Fast Execute-Only Memory for Embedded Systems

Zhuojia Shen
Department of Computer Science
University of Rochester
Rochester, NY
zshen10@cs.rochester.edu

Komail Dharsee
Department of Computer Science
University of Rochester
Rochester, NY
kdharsee@cs.rochester.edu

John Criswell
Department of Computer Science
University of Rochester
Rochester, NY
criswell@cs.rochester.edu

***Abstract*—Remote code disclosure attacks threaten embedded systems as they allow attackers to steal intellectual property or to find reusable code for use in control-flow hijacking attacks. Execute-only memory (XOM) prevents remote code disclosures, but existing XOM solutions either require a memory management unit that is not available on ARM embedded systems or incur significant overhead.**

We present *PicoXOM*: a fast and novel XOM system for ARMv7-M and ARMv8-M devices which leverages ARM's Data Watchpoint and Tracing unit along with the processor's simplified memory protection hardware. On average, PicoXOM incurs 0.33% performance overhead and 5.89% code size overhead on two benchmark suites and five real-world applications.

I. Introduction

Remote code disclosure attacks threaten computer systems. Remote attackers exploiting buffer overread vulnerabilities [41] can not only steal intellectual property (e.g., proprietary application code, for reverse engineering), but also leak code to locate gadgets for advanced code reuse attacks [37], thwarting code layout diversification defenses like Address Space Layout Randomization (ASLR) [32]. Embedded Internet-of-Things (IoT) devices exacerbate the situation; many of these microcontroller-based systems have the same Internet connectivity as desktops and servers but rarely employ protections against attacks [23], [36]. Given the ubiquity of these embedded devices in industrial production and in our lives, making them immune to code disclosure attacks is crucial.

Recent research [7]–[9], [11], [14], [18]–[20], [26], [33], [44] implements *execute-only memory (XOM)* to defend against code disclosure attacks. Despite being unable to prevent code pointer leakage from data regions such as heaps and stacks, XOM enforces memory protection on the code region so that instruction fetching is allowed but reading or writing instructions as data is disallowed. This simple and effective defense, however, is not natively available on low-end microcontrollers. For example, the ARMv7-M and ARMv8-M architectures used in mainstream devices support memory protection but not execute-only (XO) permissions [4], [5]. uXOM [26] implements XOM on ARM embedded systems but incurs significant performance and code size overhead (7.3% and 15.7%, respectively) as it transforms most load instructions into special unprivileged load instructions. Given embedded systems' real-time constraints and limited memory resources, a practically ideal XOM solution should have *close-to-zero performance penalty* and *minimal memory overhead*.

This paper presents *PicoXOM*, a fast and novel XOM system for ARMv7-M and ARMv8-M devices using a memory protection unit (MPU) and the Data Watchpoint and Tracing (DWT) unit [4], [5]. PicoXOM uses the MPU to enforce *write* protection on code and uses the unique *address range matching capability* of the DWT unit to control *read* access to the code region. On a matched access, the DWT unit generates a debug monitor exception indicating an illegal code read, while unmatched accesses execute normally without slowdown. As PicoXOM disallows all read accesses to the code segment, it includes a minimal compiler change that removes all data embedded in the code segment.

We built a prototype of PicoXOM and evaluated it on an ARMv7-M board with two benchmark suites and five real-world embedded applications. Our results show that PicoXOM adds *negligible performance overhead* of 0.33% and only has a *small code size increase* of 5.89% while providing strong protection against code disclosure attacks.

To summarize, our contributions are:

- PicoXOM: a novel method of utilizing the ARMv7-M and ARMv8-M debugging facilities to implement XOM. To the best of our knowledge, this is the first use of ARM debug features for security purposes.
- A prototype implementation of PicoXOM on ARMv7-M.
- An evaluation of PicoXOM's performance and code size impact on the BEEBS benchmark suite, the CoreMark-Pro benchmark suite, and five real-world embedded applications, showing that PicoXOM only incurs 0.33% runtime overhead and 5.89% code size overhead.

The rest of the paper is organized as follows. Section II provides background information on ARMv7-M and ARMv8-M. Section III describes our threat model and assumptions. Sections IV and VI present the design and implementation of PicoXOM, respectively. Section VII reports on our evaluation of PicoXOM, Section VIII discusses related work, and Section IX concludes the paper and discusses future work.

II. Background

PicoXOM targets ARMv7-M and ARMv8-M architectures, which cover a wide range of embedded devices on the market, and it leverages unique features of these architectures. This section provides important background material on the instruction sets, execution modes, address space layout, memory protection mechanisms, and on-chip debug support found in ARMv7-M and ARMv8-M.

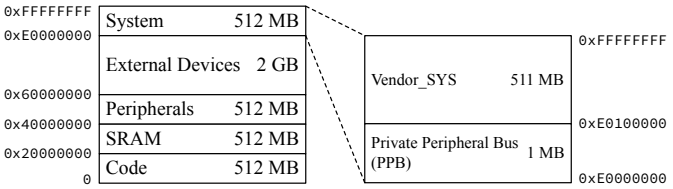

Fig. 1. Memory Layout of ARMv7-M and ARMv8-M Architectures

A. Instruction Sets and Execution Modes

ARMv7-M [4] and ARMv8-M [5] are the mainstream M-profile ARM architectures for embedded microcontrollers. Unlike ARM's A and R profiles, they only support the Thumb instruction set which is a mixture of 16-bit and 32-bit densely-encoded Thumb instructions.

ARMv7-M [4] supports two execution modes: unprivileged mode and privileged mode. An ARMv7-M processor always executes exception handlers in privileged mode, while application code is allowed to execute in either mode. Code running in unprivileged mode can raise the current execution mode to privileged mode using a supervisor call instruction (`SVC`). This is typically how ARMv7-M realizes system calls. However, embedded applications usually run in privileged mode to reduce the cost of system calls.

ARMv8-M inherits all the features of ARMv7-M and adds a security extension called TrustZone-M [5] that isolates software into a secure world and a non-secure world; this effectively doubles the execution modes as software can be executing in either world, privileged or unprivileged.

B. Address Space Layout

Both ARMv7-M [4] and ARMv8-M [5] architectures operate on a single 32-bit physical address space and use memory-mapped I/O to access external devices and peripherals. As Figure 1 shows, the address space is generally divided into eight consecutive 512 MB regions; the `Code` region maps flash memory/ROM that contains code and read-only data, the `SRAM` region typically contains heaps and stacks, and the `System` region holds memory-mapped system registers including a Private Peripheral Bus (PPB) subregion. The PPB subregion contains all critical system registers such as MPU configuration registers and the Vector Table Offset Register `VTOR`. All other regions are for memory-mapped peripherals and external devices. Note that ARMv7-M and ARMv8-M do not have special privileged instructions to access system registers mapped in the `System` region; instead, they can be modified by regular load and store instructions.

C. Memory Protection Unit

ARMv7-M and ARMv8-M devices do not have a memory management unit (MMU) that supports virtual memory; instead, they support an optional MPU that can be configured to enforce region-based access control on physical memory [4], [5]. A typical ARMv7-M device supports up to 8 MPU regions, each of which is configurable with a base address, a power-of-two size from 32 bytes to 4 GB, and separate access permissions (R, W, and X) for privileged and unprivileged modes. With TrustZone-M, ARMv8-M has separate MPU configurations for secure and non-secure worlds [5]. MPU configuration registers are in the PPB region.

There are, however, limitations on how one can configure access permissions for an MPU region. First, the privileged access permission cannot be more restrictive than the unprivileged one; this prohibits an MPU region with, for example, unprivileged read-write and privileged read-only permissions. Second, the PPB region is always privileged-accessible, unprivileged-inaccessible, and non-executable regardless of the MPU configuration. Third, and most importantly, the MPU does not have the execute-only permission necessary to support XOM; an MPU region is executable only if it is configured as both readable and executable.

D. Debug Support

Debug support is another processor feature that ARMv7-M and ARMv8-M devices can optionally support. Of all components in the architecture's debug support, we focus on the DWT unit [4], [5] which provides groups of debug registers called *DWT comparators* that support instruction/data address matching, PC value tracing, cycle counters, and many other functionalities. Most importantly, a DWT comparator enables monitoring of read accesses over a specified address range; if the processor reads from or writes to an address within a specified range, the DWT comparator will halt the software execution or generate a debug monitor exception. If, instead, the access does not fall into the specified range, execution proceeds as normal, and performance is unaffected. When multiple DWT comparators are configured for data address range matching, an access that hits any of them will trap.

On ARMv7-M, a DWT comparator can be configured to match an address range by programming its base address with a mask that specifies a power-of-two range size [4]. ARMv8-M implements DWT address range matching by using two consecutively numbered DWT comparators [5], where the first one specifies the lower bound of the address range and the second one specifies the upper bound.

III. Threat Model and System Assumptions

We assume a buggy but unmalicious application running on an embedded device with memory safety vulnerabilities that allow a remote attacker to read or write arbitrary memory locations. The attacker wants to either steal proprietary application code for purposes like reverse engineering or learn the application code layout in order to launch code reuse attacks such as Return-into-libc [42] and Return-Oriented Programming (ROP) [35] attacks. Physical and offline attacks are out of scope as we believe such attacks can be stopped by orthogonal defenses [24], [36]. Our threat model also assumes the application code and data is diversified, using techniques such as those in EPOXY [13]. Therefore, remotely tricking the buggy application into reading its code content becomes a reasonable choice for the attacker.

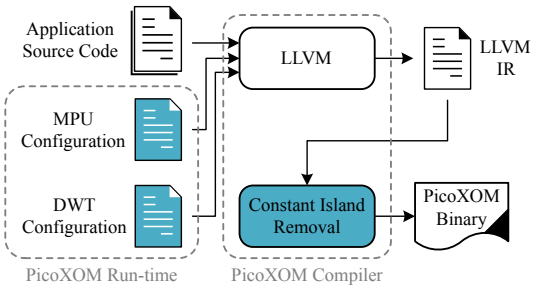

Fig. 2. PicoXOM Workflow. PicoXOM components are shown in blue.

We assume that the target embedded device supports MPU and DWT with enough configurable MPU regions and DWT comparators. We assume that the device is running a single bare-metal application statically linked with libraries, boot sequences, and exception handlers. The application is assumed to run in privileged mode, as Section II-A dictates. For ARMv8-M devices with TrustZone-M, the application is assumed to reside in the non-secure world, while software in the secure world is trusted.

IV. Design

Figure 2 shows PicoXOM's overall design. PicoXOM consists of three components that together implement a strong and efficient XOM on ARM embedded devices. First, PicoXOM uses a specially-configured DWT configuration to detect read accesses to program code. Second, it utilizes a special MPU configuration that prevents write access to the code region and prevents writeable memory from being executable. Third, it employs a small change to the LLVM compiler [27] to eliminate constant data embedded within the code region.

To use PicoXOM, embedded application developers merely compile their code with the PicoXOM compiler and install it on their embedded ARM device. On boot, the PicoXOM runtime configures MPU regions and DWT comparators using PicoXOM's MPU and DWT configurations and then passes control to the compiled embedded software.

A. W⊕X with MPU

PicoXOM requires that memory either be writeable or executable but not both i.e., the W⊕X policy [31]; otherwise, an attacker could simply inject code or overwrite code to achieve arbitrary code execution. To enforce W⊕X, PicoXOM configures the MPU regions at device boot time so that the code region is readable and executable, read-only data is read-only, and RAM regions are readable and writable. Note that the MPU *cannot* configure memory to be executable but unreadable; the MPU can configure a memory region as executable only if it is also configured as readable [4], [5].

PicoXOM runs application code in privileged mode and configures a background MPU region to allow read and write access to the remainder of the address space such as peripherals. This, however, leaves critical memory-mapped system registers in the PPB (such as MPU configuration registers and `VTOR`) open to modifications, which can be leveraged by an attacker to turn off MPU protections or, even worse, implant a custom exception handler. Section IV-B discusses how PicoXOM prevents such cases.

B. R⊕X with DWT

PicoXOM leverages ARM's DWT comparators to watch over the whole code region for read accesses. As Section II-D states, each (pair) of DWT comparators available on an ARM microcontroller can be configured to generate a debug monitor exception when a memory access of a specified type to an address within a specified range occurs. PicoXOM therefore uses one (pair) of the available DWT comparators as follows:

1) At device boot time, PicoXOM configures a DWT comparator register (say `DWT_COMP<n>`) to hold the lower bound of the code region.
2) PicoXOM then sets the address-matching range by either writing the upper bound of the code region to the next DWT comparator register `DWT_COMP<n+1>` (for ARMv8-M) or writing the correct mask to the corresponding DWT mask register `DWT_MASK<n>` (for ARMv7-M).
3) PicoXOM enables the DWT comparator (pair) by configuring the DWT function register `DWT_FUNC<n>` for data address reads. For ARMv8-M devices, `DWT_FUNC<n+1>` is also configured in order to form address range matching.
4) Finally, PicoXOM enables the debug monitor exception by setting the `MON_EN` bit (bit 16) of the Debug Exception and Monitor Control Register `DEMCR`.

With a DWT comparator (pair) set up for monitoring read accesses to the code region, R⊕X is effectively enforced. However, as Section IV-A stated, the DWT registers and `DEMCR` are also memory-mapped system registers which could be modified by vulnerable application code. An attacker could leverage a buffer overflow vulnerability to reconfigure the debug registers to neutralize PicoXOM.

We can address the issue in two ways. One approach is to break the assumption that PicoXOM runs everything in privileged mode. As code running in unprivileged mode has no access to the PPB region regardless of the MPU configuration, the system registers that PicoXOM must protect (e.g., MPU configuration registers, DWT registers, `DEMCR`, and `VTOR`) are all in the PPB region and therefore inherently safe from unprivileged tampering. However, this approach requires PicoXOM to implement system calls that support privileged operations which application code could previously perform, incurring expensive context switching between privilege modes. The other approach is to use extra (pairs of) DWT comparators to prevent writes to critical system registers. For example, on ARMv7-M, we can configure one DWT comparator to write-protect the System Control Block SCB (`0xE000ED00` – `0xE000ED8F`) and `DEMCR` (`0xE000EDFC`) by setting the lower bound and the size to `0xE000ED00` and 256 bytes, respectively. Since MPU configuration registers are in the SCB, they are protected as well. DWT registers on ARMv7-M reside in a separate range (`0xE0001000` – `0xE0001FFF`),

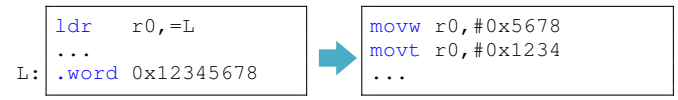

Fig. 3. Constant Island Removal of a Load Constant

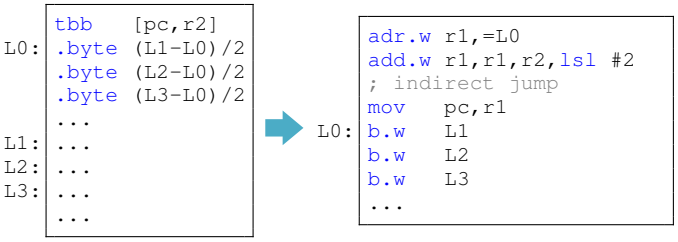

Fig. 4. Constant Island Removal of a Jump-Table Jump

so we can use another DWT comparator to write-protect that range.

C. Constant Island Removal

By default, ARM compilers generate code that has constant data embedded in the code region (so-called "constant islands"). Since PicoXOM prevents the code from reading these constant islands, these programs will fail to execute when used with PicoXOM. PicoXOM therefore transforms these programs so that all data within the program is stored outside of the code region.

We have identified two cases of constant islands generated by LLVM's ARM code generator: *load constants* and *jump-table jumps*. Figures 3 and 4 show examples of the two cases, respectively, as well as their corresponding execute-only versions to which PicoXOM transforms them. Specifically, in the left part of Figure 3, a load constant instruction loads a constant from a PC-relative memory location `L` into register `r0`. Such instructions are usually generated to quickly load an irregular constant in light of the limited immediate encoding scheme of the Thumb instruction set [4], [5]. PicoXOM transforms such load constants into `MOVW` and `MOVT` instructions that encode the 32-bit constant in two 16-bit immediates, as the right part of Figure 3 shows. Jump-table jump instructions (`TBB` and `TBH`) [4], [5] are used to implement large switch statements; the second register operand (`r2` in Figure 4) serves as an index into a jump table pointed to by the first register operand (`pc` in Figure 4), and a byte/half-word offset is loaded from the jump table to add to the program counter (`pc`) to calculate the target of the jump. Optimizing compilers like GCC and LLVM usually select `pc` as the first register operand in order to reduce register pressure, forcing the jump table to be located next to the jump-table jump itself. PicoXOM transforms such jump-table jumps into instruction sequences like that shown in the right part of Figure 4; it encodes the original jump table's contents into a sequence of branch instructions and expands the jump-table jump into a few explicit instructions that calculate which branch instruction to jump to and perform an indirect jump.

D. Return Address Nullification

Since the code segment is no longer readable, attackers cannot use buffer overread attacks [41] to read out the contents of the code memory to determine where reusable code is located. However, there may still be control data i.e., pointers into the code segment, sitting within readable data memory. Examples of control data are return addresses and function pointers. Attackers can use a buffer overread to leak control data to learn where code has been located and to rule out possible locations for brute-force attacks [**?**].

To mitigate such attacks, PicoXOM adds code before every function return that overwrites the return address stored on the stack with a zero; we call this new transformation *Return Address Nullification*. Return Address Nullification provides two key benefits. First, it limits the number of return addresses that a single buffer overread can leak. Specifically, a buffer overread can only leak the return addresses of actively executing functions i.e., if a buffer overread exists in a function `foo`, the buffer overread can only leak return addresses pointing into functions that are predecessors of `foo()` in the call graph. Second, Return Address Nullification allows program analysis to *quantify* the amount of code addresses that can be leaked by one or more buffer overreads, providing a way to quantify the danger posed by control data leakage attacks. We use this feature to measure the danger that code leakage attacks pose to PicoXOM in Section V-D.

V. Code Randomization

To protect software from code reuse attacks, PicoXOM randomizes the code before it is installed on the board; if the same software is deployed on many boards, each board will be running a variant with a different code layout. Due to memory constraints, PicoXOM does not re-randomize the code on boot or during operation. PicoXOM's XOM feature prevents buffer overreads [41] from reading the code segment. However, PicoXOM's XOM feature will be useless if attackers can simply guess the location of reusable code within a reasonable amount of time. The limited memory resources on ARMv7-M systems exacerbate this problem. The lack of an MMU [4] reduces the size of the usable address space to what is available in physical memory. Our evaluation board only has 2 MB of memory for code [40], limiting the number of options for code diversity. We therefore model brute force attacks on code randomization schemes and devise methods to make PicoXOM resistant against them.

A. Model of Brute Force Attacks

For our analysis, we model a simple code reuse attack in which an attacker attempts to call a single function within the program with no arguments (similar to a return to libc attack [42], but the function can be any function within the application code or libraries). We model this attack as it is the simplest viable attack that requires the attacker to guess the least amount of information about the code. If PicoXOM can resist such an attack, it should also be able to resist more sophisticated attacks (such as return-oriented programming

(ROP) attacks [35]) that require guessing several pieces of information about the application.

We assume that the application has a buffer overflow that the attacker can use to corrupt control data (such as a return address or function pointer). Without knowing how the code has been randomized, the attacker will merely guess the location of the desired function, exploit the buffer overflow, and see if the attack succeeds. We assume that a successful attack is easily observed by the attacker. We also assume that unsuccessful attacks will cause a trap; PicoXOM will restart the system when the trap occurs.

B. Randomization Schemes

We modeled three types of code layout randomization that PicoXOM could deploy: function layout randomization, basic block layout randomzation, and cluster randomization. Function layout randomization chooses random locations for each function within the code memory. The code for each function remains contiguous, and the instructions within each function appear in the same order. Trap instructions are placed anywhere within the code memory region in which a function is not placed.

Basic block layout randomization chooses a random location for each basic block. The transformation is interprocedural: a function's basic blocks do not (and most likely will not) be located contiguously within memory. As some basic blocks do not end with a branch instruction (because control flow "falls through" to the subsequent basic block in memory), our analysis considers the extra branch instructions that the compiler would need when a function's basic blocks will no longer be contiguous. Again, trap instructions are placed anywhere in memory where no code from the program resides.

Cluster randomization is a variation of basic block layout randomization. The difference is that it avoids adding jump instructions by grouping basic blocks into cluster. A cluster is a set of basic blocks in which control flow "falls through" from predecessor basic blocks to successor basic blocks. By grouping code into clusters, cluster randomization is identical to basic block layout randomization, but it does not add any branch instructions to the code.

C. Randomization Model Results

To study the amount of randomization feasible using PicoXOM, we extended the LLVM code generator to measure the number of different places a function, a basic block, and a basic block cluster can be placed within a 2 MB code memory region. If T is the total size of the memory in which code can be located (2 MB in our system), and if c is the largest unit of code that can be individually reordered (a function, basic block, or basic block cluster, respectively), then the maximum number l of locations in which a function can be placed is

$$l = T - c \tag{1}$$

TABLE I
RANDOMIZATION RESULTS FOR BEEBS

	Function Layouts	Basic Block Layouts	Cluster Layouts
aha-compress	2,092,550	2,096,274	None
aha-mont64	2,092,550	2,096,274	None
bubblesort	2,092,550	2,096,274	None
crc32	2,092,550	2,096,274	None
ctl-string	2,092,550	2,096,274	None
ctl-vector	2,092,550	2,096,274	None
dijkstra	2,092,550	2,096,274	None
dtoa	2,092,208	2,096,274	None
cubic	2,092,550	2,096,274	None
edn	2,092,550	2,096,274	None
fasta	2,092,550	2,096,274	None
fir	2,092,550	2,096,274	None
frac	2,092,550	2,096,274	None
huffbench	2,092,550	2,096,274	None
levenshtein	2,092,550	2,096,274	None
matmult-float	2,092,550	2,096,274	None
matmult-int	2,092,550	2,096,274	None
mergesort	2,092,550	2,096,274	None
nbody	2,092,550	2,096,274	None
ndes	2,092,550	2,096,274	None
nettle-aes	2,092,550	2,096,274	None
nettle-arcfour	2,092,550	2,096,274	None
picojpeg	2,092,550	2,096,274	None
qrduino	2,092,428	2,096,274	None
rijndael	2,092,550	2,095,068	None
sglib-arraybinsearch	2,092,550	2,096,274	None
sglib-arrayheapsort	2,092,550	2,096,274	None
sglib-arrayquicksort	2,092,550	2,096,274	None
sglib-dllist	2,092,550	2,096,274	None
sglib-hashtable	2,092,550	2,096,274	None
sglib-listinsertsort	2,092,550	2,096,274	None
sglib-listsort	2,092,550	2,096,274	None
sglib-queue	2,092,550	2,096,274	None
sglib-rbtree	2,092,550	2,096,274	None
slre	2,092,550	2,096,274	None
sqrt	2,092,550	2,096,274	None
st	2,092,550	2,096,274	None
stb_perlin	2,092,550	2,096,274	None
trio-snprintf	2,092,550	2,096,274	None
trio-sscanf	2,092,550	2,096,274	None
whetstone	2,092,550	2,096,274	None
wikisort	2,092,550	2,096,274	None

TABLE II
RANDOMIZATION RESULTS FOR COREMARK-PRO

	Function Layouts	Basic Block Layouts	Cluster Layouts
cjpeg-rose7-preset	2,092,258	2,096,274	None
core	2,092,550	2,096,274	None
linear_alg-mid-100x100-sp	2,092,550	2,096,274	None
loops-all-mid-10k-sp	2,092,550	2,096,274	None
nnet_test	2,092,550	2,096,274	None
parser-125k	2,092,550	2,096,274	None
radix2-big-64k	2,092,550	2,096,274	None
sha-test	2,090,288	2,090,288	None
zip-test	2,091,362	2,096,274	None

TABLE III
RANDOMIZATION RESULTS FOR APPLICATIONS

	Function Layouts	Basic Block Layouts	Cluster Layouts
pinlock	2,092,258	2,096,148	None
fatfs_ram	2,092,550	2,096,274	None
fatfs_usd	2,092,550	2,096,274	None
lcd_animation	2,092,550	2,095,966	None
lcd_usd	2,092,550	2,095,966	None

Our analysis pass is executed during code generation at link-time, so functions and basic blocks from different compilation units can be reordered. Tables I, II, and III show the results.

Our first observation is that the number of locations for the beginning of a function is too few to offer protection against brute force attacks. As Table I shows, the greatest number of locations is $2,096,274$ when basic block randomization is employed. If a single attack attempt (a "probe") takes 3 milliseconds (ms), then trying every possible location in crc32 will take a mere 1.7 hours. In order to make the randomization effective, we must increase the amount of time of each probe so that, collectively, the time needed to probe the system to guess the location of the code to reuse is sufficiently high.

Our second observation is that randomized function layout is nearly as effective as randomizing the layout of basic blocks and basic block clusters. As Table II shows, the smallest number of code locations is $2,090,288$ using randomized function layout on CoreMark-Pro's `sha-test`. Using probes that take 3 ms, the maximum time to brute-force an attack is also 1.7 hours.

To make randomization practical, PicoXOM uses function randomization to randomize the layout of code. Additionally, the PicoXOM boot code for the device adds an artificial delay to boot; this delay starts at 1 ms. PicoXOM modifies the trap handler so that a reboot due to a crash (for example, by execution of a trap instruction within the empy areas of the code memory) doubles the delay time up to a maximum of 30 seconds. This delay ensures that it will take the attacker at least a year to brute-force her way into a PicoXOM-protected system.

D. Control Data Leakage Model Results

A control data leakage attack is an attack in which a buffer overread [41] is used to leak a pointer to the code segment to an attacker [**?**]; examples of such control data are return addresses and function pointers. Attackers can use such leaks in two ways. First, a leaked control data value may allow an attacker to directly infer the location of a specific piece of code that the attacker wishes to use in a code reuse attack. For example, if the location of functions is randomized within the code memory, leaking a return address located within a function allows the attacker to compute that function's start address. Second, if a buffer overread cannot leak control data pointing into a function for which the attacker is searching, it will eliminate the possible set of locations to use in a brute-force attack. By learning the locations of other functions, the attacker can rule out locations for the function for which she is searching.

We first evaluated how many return addresses can be leaked by a single buffer overread. As Section IV-D explains, when

TABLE IV
CONTROL DATA LEAKAGE RESULTS FOR BEEBS

	Maximum Leak	Function Layouts
aha-compress	4	2,087,808
aha-mont64	4	2,087,808
bubblesort	4	2,087,808
crc32	4	2,087,808
ctl-string	4	2,087,808
ctl-vector	4	2,087,808
dijkstra	4	2,087,808
dtoa	5	2,086,556
cubic	4	2,087,808
edn	4	2,087,808
fasta	4	2,087,808
fir	4	2,087,808
frac	4	2,087,808
huffbench	4	2,087,808
levenshtein	4	2,087,808
matmult-float	4	2,087,808
matmult-int	4	2,087,808
mergesort	4	2,087,808
nbody	4	2,087,808
ndes	4	2,087,808
nettle-aes	4	2,087,808
nettle-arcfour	4	2,087,808
picojpeg	6	2,087,240
qrduino	4	2,085,800
rijndael	4	2,085,820
sglib-arraybinsearch	4	2,087,808
sglib-arrayheapsort	4	2,087,808
sglib-arrayquicksort	4	2,087,808
sglib-dllist	4	2,087,808
sglib-hashtable	5	2,087,808
sglib-listinsertsort	4	2,087,808
sglib-listsort	4	2,087,808
sglib-queue	4	2,087,808
sglib-rbtree	4	2,087,808
slre	5	2,087,808
sqrt	4	2,087,808
st	4	2,087,808
stb_perlin	4	2,087,808
trio-snprintf	6	2,087,808
trio-sscanf	6	2,087,808
whetstone	4	2,087,808
wikisort	4	2,087,808

TABLE V
CONTROL DATA LEAKAGE RESULTS FOR COREMARK-PRO

	Maximum Leak	Function Layouts
cjpeg-rose7-preset	7	2,087,364
core	5	2,087,808
linear_alg-mid-100x100-sp	6	2,087,808
loops-all-mid-10k-sp	6	2,087,808
nnet_test	6	2,087,808
parser-125k	7	2,087,536
radix2-big-64k	6	2,087,808
sha-test	5	2,083,024
zip-test	6	2,080,794

TABLE VI
CONTROL DATA LEAKAGE RESULTS FOR APPLICATIONS

	Maximum Leak	Function Layouts
pinlock	4	2,087,316
fatfs_ram	7	2,085,836
fatfs_usd	7	2,085,910
lcd_animation	8	2,087,808
lcd_usd	8	2,087,808

Algorithm 1 Find All Paths Algorithm

```
1:  PathSet: Set of paths within call graph
2:
3:  function FINDALLPATHS(Start,Path,Visited)
4:      if Start has no children then
5:          PathSet ← PathSet ∪ Path
6:          return
7:      end if
8:
9:      for all Children child of Start do
10:         if child ∩ Visited = ∅ then
11:             Visited ← Visited ∪ child
12:             Path ← Path + Start
13:             FindAllPaths (child,Path,Visited)
14:             Path ← Path - Start
15:             Visited ← Visited - child
16:         else
17:             PathSet ← PathSet ∪ Path
18:             return
19:         end if
20:     end for
21: end function
```

Return Address Nullification is used, a buffer overread can only leak the return address of active functions i.e., functions that are predecessors in the program's call graph. We therefore built an analysis into the PicoXOM code generator. This analysis, shown in Algorithm 1, creates a list of all paths through the call graph. Each path either terminates at a leaf function i.e., a function that calls no other functions, at a function that is part of an external library, or at the first function that is called again as part of a set of recursively called functions. A buffer overread in the last function of the longest path can leak the most return addresses.

We ran this analysis on our benchmark programs; the second column entitled *Maximum Leak* in Tables IV, V, and VI show our results. As our results show, with Return Address Nullification employed, the length of the longest path in the call graph for our embedded applications is small with a minimum of 4 and a maximum of 8.

To evaluate the amount of entropy lost when buffer overreads leak return addresses, we enhanced the above analysis to calculate the amount of native code that can be located in each path through the call graph if there is a buffer overread in the last function of each path. Our analysis uses Algorithm 2 to determine the largest number of code locations that can be

Algorithm 2 Compute Maximum Code Leak Algorithm

```
1:  PathSet: Set of paths within call graph
2:  FuncSize: Size of function computed by code generator
3:
4:  function CONTROLDATALEAK(PathSet)
5:      MaxLeakSize ← 0
6:      for all Paths path in PathSet do
7:          LeakSize ← 0
8:          for all Functions f in path do
9:              LeakSize ← LeakSize + FuncSize[f]
10:         end for
11:         if LeakSize > MaxLeakSize then
12:             MaxLeakSize ← LeakSize
13:         end if
14:     end for
15:
16:     return MaxLeakSize
17: end function
```

determined using a single buffer overread. If T is the total amount of code memory on a system, f is the size of the largest function within the program, and l is the largest amount of code locations leaked through a buffer overread, then the minimum number of code locations p that need to be probed in a brute-force attack is

$$p = T - (f + l) \tag{2}$$

Algorithm 2 therefore scans through the list of paths to determine which path can leak the most code if there is a buffer overread in the last function of the path.

The third column of Tables IV, V, and VI entitled *Function Layouts* shows the minimum number of code locations that need to be probed for each of our benchmark programs. As our results show, there are at least 2 million locations at which a function can be located even if a single buffer overread leaks the maximum number of return addresses possible. If a leaked return address does not reveal the location of the attacker's desired function, the attacker must still probe 2 million locations. With probes taking 3 ms, that will take about 1.6 hours (compared to 1.7 hours if no return addresses are leaked at all). We conclude that our method of delaying reboot when probes fail still suffices even when return addresses are leaked.

VI. IMPLEMENTATION

We built our PicoXOM prototype for the ARMv7-M architecture. Our prototype provides MPU and DWT configurations as a run-time component written in C and executed at the end of the device boot sequence. We implemented constant island removal as a simple intermediate representation (IR) pass in the LLVM 10.0 compiler [27]. The constant island removal pass simply uses the existing `-mexecute-only` option in LLVM's Clang front-end and passes it along to the link-time optimization (LTO) code generator. Our prototype runs the constant island removal pass when linking the IR of

the application, libraries (e.g., newlib and compiler-rt), and MPU and DWT configurations; this ensures that all code has no constant islands. Our prototype adds 88 source lines of C++ code to LLVM and has 177 source lines of C code in the PicoXOM run-time. We leave the PicoXOM implementation on ARMv8-M for future work.

Different ARM microcontrollers support different numbers of MPU regions and DWT comparators, and the maximum ranges of their DWT comparators may vary. Our prototype runs on an STM32F469 Discovery board which supports up to 8 MPU regions [39] and 4 DWT comparators [40]. Each DWT comparator can only watch over a maximum address range of 32 KB (a maximal mask value of 15), limiting our prototype to the following two options:

1) Use all 4 DWT comparators to support a maximum code size of 128 KB; the application must run in unprivileged mode in order for the critical system registers to be write-protected.

2) Configure one DWT comparator to write-protect the DWT registers (`0xE0001000` – `0xE0001FFF`) and another to write-protect the SCB (`0xE000ED00` – `0xE000ED8F`) and `DEMCR` (`0xE000EDFC`). This protects a maximum code size of 64 KB using the remaining 2 DWT comparators.

To accommodate a wider range of applications on our board with less performance loss, our prototype automatically chooses one option over the other based on the application code size. It rejects an application if the code size exceeds our board's 128 KB limit.

While our PicoXOM prototype only supports single bare-metal embedded applications, PicoXOM can also support multiple applications running on an embedded real-time operating system (RTOS) such as Amazon FreeRTOS [3]. On embedded systems, the application and RTOS kernel code is linked into a single shared code segment. PicoXOM can protect this code segment with little adaptation.

VII. Evaluation

We evaluated PicoXOM on our STM32F469 Discovery board [40] which has an ARM Cortex-M4 processor implementing the ARMv7-M architecture that can run as fast as 180 MHz. The board comes with 2 MB of flash memory, 384 KB of SRAM, and 16 MB of SDRAM, and has an LCD screen and a microSD card slot. We configured the board to run at its fastest speed to understand the maximum impact that PicoXOM can incur on performance.

To evaluate PicoXOM's performance and code size overhead, we used the BEEBS [30] and CoreMark-Pro [16] benchmark suites and five embedded applications (FatFs-RAM, FatFs-uSD, LCD-Animation, LCD-uSD, and PinLock). **BEEBS** targets energy consumption measurement for embedded platforms and is widely used in evaluating embedded systems including uXOM [26], the state-of-the-art XOM implementation on ARM microcontrollers. It consists of a wide range of programs characterizing different workloads seen on embedded systems, including AES encryption, data compression, and matrix multiplication. Of all 80 benchmarks in BEEBS, we picked 42 benchmarks that have an execution time longer than 500 milliseconds when executed for 10,240 iterations. **CoreMark-Pro** is a processor benchmark suite that works on both high-performance processors and low-end microcontrollers, featuring five integer benchmarks (e.g., JPEG image compression, XML parser, and SHA-256) and four floating-point benchmarks (e.g., fast Fourier transform and neural network) that stress the CPU and memory. **FatFs-RAM** and **FatFs-uSD** operate a FAT file system on SDRAM and an SD card, respectively. **LCD-Animation** displays a single animated picture loaded from an SD card. **LCD-uSD** displays multiple static pictures from an SD card with fading transitions. **PinLock** simulates a smart lock reading user input from a serial port and deciding whether to unlock (send an I/O signal) based on whether the SHA-256 hashed input matches a precomputed hash. The above five applications represent real-world use cases of embedded devices and were also used to evaluate previous work [2], [12], [13].

TABLE VII
Performance Overhead on BEEBS

	Baseline (ms)	PicoXOM (×)		Baseline (ms)	PicoXOM (×)
aha-compress	821	1.0000	nettle-arcfour	814	1.0000
aha-mont64	856	0.9988	picojpeg	43,864	1.0027
bubblesort	4,392	1.0000	qrduino	40,877	1.0030
crc32	956	1.0000	rijndael	70,024	1.0018
ctl-string	630	1.0000	sglib-arraybin...	808	1.0000
ctl-vector	786	0.9987	sglib-arrayhea...	1,039	1.0000
cubic	35,140	1.0005	sglib-arrayqui...	735	1.0000
dijkstra	36,582	1.0000	sglib-dllist	1,800	1.0000
dtoa	631	1.0127	sglib-hashtable	1,302	1.0000
edn	3,167	1.0003	sglib-listinsert...	2,030	1.0000
fasta	16,900	0.9999	sglib-listsort	1,265	1.0008
fir	16,048	1.0000	sglib-queue	1,177	1.0000
frac	5,858	1.0323	sglib-rbtree	4,808	1.0025
huffbench	20,682	0.9995	slre	2,761	0.9873
levenshtein	2,685	1.0000	sqrt	38,506	1.0748
matmult-float	1,150	0.9991	st	20,906	1.0252
matmult-int	4,532	1.0000	stb_perlin	5,132	1.0306
mergesort	24,353	1.0062	trio-snprintf	697	1.0100
nbody	128,126	1.0090	trio-sscanf	1,064	0.9915
ndes	2,039	0.9995	whetstone	112,754	1.0092
nettle-aes	5,687	0.9998	wikisort	113,195	1.0008
Min (×)					0.9873
Max (×)					1.0748
Geomean (×)					1.0046

We used the LLVM compiler infrastructure [27] to compile benchmarks and applications into the default non-XO format, with MPU and DWT disabled; this is our baseline. We then used PicoXOM's configuration, i.e. enabling MPU, DWT, and constant island removal. Note that with PicoXOM, none of the benchmarks and applications exceeds the code size limitation (128 KB) on our board. Only `cjpeg-rose7-preset` in CoreMark-Pro has a code size larger than 64 KB and thereby has to run in unprivileged mode; nevertheless, it does not require source code modifications as it does not perform privileged operations.

A. Performance

We measured PicoXOM's performance on our benchmarks and applications. We configured each BEEBS benchmark to print the time, in milliseconds, for executing its workload 10,240 times. We ran each BEEBS benchmark 10 times

TABLE VIII
PERFORMANCE OVERHEAD ON COREMARK-PRO

	Baseline (ms)	PicoXOM (×)		Baseline (ms)	PicoXOM (×)
cjpeg-rose7-...	10,200	1.0001	parser-125k	12,363	1.0012
core	83,160	0.9918	radix2-big-64k	21,955	0.9961
linear_alg-...	22,962	1.0000	sha-test	25,463	0.9995
loops-all-...	33,830	0.9995	zip-test	23,227	1.0000
nnet_test	282,398	1.0017			
Min (×)					0.9918
Max (×)					1.0017
Geomean (×)					0.9989

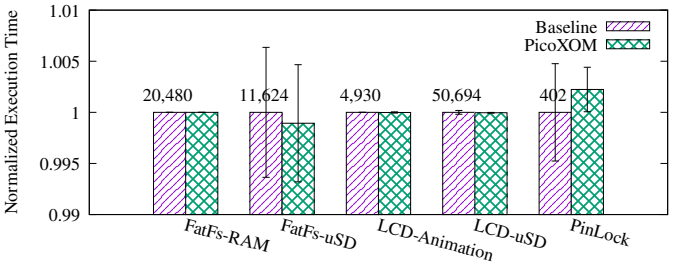

Fig. 5. Performance Overhead on Real-World Applications

and report the average execution time. Each CoreMark-Pro benchmark is pre-programmed to print out the execution time in a similar way; the difference is that we configure each benchmark to run a minimal number of iterations so that the program takes at least 10 seconds to run for each experimental trial. Again, we ran each benchmark 10 times and report the average execution time. For the real-world applications, we ran FatFs-RAM 10 times and report the average execution time. The other applications exhibit higher variance in their execution times as they access peripherals like an SD card, an LCD screen, and a serial port, so we ran them 20 times and report the average with a standard deviation. All other programs exhibit a standard deviation of zero.

Tables VII and VIII and Figure 5 present PicoXOM's performance on BEEBS, CoreMark-Pro, and the five real-world applications, respectively; Figure 5 shows baseline execution time in milliseconds on top of the Baseline bars. Overall, PicoXOM incurs negligible performance overhead of 0.33%: 0.46% on BEEBS with a maximum of 7.48%, −0.11% on CoreMark-Pro with a maximum of 0.17%, and 0.02% on the applications with a maximum of 0.22%. Thirteen programs exhibit a minor speedup with PicoXOM. We re-ran our experiments with the MPU and DWT disabled so that the only change to performance is due to constant island removal and the alignment of the code segment (the DWT on ARMv7-M requires the monitored address range to be aligned by its power-of-two size). In this configuration, we observed the same speedups, so either constant island removal and/or code alignment is causing the slight performance improvement.

B. Code Size

We measured the code size of benchmarks and applications by using the `size` utility on generated binaries and collecting the `.text` segment size.

TABLE IX
CODE SIZE OVERHEAD ON BEEBS

	Baseline (bytes)	PicoXOM (×)		Baseline (bytes)	PicoXOM (×)
aha-compress	30,164	1.0646	nettle-arcfour	29,988	1.0649
aha-mont64	31,236	1.0624	picojpeg	36,620	1.0599
bubblesort	29,868	1.0650	qrduino	37,228	1.0529
crc32	29,804	1.0654	rijndael	37,460	1.0516
ctl-string	30,668	1.0631	sglib-arraybin...	29,828	1.0654
ctl-vector	30,892	1.0624	sglib-arrayhea...	29,956	1.0651
cubic	42,428	1.0329	sglib-arrayqui...	30,036	1.0649
dijkstra	30,220	1.0644	sglib-dllist	30,364	1.0641
dtoa	36,204	1.0552	sglib-hashtable	30,164	1.0644
edn	30,940	1.0633	sglib-listinsert...	30,052	1.0649
fasta	29,956	1.0650	sglib-listsort	30,100	1.0648
fir	29,884	1.0651	sglib-queue	29,988	1.0650
frac	30,468	1.0626	sglib-rbtree	30,564	1.0639
huffbench	30,988	1.0628	slre	32,284	1.0603
levenshtein	30,140	1.0647	sqrt	30,372	1.0641
matmult-float	30,108	1.0644	st	31,124	1.0602
matmult-int	30,060	1.0650	stb_perlin	31,140	1.0627
mergesort	30,852	1.0604	trio-snprintf	33,724	1.0675
nbody	30,684	1.0633	trio-sscanf	34,156	1.0668
ndes	31,028	1.0630	whetstone	40,164	1.0371
nettle-aes	31,756	1.0614	wikisort	34,332	1.0541
Min (×)					1.0329
Max (×)					1.0675
Geomean (×)					1.0614

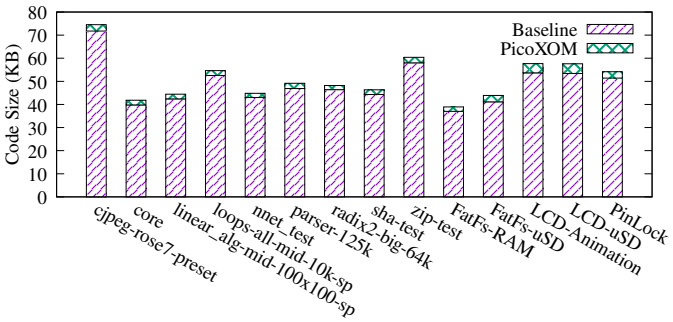

Fig. 6. Code Size Overhead on CoreMark-Pro and Real-World Applications

Table IX and Figure 6 show the baseline code size and the overhead incurred by PicoXOM on BEEBS, CoreMark-Pro, and the five real-world applications, respectively. On average, PicoXOM increases the code size by 6.14% on BEEBS, 4.39% on CoreMark-Pro, and 6.52% on the real-world applications, with a 5.89% overall overhead. We studied PicoXOM's code size overhead and discovered that constant island removal caused the majority of the code size overhead, especially for programs with relatively large code bases like CoreMark-Pro. In fact, the additional code that sets up the MPU and DWT only contributes a minor part of the overhead (1.22% and 0.53% on average, respectively).

VIII. RELATED WORK

Two other XOM implementations exist for ARM microcontrollers. uXOM [26] provides XOM for ARM Cortex-M systems by transforming loads into special unprivileged load instructions and configuring the MPU to make the code region unreadable by unprivileged loads. uXOM similarly transforms stores to protect the memory-mapped MPU configuration

registers. Since some loads and stores do not have unprivileged counterparts, transforming them requires the compiler to insert additional instructions, causing the majority of uXOM's overhead. PicoXOM is more efficient in both performance (0.33% compared to uXOM's 7.3%) and code size (5.89% compared to uXOM's 15.7%) as no such transformation is needed. A trade-off for PicoXOM is the code size limit on some ARMv7-M devices; we envision no such limit on ARMv8-M. PCROP [38] is a programmable feature of the flash memory which prevents the flash memory from being read out and modified by application code but still allows code in the flash memory to execute. However, PCROP is only available on some STMicroelectronics devices and cannot be used for other types of memory. In contrast, PicoXOM relies on the MPU and DWT features [4], [5] which can be found on most conforming devices and can protect code stored in any type of memory.

Hardware-assisted XOM has been explored on other architectures. The AArch64 [6] and RISC-V [34] page tables natively support XO permissions. NORAX [11] enables XOM for commercial-off-the-shelf binaries on AArch64 that have constant islands using static binary instrumentation and runtime monitoring. Various approaches [9], [14], [18]–[20], [44] leverage features of the MMU on Intel x86 processors [22] to implement XOM. None of these approaches are applicable on ARM embedded devices lacking an MMU. Lie et al. [28] proposed an architecture with memory encryption to mimic XOM, but it only provides probabilistic guarantees and cannot be directly applied to current embedded systems. Compared to solutions for systems lacking native hardware XOM support, PicoXOM is faster as it has nearly no overhead.

Software can emulate XOM. XnR [7] maintains a sliding window of currently executing code pages and keeps only these pages accessible. It still allows read accesses to a subset of code pages and may incur higher overhead for a smaller sliding window size due to frequent page permission changes. LR^2 [8] and kR^X [33] instrument all load instructions to prevent them from reading the code segment. While these software XOM approaches can generally be ported to embedded devices, they can be bypassed by attacker-manipulated control flow and are less efficient than hardware-assisted XOM [26].

There are also methods of hardening embedded systems. Early versions of SAFECode [15] enforced spatial and temporal memory safety on embedded applications, and nesCheck [29] uses static analysis to build spatial memory safety for simple nesC [17] applications running on TinyOS [21]. PicoXOM enforces weaker protection than memory safety but supports arbitrary C programs (unlike SAFECode and nesCheck) and does not rely on heavy static analysis like nesCheck. RECFISH [43], μRAI [2], and Silhouette [45] mitigate control-flow hijacking attacks on embedded systems. They protect forward-edge control flow using coarse-grained CFI [1] and backward-edge control flow by using either a protected shadow stack [10] or a return address encoding mechanism. EPOXY [13] randomizes the order of functions and the location of a modified safe stack from CPI [25] to resist control-flow hijacking attacks on bare-metal microcontrollers. These systems do not enforce XOM and are still vulnerable to forward-edge corruptions; they can incorporate PicoXOM's techniques to mitigate forward-edge attacks with negligible additional overhead.

IX. Conclusions and Future Work

This paper presented PicoXOM: a fast and novel XOM system for ARMv7-M and ARMv8-M devices which leverages ARM's MPU and DWT unit. PicoXOM incurs an average performance overhead of 0.33% and an average code size overhead of 5.89% on the BEEBS and CoreMark-Pro benchmark suites and five real-world applications.

In future work, we will investigate techniques to ensure that randomization techniques utilizing PicoXOM are effective against brute-force attacks. Embedded systems have limited code placement options for code layout randomization, motivating us to investigate whether the entropy is sufficient and develop techniques to strengthen code randomization if necessary. We will also explore how to leverage debug support like DWT to enforce other security policies with low overhead.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This work was funded by ONR Award N00014-17-1-2996.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. Alexandria, VA: ACM, 2005, pp. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[2] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Proceedings of the 2020 Network and Distributed System Security Symposium*, ser. NDSS '20. San Diego, CA: Internet Society, 2020. [Online]. Available: https://doi.org/10.14722/ndss.2020.24016

[3] Amazon Web Services, Inc. FreeRTOS: Real-time operating system for microcontrollers. [Online]. Available: https://aws.amazon.com/freertos

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

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

[6] *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile*, Arm Holdings, March 2020, DDI 0487F.b.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*, ser. CCS '14. Scottsdale, AZ: ACM, 2014, pp. 1342–1353. [Online]. Available: https://doi.org/10.1145/2660267.2660378

[8] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 2016 Network and Distributed System Security Symposium*, ser. NDSS '16. San Diego, CA: Internet Society, 2016. [Online]. Available: https://doi.org/10.14722/ndss.2016.23364

[9] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "ExOShim: Preventing memory disclosure using execute-only kernel code," in *Proceedings of the 11th International Conference on Cyber Warfare and Security*, ser. ICCWS '16. Boston, MA: ACPI, 2016, pp. 56–64.

[10] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, ser. SP '19. San Francisco, CA: IEEE Computer Society, 2019, pp. 985–999. [Online]. Available: https://doi.org/10.1109/SP.2019.00076

[11] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen, "NORAX: Enabling execute-only memory for COTS binaries on AArch64," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, ser. SP '17. San Jose, CA: IEEE Computer Society, 2017, pp. 304–319. [Online]. Available: https://doi.org/10.1109/SP.2017.30

[12] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," in *Proceedings of the 27th USENIX Security Symposium*, ser. Security '18. Baltimore, MD: USENIX Association, 2018, pp. 65–82. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/clements

[13] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy*, ser. SP '17. San Jose, CA: IEEE Computer Society, 2017, pp. 289–303. [Online]. Available: https://doi.org/10.1109/SP.2017.37

[14] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. San Jose, CA: IEEE Computer Society, 2015, pp. 763–780. [Online]. Available: https://doi.org/10.1109/SP.2015.52

[15] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without garbage collection for embedded applications," *ACM Transactions in Embedded Computing Systems*, vol. 4, no. 1, pp. 73–111, February 2005. [Online]. Available: https://doi.org/10.1145/1053271.1053275

[16] EEMBC. CoreMark-Pro: An EEMBC benchmark. [Online]. Available: https://www.eembc.org/coremark-pro

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. San Diego, CA: ACM, 2003, pp. 1–11. [Online]. Available: https://doi.org/10.1145/781131.781133

[18] J. Gionta, W. Enck, and P. Larsen, "Preventing kernel code-reuse attacks through disclosure resistant code diversification," in *Proceedings of the 2016 IEEE Conference on Communications and Network Security*, ser. CNS '16. Philadelphia, PA: IEEE, 2016. [Online]. Available: https://doi.org/10.1109/CNS.2016.7860485

[19] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '15. San Antonio, TX: ACM, 2015, pp. 325–336. [Online]. Available: https://doi.org/10.1145/2699026.2699107

[20] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "IskiOS: Lightweight defense against kernel-level code-reuse attacks," *arXiv preprint arXiv:1903.04654*, March 2019. [Online]. Available: https://arxiv.org/abs/1903.04654

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '00. Cambridge, MA: ACM, 2000, pp. 93–104. [Online]. Available: https://doi.org/10.1145/378993.379006

[22] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, May 2020, Order Number: 325462-072US.

[23] Y. Jin, G. Hernandez, and D. Buentello, "Smart Nest Thermostat: A smart spy in your home," in *Black Hat USA*, 2014.

[24] D. E. Kouicem, A. Bouabdallah, and H. Lakhlef, "Internet of Things security: A top-down survey," *Computer Networks*, vol. 141, pp. 199–221, August 2018. [Online]. Available: https://doi.org/10.1016/j.comnet.2018.03.012

[25] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '14. Broomfield, CO: USENIX Association, 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[26] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uXOM: Efficient execute-only memory on ARM Cortex-M," in *Proceedings of the 28th USENIX Security Symposium*, ser. Security '19. Santa Clara, CA: USENIX Association, 2019, pp. 231–247. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/kwon

[27] C. Lattner and V. Adve, "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*, ser. CGO '04. Palo Alto, CA: IEEE Computer Society, 2004. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[28] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '00. Cambridge, MA: ACM, 2000, pp. 168–177. [Online]. Available: https://doi.org/10.1145/378993.379237

[29] D. Midi, M. Payer, and E. Bertino, "Memory safety for embedded devices with nesCheck," in *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, ser. ASIACCS '17. Abu Dhabi, United Arab Emirates: ACM, 2017, pp. 127–139. [Online]. Available: https://doi.org/10.1145/3052973.3053014

[30] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," *arXiv preprint arXiv:1308.5174*, August 2013. [Online]. Available: https://arxiv.org/abs/1308.5174

[31] PaX Team. (2000) Non-executable pages design & implementation. [Online]. Available: https://pax.grsecurity.net/docs/noexec.txt

[32] ——. (2001) Address space layout randomization. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[33] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR^X: Comprehensive kernel protection against just-in-time code reuse," in *Proceedings of the 12th European Conference on Computer Systems*, ser. EuroSys '17. Belgrade, Serbia: ACM, 2017, pp. 420–436. [Online]. Available: https://doi.org/10.1145/3064176.3064216

[34] *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, RISC-V Foundation, June 2019, Document Version 20190608.

[35] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 2:1–2:34, March 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[36] A.-R. Sadeghi, C. Wachsmann, and M. Waidner, "Security and privacy challenges in industrial Internet of Things," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. San Francisco, CA: ACM, 2015, pp. 54:1–54:6. [Online]. Available: https://doi.org/10.1145/2744769.2747942

[37] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "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*, ser. SP '13. San Francisco, CA: IEEE Computer Society, 2013, pp. 574–588. [Online]. Available: https://doi.org/10.1109/SP.2013.45

[38] *AN4701 Application Note: Proprietary Code Read-Out Protection on Microcontrollers of the STM32F4 Series*, STMicroelectronics, November 2016, DocID027893 Rev 3.

[39] *PM0214 Programming Manual: STM32 Cortex®-M4 MCUs and MPUs Programming Manual*, STMicroelectronics, March 2020, PM0214 Rev 10.

[40] *UM1932 User Manual: Discovery Kit with STM32F469NI MCU*, STMicroelectronics, April 2020, UM1932 Rev 3.

[41] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the 2nd European Workshop on System Security*, ser. EuroSec '09. Nuremburg, Germany: ACM, 2009, pp. 1–8. [Online]. Available: https://doi.org/10.1145/1519144.1519145

[42] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID '11. Menlo Park, CA: Springer-Verlag, 2011, pp. 121–141. [Online]. Available: https://doi.org/10.1007/978-3-642-23644-0_7

[43] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *Proceedings of the 31st Euromicro Conference on Real-Time Systems*, ser. ECRTS '19. Stuttgart, Germany: Schloss Dagstuhl–Leibniz-Zentrum füer Informatik, 2019, pp. 2:1–2:24. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECRTS.2019.2

[44] M. Zhang, R. Sahita, and D. Liu, "XOM-Switch: Hiding your code from advanced code reuse attacks in one shot," in *Black Hat Asia*, 2018.

[45] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient protected shadow stacks for embedded systems," in *Proceedings of the 29th USENIX Security Symposium*, ser. Security '20. Boston, MA: USENIX Association, 2020, pp. 1219–1236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhou-jie