}{p_{S,1b}} = \frac{S_D - S_{D'}}{4}.$$

Derivation of Equation 3. Similar to the derivation of Equation 1, let F_{1c} be the search space size of Strategy 1c. We have

$$F_{1c} = \frac{S_D - S_{D'}}{4}$$

and

$$p_i = \frac{1}{F_{1c}}.$$

So

$$E(P_{1c}) = \sum_{i=1}^{F_{1c}} i \cdot p_i = \sum_{i=1}^{F_{1c}} i \cdot \frac{1}{F_{1c}} = \frac{S_D - S_{D'} + 4}{8}.$$

Derivation of Equation 4. Similar to the derivation of Equation 2, we have

$$p_i = (1 - p_{S,2a})^{i-1} p_{S,2a}.$$

and

$$E(P_{2a}) = \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i(1 - p_{S,2a})^{i-1} p_{S,2a}.$$

According to geometric distribution,

$$E(P_{2a}) = \frac{1}{p_{S,2a}} = \frac{S_D - S_{D'}}{4N}.$$

Derivation of Equations 5, 6, and 7. For Strategy 2b, the condition of success is by corrupting at least one of N control data slots while not hitting any of the current global guards, and the condition of trapping the system is by hitting any of the current global guards. This can be modeled as the following situation:

- The attacker picks $\frac{S_W}{4}$ consecutive bins out of $\frac{S_D}{4}$ bins sorted in a certain order, N of which are black (representing control data slots) and $\frac{S_G}{4}$ of which are red (representing the current global guards).
- The attacker succeeds if the $\frac{S_W}{4}$ bins she picks contain no red bin and at least one black bin.
- The attacker traps the system if the $\frac{S_W}{4}$ bins she picks contain at least one red bin.

The total number of different bin permutations (denoted as A) is $\frac{S_D}{4}!$. The number of bin combinations in which the attacker succeeds (denoted as C_S) is the number of all possible combinations of i black bins and $\frac{S_W}{4} - i$ non-black non-red bins ($i \in \{1, 2, \ldots, \min(N, \frac{S_W}{4})\}$), which can be calculated by

$$C_S = \sum_{i=1}^{\min(N, \frac{S_W}{4})} C(N, i) C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4} - i).$$

This number can then be used to calculate the number of bin permutations in which the attacker succeeds (denoted as A_S), by multiplying it with the number of all possible permutations with the starting location of the $\frac{S_W}{4}$ bins fixed (as the bins the attacker picks must be consecutive). So we have

$$A_S = C_S \cdot \frac{S_W}{4}! \cdot \frac{S_D - S_W}{4}!$$

and therefore

$$p_{S,2b} = \frac{A_S}{A} = \frac{\sum_{i=1}^{\min(N, \frac{S_W}{4})} C(N, i) C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4} - i)}{C(\frac{S_D}{4}, \frac{S_W}{4})}.$$

We can calculate $p_{T,2b}$ indirectly by first calculating the probability of the attacker picking all $\frac{S_W}{4}$ bins as non-black non-red bins and then doing a subtraction from 1. Since the number of bin combinations of $\frac{S_W}{4}$ non-black non-red bins is $C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4})$, we can easily get

$$p_{T,2b} = 1 - p_{S,2b} - \frac{C(\frac{S_D - S_G}{4} - N, \frac{S_W}{4})}{C(\frac{S_D}{4}, \frac{S_W}{4})}.$$

Finally, $E(P_{2b})$ is derived in a similar way to that in $E(P_{1b})$ and in $E(P_{2a})$. We have

$$p_i = (1 - p_{S,2b})^{i-1} p_{S,2b}$$

and

$$E(P_{2b}) = \sum_{i=1}^{\infty} i \cdot p_i = \sum_{i=1}^{\infty} i(1 - p_{S,2b})^{i-1} p_{S,2b}.$$

According to geometric distribution,

$$E(P_{2b}) = \frac{1}{p_{S,2b}}.$$