μRAI: Securing Embedded Systems with Return Address Integrity

Naif Saleh Almakhdhub
Purdue University and
King Saud University
nalmakhd@purdue.edu

Abraham A. Clements
Sandia National Laboratories
aacleme@sandia.gov

Saurabh Bagchi
Purdue University
sbagchi@purdue.edu

Mathias Payer
EPFL
mathias.payer@nebelwelt.net

***Abstract*—Embedded systems are deployed in security critical environments and have become a prominent target for remote attacks. Microcontroller-based systems (MCUS) are particularly vulnerable due to a combination of limited resources and low level programming which leads to bugs. Since MCUS are often a part of larger systems, vulnerabilities may jeopardize not just the security of the device itself but that of other systems as well. For example, exploiting a WiFi System on Chip (SoC) allows an attacker to hijack the smart phone's application processor.**

Control-flow hijacking targeting the backward edge (e.g., Return-Oriented Programming–ROP) remains a threat for MCUS. Current defenses are either susceptible to ROP-style attacks or require special hardware such as a Trusted Execution Environment (TEE) that is not commonly available on MCUS.

We present μRAI [1], a compiler-based mitigation to *prevent* control-flow hijacking attacks targeting backward edges by enforcing the *Return Address Integrity (RAI)* property on MCUS. μRAI does not require any additional hardware such as TEE, making it applicable to the wide majority of MCUS. To achieve this, μRAI introduces a technique that moves return addresses from writable memory, to readable and executable memory. It repurposes a single general purpose register that is never spilled, and uses it to resolve the correct return location. We evaluate against the different control-flow hijacking attacks scenarios targeting return addresses (e.g., arbitrary write), and demonstrate how μRAI prevents them all. Moreover, our evaluation shows that μRAI enforces its protection with negligible overhead.

I. Introduction

Network connected embedded systems, which include the Internet of Things (IoT), are used in healthcare, industrial IoT, Unmanned Aerial Vehicles (UAVs), and smart-home systems [1]. Although these devices are used in security and privacy critical applications, they are vulnerable to an increasing number of remote attacks. Attacks on these systems have caused some of the largest Distributed Denial-of-Service (DDoS) attacks [2], [3], hijacked the control of UAVs [4], [5], and resulted in power grid blackouts [6] among others.

A significant, yet particularly vulnerable portion of embedded devices are microcontroller-based embedded systems (MCUS). MCUS run a single binary image either as bare-metal (i.e., with no OS), or are coupled with a light-weight OS (e.g., Mbed-OS or FreeRTOS [7], [8]). Existing solutions to protect MCUS [9]–[20], are still not deployed as they either require special hardware extensions, incur high overhead, or have limited security guarantees. So far, deployed MCUS lack essential protections that are available in their desktop counterparts [16]–[18], such as Data Execution Prevention (DEP), stack canaries [21], and Address Space Layout Randomization (ASLR). More importantly, vulnerabilities of these systems are not confined to the device itself, but can be a prominent attack vector to exploit a more powerful system. For example, a WiFi System-on-Chip (SoC) can be used to compromise the main application processor of a smart phone as shown by Google's P0 [22]. These attacks gain arbitrary code execution by hijacking the control-flow of the application.

Control-flow hijacking on MCUS and desktop systems originates from memory safety or type safety violations that corrupt indirect control-flow transfers. This can be through the forward edges (i.e., function pointers, and virtual table pointers) or backward edges (i.e., return addresses). On MCUS, Control-Flow Integrity (CFI) [23] can be applied to protect forward edges as was done in desktop systems [24], [25]. These mechanisms reduce the attack surface of forward edges since the target set of indirect calls for CFI is much smaller on MCUS (i.e., the highest is five in our evaluation). *In contrast, return addresses remain prime attack targets for adversaries on MCUS.* This is because return addresses are plentiful in any application, easier to exploit, and more abundant than forward edges. When DEP is enforced, attackers leverage Return-Oriented Programming (ROP) [26] to launch attacks. ROP is a code reuse attack targeting backward edges, allowing an attacker to perform arbitrary execution. ROP remains a viable attack vector even in presence of other defenses such as stack canaries [20], [21], and randomization [17].

Protecting MCUS from control-flow hijacking attacks targeting backward edges, imposes unique challenges compared to desktop systems. MCUS have constrained resources (i.e., a few MBs Flash and hundreds of KBs RAM) and lack essential features required to enforce standard desktop protections. For example, desktop randomization-based defenses (e.g., ASLR) rely on the *OS* to randomize the location of the stack and code layout for each run of the application . However, MCUS use a single static binary image that is responsible for controlling the application logic, configuring hardware (e.g., setting read, write, and execute permissions), and enforcing security mech-

[1] https://github.com/embedded-sec/uRAI

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24016
www.ndss-symposium.org

anisms. This single binary image—containing the application, all libraries, and a light-weight OS—is *loaded once* onto the device and has a single address space. Changing the stack location for each run is not possible without re-flashing the firmware to the device. Even then, the stack is located in RAM which only has tens to hundreds of KBs of *physical* memory, as opposed to GBs of virtual memory on a desktop system. Thus, an attacker can have at least approximate prior knowledge of the device's physical address space.

While researchers proposed several techniques to improve MCUS security, existing techniques cannot prevent control-flow hijacking attacks on all backward edges unless they incur prohibitive runtime overhead. Current defenses protect from control-flow hijacking through randomization [17], [19], [20], memory isolation [16], [18], [19], or CFI [10], [11]. However, these defenses only reduce the attack surface, but remain bypassable by ROP style attacks [27]–[31]. For example, applying CFI for backward edges limits the attacker's ability to divert the control-flow to an over-approximated target set, but is still vulnerable to control-flow bending style attacks [28]. An alternative approach is to rely on information hiding. However, information hiding based defenses [17], [19], [20] are vulnerable to information disclosure [31], [32] and profiling [33] attacks. Ultimately, the security guarantees of information hiding remain limited by the small amount of memory available on MCUS. For example, randomizing the location of a safe-stack [17], [34] only results in a few bits of entropy. A safe stack protected through Software Fault Isolation (SFI) [35], [36] removes the need for information hiding, but will incur high overhead [37].

Defenses also limit their applicability by requiring special hardware extensions that are not available for the wide majority of MCUS such as a Trusted Execution Environment (TEE) [20]. In order to enforce stronger guarantees to protect return addresses one option is to use a shadow stack [37], [38]. However, a shadow stack requires hardware isolation to protect it from information disclosure [37], [39]. One option is using the Memory Protection Unit (MPU), and thus require a system call to access the protected shadow stack region at each function return [40]. The other option is to rely on a TEE such as ARM's TrustZone [10], [41]. Both will result in high overhead (e.g., 10–500% [10]). More importantly, TEEs are not commonly available on MCUS [20]. The most common architecture currently and for the foreseeable future is ARMv7-M, which does not provide a TEE. Moreover, ARMv7-M is still actively used and deployed in new MCUS designs [42]–[46], requiring protections via software updates [47], [48]. Without such protections, control-flow hijacking attacks such as ROP remain a threat to the vast majority of MCUS.

In order to prevent ROP style attacks against many currently deployed MCUS, a defense must enforce the *Return Address Integrity (RAI)* property without relying on extra hardware (e.g., TEE) or incurring large overhead. The RAI property ensures that return addresses are *never writable except by an authorized instruction*. All control-flow hijacking attacks targeting backward edges require corrupting the return address by overwriting it. Enforcing RAI eliminates all such attacks since return addresses are *never writeable* by an attacker. This is different from existing defenses such as CFI implementations [23], randomization, or stack canaries. These defenses do not enforce RAI since return addresses *remain writable*. Such defenses only *limit the use of a corrupted return address*, and thus remain vulnerable to ROP.

Enforcing the RAI property on MCUS without a TEE is challenging as return addresses reside in writable memory (e.g., pushed to or popped from a stack). This leads to three options to enforce RAI on MCUS and protect return addresses. The first is enforcing SFI or allocating a protected privileged region of return addresses (e.g., shadow stack [40]) for *the entire execution of the application*. However, this requires isolating large parts of memory and results in high runtime overhead (i.e., 5–25% [37], [40], [49]). If SFI is used within the application, it should be limited. An alternative option is to keep return addresses in a reserved register that is never spilled to memory or modified by unauthorized instructions. The reserved register cannot be corrupted directly since it is not memory mapped. However, folding the entire control-flow chain at runtime within a single register is challenging. The final option is to remove the need for return addresses by inlining the entire code (i.e., since code is in R+X memory). However, this will lead to code size explosion and require determining all execution paths statically.

Our Solution: This paper presents μRAI, a mechanism that prevents *all* control-flow hijacking attacks targeting backward edges by enforcing the RAI property on MCUS, with a low runtime overhead. μRAI only requires an MPU and the exclusive use of a general purpose register, both of which are readily available on modern MCUS. μRAI inserts a list of direct jumps in the code region of each function (i.e., in R+X memory), where each jump corresponds to a possible return target (i.e., a call site) for the function. All functions have a finite set of call sites, and thus have a finite set of possible return targets. By adding the set of possible return targets for each function as direct jumps (i.e., in R+X memory, rather than writable stack memory), a function can return by using the correct direct jump according to the program execution.

The key to enforce the RAI property is to resolve the correct return target from the appended list of direct jumps during runtime. At runtime, μRAI provides each function a uniquely encoded ID (e.g., a hash value) each time the function is executed. This ID value is unique and corresponds only to one of the possible return targets. A function returns by reading the provided ID, and executing the direct jump corresponding to the given ID. Intuitively, the unique ID μRAI provides must also be protected by the RAI property (i.e., the ID is only readable), as an attacker can modify the provided ID to divert the execution of the application. Moreover, it must be encoded efficiently without incurring high runtime overhead.

μRAI provides each function with its ID by re-purposing and encoding a general purpose register—hereafter known as the *State Register (SR)*. As shown in Figure 1, the SR is encoded through an XOR chain with hard-coded keys before each call and XORed again with the same hard-coded key after returning from each call to restore its previous value. SR *is a dedicated register to μRAI only and is never spilled*. By our design an adversary can have full knowledge of what the used keys are, yet cannot corrupt the SR.

Moreover, μRAI enforces the RAI property even within the execution context of exception handlers (i.e., system calls

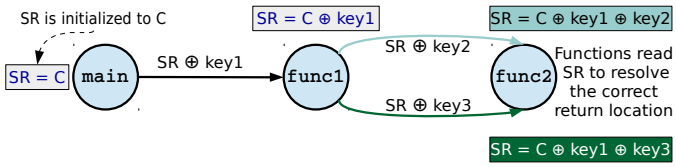

Fig. 1. Illustration of encoding SR through an XOR chain. Arrows indicate a call site in the call graph. SR is XORed each time an edge is walked.

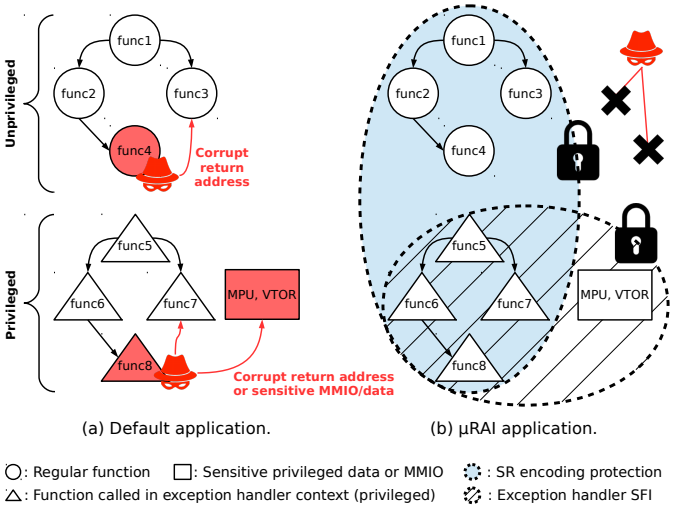

Fig. 2. Illustration μRAI's protections. μRAI prevents exploiting a vulnerable function (e.g., `func8`) to corrupt the return address or disable the MPU in privileged execution by coupling its SR encoding with exception handler SFI.

and interrupts). Exception handlers execute in privileged mode, and can execute asynchronously (i.e., interrupts). As shown in Figure 2(a), enforcing the RAI property for a function called within an exception handler requires more than just protecting return addresses. For example, an attacker can exploit an arbitrary write during an exception to disable the MPU, thus eliminating any defense relying on the MPU (e.g., DEP). To overcome this limitation, μRAI enforces SFI on sensitive privileged Memory Mapped I/O (MMIO) such as the MPU, in addition to encoding SR as shown in Figure 2(b). Enforcing SFI *within an exception handler context only* has negligible overhead since these are only a limited portion of the entire application, unlike applying SFI for the entire application.

As shown in Figure 3, we implement μRAI as an LLVM extension that takes the unprotected firmware and produces a hardened binary that enforces the RAI property. While our prototype focuses on attacks targeting backward edges, we also couple our implementation with a type-based CFI [50]–[52] to demonstrate its compatibility with techniques protecting forward edges. μRAI can ensure its security guarantees and reason about the complete state of application at compile time since it targets MCUS that have a smaller code size compared to their desktop counter parts. We evaluate μRAI on five representative MCUS applications and the CoreMark benchmark [53]. μRAI shows an average overhead of 0.1%. In summary, our contributions are:

1) Return Address Integrity property: We propose the RAI property as a fundamental invariant to protect MCUS against control-flow hijacking attacks targeting backward edges. The RAI property ensures absence of such attacks without requiring special hardware extensions.

2) Exception handler context protection: We enforce the RAI property even for privileged and asynchronous executions of interrupts without special hardware extension by coupling SFI with our SR encoding mechanism.

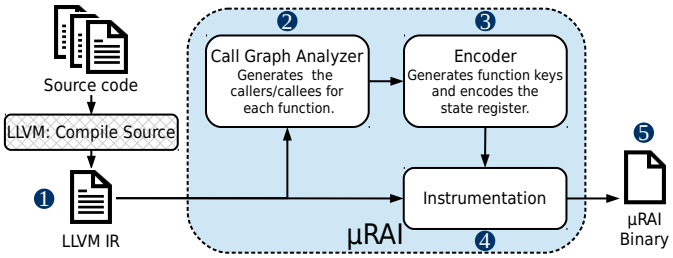

Fig. 3. An overview of μRAI's workflow.

3) μRAI: We design and implement a prototype that enforces the RAI property on MCUS. We evaluate our implementation on CoreMark [53], representative MCUS applications, and proof-of-concept attacks. Our results show that μRAI enforces the RAI property with a runtime overhead of 0.1%.

II. Threat Model

We assume an attacker with arbitrary read and write primitives aiming to hijack the control-flow (e.g., via ROP [26]) of the execution and gain control of the underlying device. Unlike information hiding techniques, we also assume the attacker *knows the code layout*. We target MCUS as our underlying system, which execute a single statically linked binary image. We assume the application is compiled and loaded to the underlying system safely, i.e., the application is buggy, but not malicious. We do not assume the presence of any randomization-based techniques (e.g., ASLR) or stack canaries, due to their shortcomings in our target class of devices as mentioned above. We assume the device has an MPU enforcing DEP (i.e., **W**rite $\oplus$ e**X**ecute) and supports two privilege levels of execution (i.e., privileged and unprivileged).

We complement our prototype with a type-based CFI [50]–[52] to protect forward edges and show our technique is compatible with forward-edge defense mechanisms, however, our focus is on protecting *backward-edges*. The attacker's aim is to corrupt a *backward-edge* to divert control flow. We assume μRAI controls the entire system (i.e., the application is compiled with μRAI). Since we protect against attacks targeting code-pointers, attacks targeting non-control data such as Data-Oriented Programming (DOP) are out of scope [54].

III. Background

MCUS use different architectures with different registers and calling conventions. However, to understand the implementation of μRAI, we focus our discussion on our target architecture, the ARMv7-M [55]. ARMv7-M is applied to Cortex-M (3,4,7) processors, the most widely-deployed processor family for 32-bit MCUS [47], [48].

Memory layout: As shown in Figure 4, ARMv7-M uses a single *physical* address space for all code, data, and peripherals. It uses MMIO to access peripherals and external devices. The memory model defines 4GB (32bit) *physical* address space, however, devices are only equipped with a small portion of it. A high-end Cortex-M4 [56] has only 2MB Flash for its code region, and only 320KB for RAM.

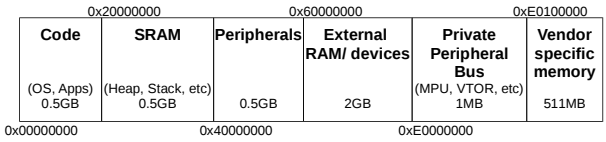

Fig. 4. ARMv7-M memory layout.

TABLE I. A SUMMARY OF CALL INSTRUCTIONS IN ARMV7-M.

Description	Instruction
Direct branch	`b <Label>`
Direct branch and link	`bl <Label>`
Indirect branch	`bx <Register>`
Indirect branch and link	`blx <Register>`

Memory Protection Unit: To enforce access controls (i.e., read, write, and execute) on memory regions, ARMv7-M uses an MPU. Unlike the Memory Management Unit (MMU) present in desktop systems, an MPU does not support virtual addressing, but rather enforces permissions of physical address ranges. Moreover, MPUs only support enforcing a limited number of regions (e.g., eight in ARMv7-M [55]).

Privilege modes: ARMv7-M supports two modes of execution: privileged and user mode. Exception handlers (i.e., interrupts and system calls) are always executed in privileged mode. User mode can execute in privileged mode by executing a Supervisor call (SVC), the ARMv7-M system call. In both privileged and user (i.e., unprivileged) mode, the accessible memory regions can be configured by the MPU. One exception to the MPU configuration is the System Control Block (SCB), which is only accessible in privileged mode and remains writable even if the MPU sets the permissions as read only. The MPU and Vector Table offset Register (VTOR) both reside in the SCB, and thus remain writable in privileged mode.

Core registers: ARMv7-M provides 16 core registers. Registers `R0`–`R12` are general purpose registers. The remaining three registers are special purpose registers. Register 13 is the Stack Pointer (`SP`). Register 14 is the Link Register (`LR`), and register 15 is the Program Counter (`PC`). `LR` is used to store the return address of functions and exception handlers. If a function does not require pushing the return address to the stack (i.e., has no callees), the program can use `LR` to return directly from the function. This method is more efficient than pushing and popping the return address from the stack. Since `LR` is initially reserved for return addresses by ARMv7-M, μRAI uses it as its choice for the state register, SR.

Call instruction types: A call instruction in ARMv7-M has four possible types, shown in Table I. Both direct and indirect calls can automatically update `LR` to hold the return address (i.e., the instruction following the call site) by using any of the branch and link instructions in Table I. Subsequent functions push `LR` on the stack to store the return address in case they use another branch and link instruction to call another function.

IV. DESIGN

μRAI enforces the RAI property by removing the need to spill return addresses to the stack (i.e., writable memory). Instead, μRAI uses *direct jump* instructions in the code region (i.e., R+X only memory) and the SR to determine the correct return location. Both the direct jump instructions and the SR are not writable, and therefore cannot be corrupted by an attacker. This protects against control-flow hijacking attacks that corrupt return addresses (e.g., ROP [57], [58]), even if the code or its layout is completely disclosed to an adversary.

μRAI achieves this by modifying the program at the *function* level. Each function will always have a finite set of possible return targets within the whole application. Such a target set is obtained through analyzing the firmware's call graph [59] statically. At runtime, a function can only have one unique correct return location from the collected target set corresponding to a given call in the control flow. μRAI adds a list of *direct jumps* to the possible return targets as part of the function itself at compile time. At runtime, the unique return location from the list is resolved by using the SR.

A key insight in designing μRAI is that no matter how large the list of possible return targets, μRAI still provides the same security guarantee. This is in contrast to CFI mechanisms [23], where the security guarantees are reduced by over-approximating the valid target set. There is no known method to *statically* compute a fully precise target sets for CFI [23], while dynamic methods [60] require special hardware extensions that are not available on MCUS. Thus, an attacker can perform a control-flow bending style attack by *over-writing* the return address with a return target within the over-approximated target set and divert the control-flow [28], [61]. Unlike CFI implementations, μRAI does *not allow diverting the control-flow*. For μRAI, corrupting the control-flow requires either: (1) over-writing the direct jump μRAI uses to return, which is not possible as the direct jump is in R+X memory; or (2) corrupting the SR. This is again not possible as the SR is never spilled, but only modified through XOR instructions using hard-coded values in R+X memory.

For μRAI, minimizing the possible return targets only affects the memory overhead. μRAI encodes SR with a *unique* value for each possible return target. Each function adds a direct jump corresponding to each unique value of the SR when the function is entered as shown in Figure 5. Over-approximating the target set increases the list direct jump instructions for the function. However, the direct jumps from over-approximation are never executed since, during execution, the SR will never be encoded with their corresponding values. In the following sections, we describe in detail how the SR uniquely encodes each return target.

A. μRAI Terminology

Before discussing the details of μRAI's design, we first define its main components, which are illustrated in Figure 5.

Function Keys (FKs): These are hard-coded values to encode the value of the SR at runtime. The SR is XORed with the key before the call to encode the call location, and after the call to retrieve the previous value of the SR.

Function IDs (FIDs): These are the possible valid encoded values of the SR when entering each function. Each FID value corresponds only to a single return target in the application. A function *cannot* have two FIDs with the same value corresponding to different return locations. The FID values depend on which FKs we embed in the code (i.e., FID = SR $\oplus$ FK).

Function Lookup Table (FLT): FLT is the list of possible FIDs for the function and their respective return targets.

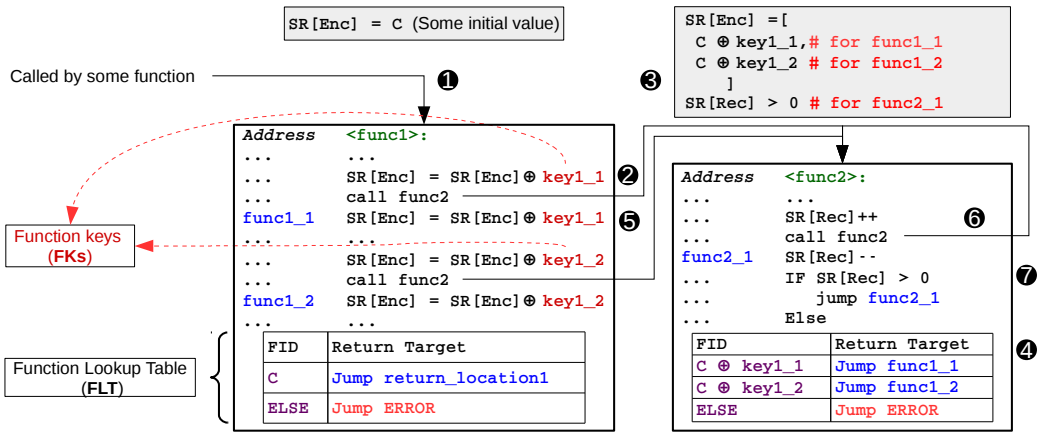

Fig. 5. Illustration of μRAI's design. `Enc`: Encoded SR. `Rec`: Recursive SR.

Target Lookup Routine (TLR): TLR is the policy used to resolve the correct return location. TLR must be designed with care to maintain negligible runtime overhead.

B. Encoding The State Register

A central component in designing μRAI is the use of the SR to enforce the RAI property. As shown in Figure 5, within each function, every FID in the FLT is associated with a direct jump to a return target. At runtime, a function resolves the correct return location by reading the value of the SR, and executing the direct jump where `FID = SR`. At the beginning of the application, μRAI initializes the SR to a known value (i.e., `C` at ❶). For the rest of the application, μRAI dynamically encodes the SR according to two methods.

The first is with an XOR chain at each call site. Before the call site, the SR is XORed with a hard-coded key (❷) to provide each function a list of unique values of the SR (i.e., FIDs), where each FID corresponds to a direct jump to the correct return location (`SR = C` $\oplus$ `key1_1` at ❸). To return from a function, the application reads the current value of the SR and uses the direct jump associated with it (`FID = C` $\oplus$ `key1_1` $\rightarrow$ `Jump func1_1` at ❹). After returning, the SR is XORed again with the same hard-coded key (❺) to restore its previous value (i.e., `C`). The same process is done for the following call sites, and the callee function can resolve the correct return location by *only reading* the value or the SR. For example, if `SR = C` $\oplus$ `key1_2` at ❹, then `func2` was called from the second call site. Thus, `func1_2` is the correct return location and μRAI executes the `Jump func1_2` instruction.

The second use of the SR is a special case when handling recursive functions. Recursive calls may cause a collision in the values of the SR (i.e., FIDs) inside the recursive function. For example, `func2` in Figure 5 is a recursive function. Assume `func2` is called from the first call site in `func1` (i.e., `SR = C` $\oplus$ `key1_1`). Then, `func2` calls itself twice at ❻ (i.e., `SR = C` $\oplus$ `key1_1` $\oplus$ `any_key` $\oplus$ `any_key`), the value of the SR will be `C` $\oplus$ `key1_1`, thus colliding with the existing FID, and `func2` is not able to distinguish between the call at ❷ and ❻. Thus, μRAI reserves some predetermined bits in the SR that serve as a recursion counter. μRAI identifies recursive call sites, and adjusts its instrumentation. Instead of an XOR instruction, μRAI increments the recursion counter bits in the SR before the call (❻). After the returning from the call, μRAI decrements the recursion counter (❼). When the recursion counter reaches zero, the recursive function can return normally using the FLT. Otherwise, it means the function is still in a recursive call, and returns to itself to decrement the recursion counter in the SR. We note that *recursion is generally avoided in MCUS since they have fixed memory, and should only occur with a known maximum bound [17], [40].* μRAI allows bits reserved for the recursion counter to be adaptable according to the underlying application.

Using the SR as a single value however is prone to path-explosion and thus large increases in the FLT. It also cannot handle corner cases for recursive calls (e.g., indirect recursion). μRAI resolves these issues by partitioning the SR.

C. SR Segmentation

To encode the SR, μRAI needs to determine the possible call sites (i.e., return targets) of each function. Thus, the first step in μRAI's workflow (i.e., Figure 3) is to construct the firmware's call graph [59]. μRAI uses the call graph to obtain the possible return targets for each function in the FLT. As mentioned previously in Section IV, over-approximating the possible return targets because of imprecisions in the call graph does not affect the security guarantees of μRAI.

The number of return targets for the function in the call graph provides a minimum limit for the size of the function's FLT. That is, if a function is called from three locations, its FLT can be greater or equal to three, but never less than three. However, the actual FLT (i.e., FIDs) can be larger than the actual possible call sites of a function because of path-explosion. Consider the simple call graph in Figure 6(a), `func3` is called from two locations (i.e., `func1` and `func2`). Ideally, `func3` would have only two possible values of the SR (i.e., FIDs), and thus an FLT size of two. However, the FLT size is four in Figure 6(a), since it can be reached by multiple paths (i.e., two paths from `main`$\rightarrow$`func1` or two paths from `main`$\rightarrow$`func2`). While this does not affect the security guarantees of μRAI, it affects the memory overhead.

To generate efficient FLTs and minimize the effects of path-explosion, μRAI divides the SR into multiple *segments*. As shown in Figure 6(b), the SR is divided into two segments. Each function only uses its specified segment. All functions use segment `Enc1`, while `func3` uses `Enc2`. Thus, when either

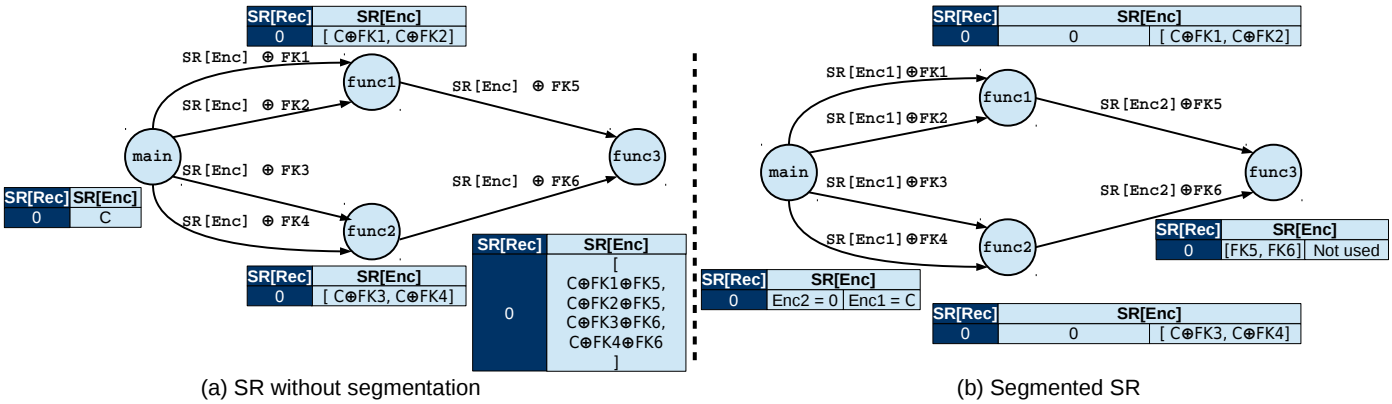

Fig. 6. Illustration of using SR segmentation to reduce path explosion. Segmentation reduced the possible SR values for `func3` by half.

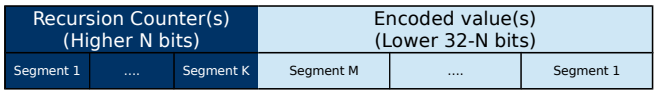

Fig. 7. Illustration segmenting the state register.

`func1` or `func2` call `func3`, only the second segment (i.e., `Enc2`) is encoded. This reduces the size of `func3`'s FLT to two, as opposed to four without segmentation.

Segmenting SR also enables μRAI to resolve the correct return location in case of multiple recursive functions in a call path, as each function can use a segment as a separate recursion counter. In addition, it allows handling other special cases for recursion (e.g., functions with multiple recursive calls). As mentioned in Section IV-B, recursion is rare in MCUS [17], [40]. Since recursion is discouraged on MCUS, we provide the details for handling special cases of recursion in Appendix A.

Figure 7 shows an overview of the SR. Each part can be divided to multiple segments. μRAI automatically enables the number and size of segments to be adaptable depending on the underlying application. In the next sections, we will use the encoded value for our discussion as it is the more general case in MCUS. The concepts however cover both cases.

D. Call Graph Analysis and Encoding

μRAI performs several analyses on the call graph to: (1) calculate the number and size of the SR segments; (2) generate the FKs for each call site to populate the FLTs with its FIDs. To calculate the size and number of segments needed, μRAI uses a pre-defined value of the maximum possible FLT size within an application—which we refer to hereafter as FLT_{Max}. This value can be set by the user or is a limit defined by the underlying architecture. Since each segment in the SR can be equal to or lower than FLT_{Max}, μRAI divides the SR into equal segments each $\log_2(FLT_{Max})$ bits.

To assign each function a segment in the SR, μRAI performs a Depth First Search (DFS) of possible call paths for the application to calculate the FLT size for each function *without segmentation*. When μRAI finishes the DFS analysis, it checks the FLT size for each function. Functions with an FLT size $< FLT_{Max}$ are assigned to the first segment of the SR. Other functions with FLT $\geq FLT_{Max}$ are marked to use the next segment, and DFS is repeated only on marked functions and their callees to calculate their new FLT size when using the second segment. As shown in Figure 6, segmentation reduces the size of the FLT in marked functions (i.e., 50% for `func3`'s FLT). When the DFS analysis completes, the FLT size for each marked functions is rechecked. Marked functions with an FLT size $< FLT_{Max}$ are assigned to the second segment, and other functions are marked for the next iteration of DFS. The analysis is repeated until all functions are assigned to a segment and each function has an FLT size $< FLT_{Max}$.

However, since the number of segments in the SR is ultimately limited, it is possible that some call graphs will require more segments than is available in the SR. Consider the call graph illustrated in Figure 8, both `func10` and `func11` require additional segments in the SR, or an FLT with size $> FLT_{Max}$. To overcome this limitation, μRAI *partitions* the call graph. μRAI instruments calls to these functions with an *inlined* system call to: (1) save the current SR to a *privileged and isolated region*—which we call hereafter as the *safe region*; (2) reset the SR to its initial value. The system call only occurs when calling into `func10` and `func11`, however callees of `func10` and `func11` do not require a system call, and are instrumented normally. When returning to the prior partition, another system call restores the previous SR to enable `func7` and `func8` to return correctly. Thus, μRAI can scale to any call graph, regardless of path explosion. However, it is desirable to minimize such system call transitions in order to maintain a low overhead.

Next, μRAI generates the FKs and populates the FLT for each function with its FIDs. Each FID results from XORing the SR with an FK before the call site to encode the SR. The FID values for each function must be *unique* (i.e., no collisions). Therefore, FKs are chosen to avoid repeating FIDs within each function. However, collisions *are allowed across functions*. For example, if `key1_1` is chosen as zero in Figure 5, `func1` and `func2` can have the same FID value of `C` (i.e., `func2` will have FIDs of `C` and `C` $\oplus$ `key1_2`). These FIDs correspond to different return targets within each function.

To generate the FKs and FIDs efficiently, μRAI uses a modified Breadth First Search (BFS). μRAI records the possible depths of each function in the call graph (e.g., root functions have a depth of zero) from its previous DFS analysis. It traverses each function in the call graph once by ordering functions according to their *maximum call depth*. Starting from root functions (i.e., depth zero), μRAI generates the FKs for each call site such that the result will not cause a collision. Once the FKs for all functions in the current depth level

are generated, μRAI generates the FKs for the next level until all the FKs and FID are generated. Using our method is more efficient than performing DFS again to generate the FKs and FIDs. Applying DFS to generate the FKs and FIDs can cause large delays in compile time, since once a collision occurs, DFS must be performed recursively starting from the violating call site to update all its callees (i.e., until reaching the call graph's leaves). However, using our modified BFS, any collision is directly resolved between the caller and callee functions, without the need for a costly updating process. Since applications on MCUS are smaller in size, our analysis explores the possible states of the call graph.

E. Securing Exception Handlers and The Safe Region

An important aspect for defense mechanisms targeting MCUS is enabling protections for exception handlers. Interrupts execute asynchronously, making their protection more challenging than regular code. An interrupt can execute during any point in the application, thus it is not possible to find a particular caller for an interrupt. However, this makes interrupts, and exception handlers in general appear as root functions in the call graph, since there is no exact call location in the call graph, but rather they execute responding to an external event (e.g., a user pushing a button). Consider Figure 8; root functions other than `main` are exception handlers.

At exception entry, μRAI saves the SR to the *safe region*, and resets the SR to its initial value. Thus, at any time the exception handler executes, it will always have the same SR value (i.e., the initial SR value). Callees of exceptions handlers are then instrumented the same way regular functions are instrumented. At exception exit, μRAI restores the saved SR value from the *safe region* so that code prior to the exception executes correctly, and exits the exception context normally as defined by the underlying architecture.

However, in order for μRAI to enforce the RAI property for exception handlers, it needs to ensure the *safe region* is never corrupted within an exception handler. The safe region resides in a privileged region, and thus cannot be corrupted in user (i.e., unprivileged) mode. However, protecting the safe region during exception handlers (i.e., privileged) requires additional measures since an arbitrary write within an exception handler can access the safe region.

To protect the safe region, μRAI marks exception handler functions and any function that can be called within an exception handler context. μRAI then *masks every store instruction in the marked functions* to enforce SFI [35], [36] of the safe region (e.g., clear most significant bit of the destination). This makes the safe region only accessible at exception entry and exit, which are handled by μRAI. An attacker cannot divert the control-flow to μRAI's exception entry and exit instructions that access the safe region since exception execution is protected by μRAI through the SR and type-based CFI. As shown in Figure 8, functions called within an exception handler amounts to only a limited portion of the functions in the application. This is because interrupts must execute quickly and in fixed time, so that the application can return to the normal execution prior to the interrupt for correct functionality. Enforcing SFI for every store can degrade the performance. However, enforcing SFI of the safe region

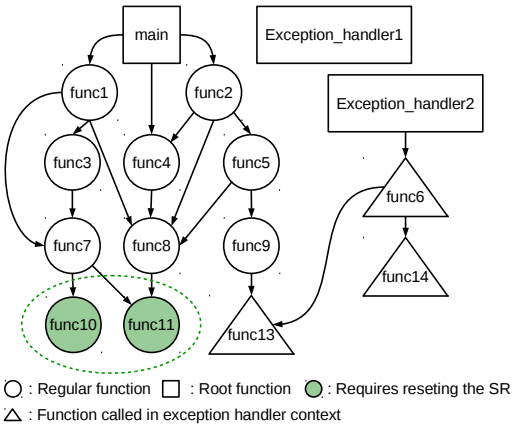

Fig. 8. Conceptual illustration of μRAI's call graph analysis.

for only functions called within an exception handler context, enables an efficient implementation and limit the effect on the runtime overhead since many of the instrumented functions execute for a limited portion of application.

F. Instrumentation

μRAI ensures the RAI property by instrumenting the application in six steps. First, it instruments the call sites with an XOR instruction before the call instruction to encode the SR, and after it to decode its previous value. In the case of recursion, it increments the recursive counter before the call and decrements it afterwards. Second, it reserves the SR so that other instructions previously using the SR will use a different register. Third, it adds the FLT and TLR policy to resolve the correct return target to each function. Fourth, it replaces any return instruction with the direct jumps to the TLR. Fifth, it instruments exception handlers with μRAI's entry and exit routines. Finally, it instruments store instructions for functions callable in exception handler context with masking instruction to protect the *safe region*. As discussed in Section IV-C, path-explosions affects the FLT size depending on the function call sites and its depth in the call graph. Without carefully choosing suitable TLR policy, the performance overhead of resolving the correct return location can become prohibitive.

G. Target Lookup Routine Policy

Enforcing the RAI property is important, however, it is equally important to maintain an acceptable performance overhead [62]. One simple TLR policy to resolve the correct return location is to use a switch statement and compare the value of the SR sequentially to the FID values in the FLT, and return once a match is found. While this policy enforces the RAI property, it has unbounded, and possibly high performance overhead for large FLTs.

An important aspect of μRAI is it ensures low, and deterministic overhead that is independent of the FLT size. Therefore, μRAI uses a *relative-jump* policy to resolve the correct return location using two instructions: (1) the first instruction in the TLR is a relative jump (i.e., `jump PC+SR`); (2) a direct jump to the correct return location. The relative-jump policy uses the SR as an index of a jump table, where the direct jump pointing to the correct return location is at a distance equal to the SR (i.e., FID) from the first relative jump.

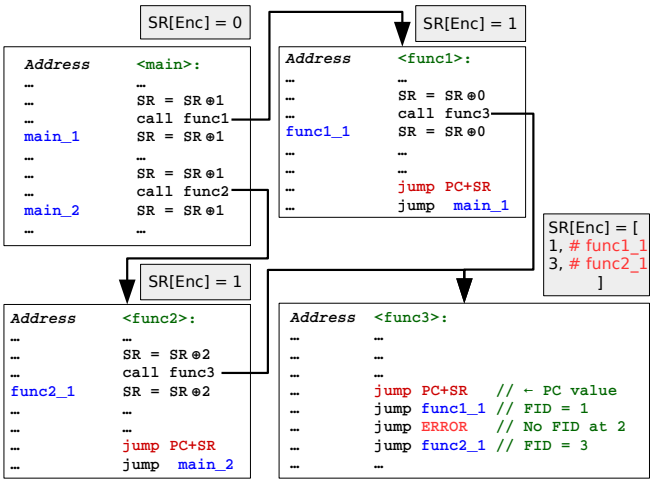

Fig. 9. Illustration μRAI's relative-jump TLR policy.

Both instructions are impossible to modify by an attacker since they reside in R+X memory. In addition, the attacker cannot modify the SR in the first relative jump since it is never spilled. In case of SR segmentation, only the specified segment is used for the first relative jump instruction.

Figure 9 illustrates an example of the relative-jump TLR. Consider `func3` and assume the PC points to the current location and each instruction size is 1—so that PC+1 will lead to the next instruction. If `SR = 3` at `func3`, then `jump PC+SR` will jump to `jump func2_1` which jumps to the correct return location (i.e., after the call in `func2`). We can also conclude that the call path was `main`→ `func2`→ `func3`. For `func3`, there is no $FID = 2$, and thus a `jump ERROR` was placed at index 2. This is needed to ensure the correct return instructions are always at the correct distance from the first relative jump. No matter how large the FLT size for a function is, the performance overhead should be deterministic. However, to minimize the memory overhead, it is better to have the SR as small as possible since the FLT size is equal to the largest possible FID value (e.g., if we have `SR=[1, 1024]` we need to fill the remaining 1022 with jump ERROR). Controlling the SR is done by minimizing the values of the FKs. Thus, at each call site, μRAI chooses the FK that will minimize the maximum FID value.

V. Implementation

μRAI is comprised of four components (see Figure 3). The first component constructs the call graph of the application. The second component uses the call graph to generate the encoding (i.e., FKs and FIDs) for each function. The third component instruments the application with the generated FKs and FIDs. The fourth component is a runtime library that secures saving the SR and restoring it from the safe region. The call graph analysis and instrumentation are implemented as additional passes to LLVM 7.0.1 [63]. The encoder is implemented as a Python script to leverage the graph libraries [64]. We provide a Makefile package that automates the compilation process of all four components. We implement μRAI to enforces the RAI property for the ARMv7-M architecture, enabling it to protect a wide range of deployed MCUS. As the Link Register (`LR`) in normally used to store return addresses, we use it as the SR, and prohibit its use in any other operation.

A. Call Graph Analyzer

The call graph analyzer constructs the call graph of the application, and is implemented as an IR pass in LLVM. For each call site it identifies all possible targets, and for each function it generates a list of callers and callees. For direct calls, the call graph analyzer determines the exact caller and callee relation. While μRAI's primary goal is to protect the backward edges of control-flow transitions (i.e., return addresses), we complement it with a type-based forward-edge CFI to provide an end to end protection against control-flow hijacking along both the forward and backward edge. Thus, the call graph analyzer uses a type-based alias analysis [65] to determine the possible callees from indirect call sites. That is, we identify any function matching the type-signature of any indirect call site within the current function as a valid transition in the call graph. Thus, we generate an over-approximated call graph (see Appendix Section B-A for details). Finally, the call graph analyzer exports the call graph to the encoder, which uses the call graph to generate the FKs and FIDs.

B. Encoder

The encoder generates the hard-coded FKs for each call site and populates the FLT of each function with its FIDs. It first calculates the possible limits of the FLT in the application (i.e., minimum and maximum). These limits are then used to configure and optimize the number and size of segments in the SR. FLT_{Max} in ARMv7-M is 4095 bytes [55]. The minimum limit of the FLT is the log_2 of highest number of return targets for any function. For example, if a function is called from eight locations its FLT_{Min} must use at least eight FIDs. Thus, each SR segment must at least have $\log_2(8) = 3$ bits.

Using both limits, the encoder then searches for the SR segment size configuration that will minimize the memory overhead (i.e., $FLT_{Min} \leq 2^{segment\ size} < FLT_{Max}$). Not all options are possible to support. For example, the SR segment size can be set to 16 bits, but it will require using three segments (i.e., 48 bits). Since registers have only 32 bits, such a solution is not possible without using an additional register to serve as additional segments for the SR. An alternative option is saving the SR to the safe region as discussed in Section IV-D. For μRAI, we limit the SR to only one register to minimize the effect on the system resources, and limit using a system call transition to save the SR in the safe region to only one transition within a call path. However, these are configurable and can be changed by the user. To estimate the memory overhead of the possible configurations for the SR segments, the encoder performs a DFS search for the possible call paths over the call graph to calculate the FLT size for each function, and calculates the total summation of FLTs. The process is repeated using each configuration. The configuration with the least summation of FLT sizes for all the functions will be used since it minimizes the added instructions for the application. The DFS search also assigns a segment for each function in the call graph (Section IV-D).

The encoder uses the chosen SR segment size to: (1) calculate the initial value of `LR`; (2) generate the FKs and FIDs for each function. As discussed in Section IV-D, μRAI uses a modified BFS to generate the FKs. The generated FKs only affect the segment assigned to the callee function at the call

site. FKs must satisfy three conditions. First, the largest value allowed for FKs is $2^{segment\ size}-1$, since any larger value will overflow to the next SR segment. Second, FKs must generate FIDs that will result in an aligned FLT. For ARMv7-M, a direct jump is equivalent to a branch instruction (Section III). Since the size of branch instruction for ARMv7-M is four bytes, each FID in the FLT need be at a distance of four. Third, the generated FIDs must not cause a collision in the callee's FLT. Each generated FID is a result of encoding the SR with an FK at a call site (i.e., $FID = SR_{at\ call\ site} \oplus FK$). Since FIDs cannot repeat within an FLT, the chosen FK must not result in a collision. μRAI searches through the valid FKs, and chooses the FK value that will result in lowest possible FID. A large FID value can result in a sparse FLT (Section IV-G), thus increasing the memory overhead.

C. Instrumentation

The generated encoding is then used to produce the binary enforcing the RAI property. Instrumenting the application is done in three steps: (1) instrument each call site to enable encoding `LR` with the generated FKs; (2) reserve `LR` to be used by μRAI only; (3) add the TLR and FLT generated from the encoder for each function. These are done by modifying and adding a pass to LLVM's ARM backend. In the following we provide a detailed description for each instrumentation step. We refer readers interested in a detailed disassembly of each instrumentation step to Appendix B-B.

μRAI transforms each call site by inserting XOR instructions before and after each call site to encode and decode `LR` (our SR), respectively. In case the call is a recursive call, the instrumentation increments the designated recursion counter segment before the call, and decrements it afterwards. If the call site uses a (`bl` or `blx`) instruction, μRAI transforms it to a (`b` or `bx`) instruction (see Table I).

To ensure `LR` is only used by μRAI, we modify LLVM's backend to reserve `LR` so it cannot be used as general purpose register. Moreover, μRAI transforms any instruction using `LR`, so that it will not use `LR`. For example, transforms `push {R7,LR}` to `push {R7}`.

Finally, μRAI adds the TLR and FLT for each function. Since functions can have multiple return locations, μRAI replaces any return instruction with a trampoline to the beginning of the inserted TLR. The exact TLR depends on the application and whether it uses SR segmentation or not. Many applications for MCUS have a small code size, and can be instrumented without segmenting the SR. For ARMv7-M, a relative jump with `PC` is achieved by using (`ADD PC,<register>`). Thus, μRAI uses it for its first relative jump in TLR with `LR` as the offset for the relative jump (i.e., as `ADD PC,LR`). To add the FLT, μRAI uses direct branches, with each direct branch at a distance equal to its pre-calculated FID. In case of segmentation, μRAI requires three instructions for its TLR. The first instruction copies the function designated segment along with segments in lower order bits from `LR` to `R12`, which is a scratch register in ARMv7-M. Next, μRAI clears any lower order bits that are not part of the function designated segment from `R12`. Thus, only the needed bits that form a segment are in the lower bits of `R12`. This enables using the relative jump instruction as before. Using `R12` with a segmented SR does not affect the security guarantees of μRAI. The value used is only read inline from `LR`, which is not writable by an attacker.

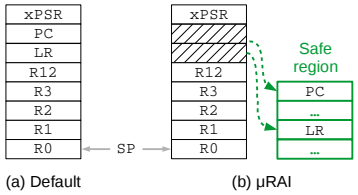

Fig. 10. Illustration of exception stack frame for ARMv7-M.

D. Runtime Library

The runtime library: (1) configures the MPU to enforce DEP at the beginning of the application execution; (2) sets the initial value of `LR` (i.e., the state register); (3) secures transitions of exception handlers entry and exit. At the start of the application, the runtime library initializes `LR` to the defined value by the encoder in Section V-B. In addition, the runtime library configures the MPU to support DEP automatically. The code region is set as readable and executable, while data regions are set readable and writable. The *safe region* is configured as readable and writable in *privileged* mode only to protect it from unprivileged code. Protecting the safe region requires additional mechanisms within exception handlers context, which we describe in Section V-E.

E. Securing Interrupts and System Calls

Securing the execution of exception handlers (i.e., interrupts and systems calls) requires overcoming limitations of the architecture. First, entering and exiting exceptions is handled by the hardware in ARMv-M. When an exception occurs, the underlying hardware pushes the registers from the user mode to the stack. As shown in Figure 10(a), the stack frame includes `PC` and `LR`. The hardware also sets `LR` to a special value called `EXC_RETURN`. To return from the exception, the hardware uses the saved stack frame it pushed when entering the exception and loads `EXC_RETURN` to `PC` or executes a `bx` instructions with a register holding the `EXC_RETURN` value. While we can still use our TLR for the rest of the exception handler execution, the restriction of using `EXC_RETURN` to exit exception handlers prohibits using our TLR to exit exception handlers. Thus, μRAI instruments interrupt handlers entry and exit with special routines that moves the pushed values of `PC` and `LR` from the stack to the safe region, as shown in Figure 10(b). It also moves the special value stored in `LR` to the safe region. It then clears the locations of `PC` and `LR` from the exception stack frame.

Since exception handlers are root functions in the call graph, the runtime library sets `LR` to the initial value specified by the encoder. Functions are then instrumented using the regular TLR and FLT instrumentation. When an exception handler exits, μRAI restores the previously saved values of `PC` and `LR` from the safe region to their location on the stack frame. It also sets `LR` to the special value required by the hardware. Thus, enabling the exception to exit correctly.

This instrumentation enables correct execution, but alone fails to enforce the RAI property. As exception handlers execute in *privileged* mode, an attacker can corrupt the saved `PC` or `LR` in the safe region to force the exception exit to return to

a different location. Alternatively, an attacker can relocate the vector table by overwriting the VTOR (see Section III). Finally, an attacker can disable the MPU or alter its permissions to allow code injection. Simply setting VTOR and MPU as read only is not effective and the MPU registers remain writable within exception handlers. We verified this in our experiments.

To protect the above resources against these attacks, μRAI applies SFI only *to store instructions that can execute within exception handler context*. The MPU registers are mapped within a contiguous address range (i.e., 16 bytes), while VTOR is mapped separately. For the MPU and VTOR, we verify that the destination does not point to within the MPU address range or VTOR. To protect the safe region, μRAI places the safe region in a separate region. One option is to protect the safe region the same way as the MPU. A more efficient approach is to leverage Core Coupled RAM (CCRAM), which starts at a different address and is smaller than normal RAM (e.g., 64KB compared to 320KB RAM in our board [56]). Placing the safe region in CCRAM enables efficient protection through *bit-masking the destination* of store instructions to ensure it does not point to the safe region [35]. For our evaluation, we leverage this more efficient bit-masking approach. See Appendix B-A (i.e., Listing 5) for a detailed disassembly.

Our exception handler SFI routine can degrade performance if instrumented for every store instruction in the application. However, we only instrument store instructions for functions that can be called within an exception handler context. These are a small fraction of the entire application, and thus limits the effect of of the verification routine. Furthermore, some exception handlers (i.e., `SysTick`, which increments a fixed global) do not require the SFI instrumentation since the store performed is always to a fixed address. Similarly, store instructions using `SP` for its destination are not instrumented with SFI since instructions assigning `SP` use a bit-masking instruction to ensure it points to the stack region. Coupling the SR encoding with exception handler SFI enforces the RAI property for exception handlers.

VI. Evaluation

Our evaluation aims to answer the following questions:

1) Can μRAI fully prevent control-flow hijacking attacks targeting backward-edges?
2) What are the security benefits compared to CFI?
3) What is the performance overhead of μRAI?
4) What is the memory overhead of μRAI?

We evaluate the effectiveness of μRAI using five representative MCUS applications (PinLock, FatFs-RAM, FatFs-uSD, LCD-uSD, and Animation) and the CoreMark benchmark [53]. *PinLock* demonstrates a smart-lock receiving a user entered pin over a serial port. The pin is hashed and compared against a precomputed (i.e., correct) hash. If the comparison succeeds, the application sends an IO signal to mimic opening the lock. Otherwise, the door remains locked. *FatFs-uSD* demonstrates a FAT file system on an SD card, while *FatFs-RAM* mounts the file system in the device's RAM. Both applications perform similar functionality (i.e., accessing the FAT file system) however their internals (e.g., Hardware Abstract Layer libraries) are different. *LCD-uSD* reads multiple bitmap pictures from an SD card and displays them on the LCD display. *Animation* demonstrates the animations effect on the LCD by displaying multiple layers of bitmap images. All application are provided by STMicroelectronics except PinLock, thus they represent realistic applications used for deployed MCUS. CoreMark is a standardized benchmark developed by EEMBC [53] to measure MCUS performance. The evaluation is performed using the STM32f479-EVAL [56] board and includes the cost of type-CFI.

A. Security Analysis

In order to evaluate μRAI's protections, we implement three control-flow hijacking attacks on backward edges. The goal of these experiments is *not* to investigate whether μRAI can protect from certain attack cases such as [66]–[68], but rather to demonstrate μRAI's ability to prevent *any* control-flow hijacking attack targeting backward edges even in the presence of memory corruption vulnerabilities.

Control-flow hijacking attacks targeting backward edges must start from one of three types of memory corruption vulnerabilities. *First*, a buffer overflow [69], where an attacker leverages this vulnerability to overwrite the return address with the attackers desired value. However, the attacker also corrupts all sequential memory locations between the vulnerable buffer and the return address. *Second*, an arbitrary write (e.g., format string [70]), where the attacker directly over-writes the return address, without needing to corrupt other memory locations. *Third*, a stack pivot [71], where instead of over-writing the return address, the attacker controls the stack pointer in this scenario. To launch the attack, the attacker sets the value of the stack pointer to a buffer controlled by the attacker. Thus, when the application pops the return address from the stack, it will pop the value from the attacker controlled buffer.

Our experiments demonstrate the three types of attacks based on the *PinLock* application. We assume these vulnerabilities exist in the application in the function receiving the pin from the user, namely `rx_from_uart`. A successful attack uses the underlying vulnerability to directly execute the `unlock` function to unlock the smart-lock without entering the correct pin. As discussed in Section II, we assume the attacker is aware of the entire code layout.

Buffer overflow: This attack assumes the return address is available in a sequentially writable memory from the vulnerable buffer (e.g., on the stack). However, μRAI uses R+X memory in Flash and an inaccessible SR. Both are not modifiable and the attacker cannot modify the instructions that update the SR. The vulnerability here only corrupts data available on the stack, but the return address is not affected. The control flow is not diverted and μRAI successfully prevents the attack.

Arbitrary write: While the attacker is capable of writing to any available memory for user code, such a vulnerability cannot be used to launch a successful attack. The attacker cannot write directly to the SR (i.e., `LR` register) since it is not memory mapped. Furthermore, the attacker cannot use μRAI's return sequence in Listing 2 or Listing 3, as these only read the SR and never write to it. Modifying the instructions is also not possible as the MPU configures them as only readable and executable. A final option is to corrupt the saved `PC` or `LR` in the safe region from an interrupt context entry in order to divert the return from the interrupt. When the attack is attempted in

TABLE II. ANALYSIS OF THE TARGET SET SIZES FOR BACKWARD EDGE TYPE-BASED CFI.

App	Type-based CFI Target Set			
	Min.	Median	Max.	Ave.
PinLock	1	2	8	3
FatFs_uSD	1	6	94	21
FatFs_RAM	1	5	94	27
LCD_uSD	1	5	49	11
Animation	1	4	49	11
CoreMark	1	3	52	12

unprivileged mode, it causes a fault since the safe region is protected by the MPU. If the attack occurs during privileged execution, the safe region is protected through our exception handler SFI mechanism. Thus, μRAI prevents the attack.

Stack Pivot: Even when the attacker changes the stack pointer, this attack relies on popping the return address from the stack. Since μRAI only uses the SR and the instructions in Listing 2 or Listing 3, the attacker controlled buffer can corrupt the function's data, but is never used to return from the function. As a results, μRAI successfully prevents control-flow hijacking through stack pivoting. Note that μRAI does *not* prevent stack pivoting from occurring, but prevents using a stack pivot to corrupt the return addresses.

B. Comparison to Backward-edge CFI

To understand the benefits of μRAI's protections, we analyze the possible attack surface compared to an alternative backward-edge CFI mechanism. With such a mechanism, the function can return to only specific locations in the application. These locations define the function's *target set*. The target set is comprised of the set of possible return sites for each function, enumerating the addresses of all instructions immediately after a function call. For example, if function `foo` is called from three different locations, the three instructions right after the return from the function call are in the target set for `foo`. For indirect calls, we identify any call site matching the function type signature of any indirect call within the current function to be a possible call site [50], [52], [61]. We build our prototype on top of ACES' [16] type-based CFI as it provides a more precise target set than other existing work for MCUS. Intuitively, the chance of a control-flow bending style attack [28] increases as the function target set size increases. That is, an attacker can still divert the control-flow to any location within the target set.

Table II shows the minimum, median, maximum, and average target set sizes for the functions within each application. Many applications share the same libraries and Hardware Abstraction Layers (HALs). As these are called most frequently, the worst case scenario for the applications (i.e., maximum target set size) can be shared between applications sharing these libraries of HALs (e.g., FatFs-uSD and FatFs-RAM). Averaged across all the applications in Table II, a backward edge CFI will have an average target set of 14 possible return locations. However, the effect of imprecision on CFI are clearer when considering the maximum target set for each application in Table II. Averaged across all applications, an attacker will have a target set of 58 possible return locations. In contrast to existing CFI implementation, μRAI eliminates this remaining attack surface since it does not allow corrupting the return address, rather than focusing on minimizing the target set, which is ultimately limited to imprecisions [23].

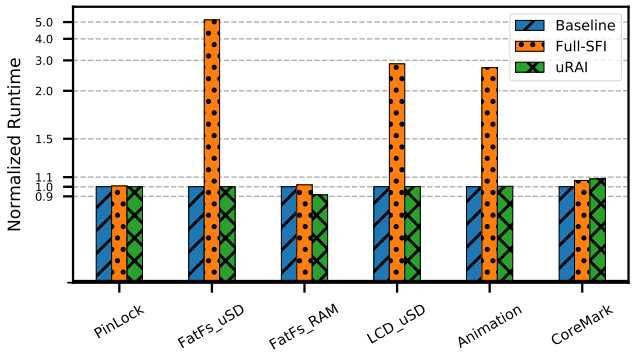

Fig. 11. Comparison of runtime overhead for μRAI, Full-SFI, and baseline.

C. Runtime Overhead

For defense mechanisms to be deployable, they must result in low performance overhead [62]. This is highly relevant for MCUS, where they can have a real-time constraint as well. To evaluate the performance overhead, we modify the applications to start the runtime measurement at the beginning of `main` and stop at the end of the application. For PinLock, we stop the application after receiving 1000 pins that alternate between incorrect pins, a correct pin, and a locking command that requests the pin again. For CoreMark, we used its own performance reporting mechanism to collect the measurement. The results are averaged across 20 runs.

Figure 11 compares the performance of the baseline, μRAI, and applying SFI to all store instructions in the application—which we denote *full-SFI*. μRAI results in an average overhead of 0.1%, with the highest overhead for CoreMark with 8.1%. μRAI shows an improvement of 8.5% for FatFs_RAM. This is not an inherent feature of μRAI but an effect of changing code layout and register usage (reserving the `LR` register) in a hot loop in the application. Particularly, the baseline calls `__aeabi_memclr` eight times to clear 64 bytes during each iteration. For μRAI, the compiler optimized this to one call to clear 512 bytes at each iteration. To confirm this, we evaluated an intermediate binary that uses the compiler changes without applying any instrumentation. The optimization appeared and the intermediate binary showed an improvement of 14.4%. Compared to this intermediate binary, μRAI has an overhead of 6.9%. Considering this effect μRAI yields an average overhead of 2.6%. In other applications, no improvement in runtime was observed between the baseline and intermediate binaries. μRAI is efficient since it only adds three to five cycles per call-return (see Table VI in Appendix B-B for details). Return instructions are not a large part of the application, thus μRAI yields a low overhead.

An alternative to μRAI is to apply a safe stack. Safe stack only prevents buffer overflow attacks. To *prevent* other attack vectors (e.g., arbitrary write), a safe stack must be coupled with SFI since information hiding offer limited guarantees on MCUS. We use full-SFI to mimic protecting the safe stack by instrumenting all store instructions with a single bit-masking instruction except ones using `SP` (i.e., we assume `SP` is verified at the point of assignment instead). The average overhead for full-SFI was 130.5%. In contrast to full-SFI, μRAI remains efficient since it limits SFI to functions that can be called within an exception handler context, which are a small portion of the application. Table III shows both the

TABLE III. SUMMARY OF EXCEPTION HANDLER SFI PROTECTION FOR STORE INSTRUCTIONS. % SHOWS THE PERCENTAGE OF STATICALLY PROTECTED INSTRUCTIONS W.R.T THE TOTAL BASELINE INSTRUCTIONS.

App	# of Store instruction			
	Static	Total	%	Dynamic
PinLock	56	516	10.9	7
FatFs_uSD	99	1,802	5.5	906K
FatFs_RAM	7	1,116	0.6	7
LCD_uSD	99	2,814	3.5	48K
Animation	99	2,760	3.6	66K
CoreMark	56	1,024	5.5	7

TABLE IV. SUMMARY OF μRAI'S ENCODER FLT AND SR SEGMENT CONFIGURATION COMPARED TO FLT_{Min} OF EACH APPLICATION.

Application	FLT_{Min}	$FLT_{\mu RAI}$	SR Segment Size (bits)
PinLock	8	8	5
FatFs_uSD	94	128	9
FatFs_RAM	94	128	9
LCD_uSD	49	64	8
Animation	49	64	8
CoreMark	52	64	8

number of instrumented store instructions both statically and dynamically. On average, μRAI statically instruments only 4.9% of *all* store instructions in the baseline firmware.

D. FLT Encoding Analysis

A central component of μRAI is its encoder (see Section V-B) and how efficiently it configures and populates the FLTs to reduce the effects of path explosion on the memory overhead. As discussed in Section V-B, the encoder searches the possible FLT sizes between FLT_{Min} (i.e., the function with the highest number of call sites in the application) and FLT_{Max} (i.e., the highest possible FLT as defined by the architecture) and chooses the configuration that will provide the lowest possible memory overhead. Intuitively, the closer the encoder's FLT is to FLT_{Min}, the lower the memory overhead is due to FLTs, since a larger FLT indicates FID collisions in the FLT due to path explosion. Thus, to evaluate our encoder, we compare its configured FLT size and SR segment size to the application FLT_{Min}. Table IV shows FLT_{Min} and μRAI's configured FLT (i.e., $FLT_{\mu RAI}$). μRAI's FLT can only be at powers of two since it partitions the SR's bits into several segments, where each segment is $\log_2(FLT_{\mu RAI}) + 2$. The additional two bits are because the size of each FID in the FLT is a four byte branch instruction. μRAI consistently chooses the closest power of two to FLT_{Min}, and thus it is close to the best possible FLT configuration.

A key mechanism for μRAI's encoder to achieve these results is partitioning the SR into several segments where each function only uses a designated segment as discussed in Section IV-C. To demonstrate this effect, we show the FLT sizes both with and without segmentation in Table V. Averaged across all applications, segmentation reduces FLT sizes by 78.1%. Intuitively, PinLock can be instrumented without segmentation. As mentioned in Section V-C, many MCUS applications use small code size and thus can be instrumented without segmentation. However, it is segmented by μRAI since segmentation will result in lower memory overhead. For further analysis of encoder efficiency, see Appendix B-B.

E. Memory Overhead

μRAI requires adding the instrumentation for encoding and decoding the SR at each call site, adding the FLTs, instrumenting exception handler SFI, and using its runtime library. In addition, we couple μRAI with a type-based CFI for forward edges. This however increases the total utilized memory. Figure 12(a) shows the overhead of μRAI in RAM. For LCD-uSD and animation applications, μRAI incurs negligible overhead. This is expected since the majority of μRAI's instrumentation utilizes Flash. Averaged across all applications, μRAI shows an increase of 15.2% for RAM.

TABLE V. SUMMARY OF THE SEGMENTATION EFFECT ON FLT SIZE.

App	Without Segmentation			Segmented			Ave. Reduction
	Min.	Max.	Ave.	Min.	Max.	Ave.	
PinLock	1	12	3	1	8	2	33.3%
FatFs_uSD	1	8,650	699	1	106	21	97.0%
FatFs_RAM	1	632	86	1	105	20	76.7%
LCD_uSD	1	11,898	727	1	59	12	98.3%
Animation	1	11,570	683	1	59	12	98.2%
CoreMark	1	352	23	1	52	8	65.2%

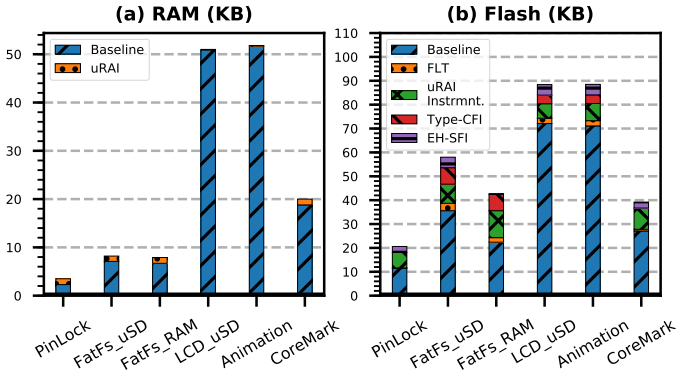

Fig. 12. Illustration of μRAI's memory overhead.

The Flash increase of μRAI's instrumentation, exception handler SFI (EH-SFI), and type-based CFI is shown in Figure 12(b). The majority of μRAI's instrumentation occurs in Flash, thus it is expected for μRAI to have a higher overhead for Flash than for RAM. Averaged across all applications, μRAI shows an overhead of 34.6% for its instrumentation and FLT, and 9.5% for EH-SFI. Our type-based CFI implementation shows an average increase of 10%. Combined, the average is 54.1% for Flash. μRAI adds at most 22.4KB (i.e., Fats_uSD). The increase is large for small applications (e.g., PinLock), as any change can drastically affect their size. However, μRAI performs better for larger application (e.g., 22.7% for LCD_uSD). That is, μRAI overhead does not grow as the application size increases. We note that Flash is available in larger sizes (i.e., MBs) than RAM (i.e., hundreds of KBs).

VII. RELATED WORK

Vast related work exists in the area of control-flow hijacking attacks and defenses. However, not all are applicable to MCUS. Thus, our discussion focuses on related work that targets MCUS or is applicable to it. We refer to the relevant surveys [23], [26], [62], [72]–[75] for readers interested in the general area of control-flow hijacking attacks and defenses.

Remote attestation: Remote attestation use a challenge-response protocol to establish the authenticity and integrity of MCUS to a trusted server (i.e., verifier). Remote attestation requires using a *trust anchor* on the prover (e.g., MCUS) to respond the verifier's challenge. C-FLAT [11] attests the integrity of the control-flow by calculating a hash chain of

the executed control-flow. LiteHAX [15] attests the integrity of both the control-flow and the data-flow. DIAT enables on-device attestation by relying on security architectures that provide isolated trusted modules for the program. Overall, remote attestation defenses require additional hardware (e.g., TEE or additional processor). In addition, they only *detect* the occurrence of an attack. μRAI *prevents* control-flow hijacking on backward edges without requiring hardware extensions.

Memory isolation: Minion [18] enables partitioning the firmware into isolated compartments on a per-thread-level, thus limiting a memory corruption vulnerability to affect a single compartment and not the entire system. ACES [16] automates compartmentalizing the firmware on a finer-grained intra-thread level. TyTan [13] and TrustLite [14] enforce memory isolation through hardware extensions. Memory isolation techniques can enable integrity and confidentiality between different compartments. However, they only confine the attack surface to a part of the firmware, while μRAI focuses on preventing ROP style attacks against the entire firmware.

Information hiding: LR^2 [76] uses SFI-based execute only memory (XoM) and randomization to hide the location of code pointers. However, its implementation is inefficient on MCUS as was shown by μXOM [19] which enables efficient implementation of XoM for Cortex-M processors. EPOXY [17] uses a modified and randomized location of a SafeStack [34] to protect return addresses against buffer overflow style attacks. μArmor protects from the same attack using stack canaries. Both EPOXY and μArmor enforce essential defenses for MCUS efficiently and apply code randomization to hinder ROP attacks and produce different randomized binaries per device to probabilistically mitigate scaling such attacks to a large number of devices. In general, information hiding techniques remain bypassable and do not prevent attacks with an arbitrary write primitive. μRAI however enforces the RAI property to prevent ROP, and extends its protections to exception handlers.

CFI protections: SCFP [12] uses a hardware extension between the CPU's fetch and decode stage to enforce control-flow integrity and confidentiality of the firmware. SCFP only mitigates attacks on backward edges (i.e., it does not prevent control-flow bending style attacks [28]). CFI-CaRE [10] enforces CFI on forward-edges and a hardware isolated shadow-stack to protect the backward edges. CFI-CaRE provides strong protections against ROP style attacks, however, it requires using a TrustZone, thus is not usable by a wide range of MCUS. RECFISH [40] applies CFI and a shadow stack to MCUS binaries, without requiring source code. However, it places shadow stack in a privileged region, thus requiring a system call to return from a function. Thus, both CFI-CaRE and RECFISH incur a high overhead (e.g., 10-500% [10], [40]). μRAI enforces the RAI property without requiring a TEE and with a modest runtime overhead.

VIII. Discussion

Protecting privileged user code: Protecting sensitive resources (e.g., MPU) require the confinement of store instructions within a privileged execution of user code, where the developer provides privileges for restricted operations for the given application. Identifying these restricted operations automatically is non-trivial since they are application specific as shown by previous work [17]. For our evaluation, these operation occur during initialization. An attacker cannot divert the control-flow to these operations again (i.e., since μRAI enforces the RAI property and type-based CFI). However, to enable flexible use of μRAI we enable developers to apply our SFI mechanism to their privileged operations through annotation as was done by EPOXY [17].

Corrupting indirect calls: μRAI enables preventing attacks targeting backward edges and within exception handler contexts. To protect the forward edge, μRAI leverages a state-of-the art forward edge type-based CFI mechanism. We acknowledge the limitations of forward edge CFI.

Limiting the overhead of SFI: Interrupts are designed to be short and execute in deterministic time on MCUS [77]. While μRAI efficiently restricts SFI to exception handlers, SFI may still result in higher overhead in some cases. An alternative to SFI is formal verification of exception handlers which we leave for future work.

Applicability to ARMv8-M and systems with an OS: We demonstrated μRAI on bare-metal systems and ARMv7-M to show its applicability to most constrained systems. Since ARMv8-M is backward compatible, we believe μRAI is extensible to it. Moreover, μRAI can utilize the TrustZone provided in ARMv8-M for its safe region. Extending μRAI to systems with an OS requires modifying the context switch handler to save and restore the SR per each thread. In addition, μRAI require restricting the use of the register chosen as the SR. If such changes are made, μRAI can apply its defenses to systems with a lightweight OS.

IX. Conclusion

MCUS are increasingly deployed in security critical applications. Unfortunately, even with proposed defenses, MCUS remain vulnerable against ROP style attacks. We propose μRAI, a security mechanism able to *prevent* control-flow hijacking attacks targeting backward edges by enforcing the RAI property on MCUS. μRAI does not require any special hardware extensions and is applicable to the majority of MCUS. We apply μRAI on five realistic MCUS applications and show that μRAI incurs negligible runtime overhead of 0.1%. We evaluate μRAI against various scenarios of control-flow hijacking attacks targeting return addresses, and demonstrate its effectiveness in preventing all such attacks.

Acknowledgements

We thank Hui Peng, Adrian Herrera, Derrick McKee, and Yuseok Jeon for their insightful comments. This work was supported by ONR award N00014-17-1-2513, by NSF under award numbers CNS-1801601 and CNS-1718637, and by Sandia National Laboratories through its Academic Alliance partnership. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525 SAND2020-0869 C.

REFERENCES

[1] I. Analytics, "State of the IoT 2018: Number of IoT devices now at 7B Market accelerating," 2018, https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[3] S. Edwards and I. Profetis, "Hajime: Analysis of a decentralized internet worm for iot devices," *Rapidity Networks*, vol. 16, 2016.

[4] "Hijacking drones with a MAVLink exploit," 2016, http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.

[5] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling uavs with sensor input spoofing attacks." in *WOOT*, 2016.

[6] A. Cherepanov, "WIN32/INDUSTROYER: A new threat for industrial control systems," 2017, https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf.

[7] ARM, "Mbed-OS," https://github.com/ARMmbed/mbed-os, 2019.

[8] FreeRTOS, "FreeRTOS," https://www.freertos.org, 2019.

[9] T. Kobayashi, T. Sasaki, A. Jada, D. E. Asoni, and A. Perrig, "Safes: Sand-boxed architecture for frequent environment self-measurement," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, ser. SysTEX '18. New York, NY, USA: ACM, 2018, pp. 37–41. [Online]. Available: http://doi.acm.org/10.1145/3268935.3268939

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

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

[12] M. Werner, T. Unterluggauer, D. Schaffenrath, and S. Mangard, "Sponge-based control-flow protection for iot devices," *arXiv preprint arXiv:1802.06691*, 2018.

[13] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: tiny trust anchor for tiny devices," in *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*. IEEE, 2015, pp. 1–6.

[14] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 10.

[15] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 106.

[16] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "Aces: Automatic compartments for embedded systems," in *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[17] 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 *Security and Privacy Symp.* IEEE, 2017.

[18] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing real-time microcontroller systems through customized memory view switching," in *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[19] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uxom: Efficient execute-only memory on ARM cortex-m," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 231–247. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/kwon

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

[21] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[22] G. Beniamini, "Project Zero: Over The Air: Exploiting Broadcoms Wi-Fi Stack," 2017, https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017.

[23] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, p. 16, 2017.

[24] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[25] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 941–955.

[26] H. Shacham, "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*. ACM, 2007, pp. 552–561.

[27] N. Carlini and D. Wagner, "{ROP} is still dangerous: Breaking modern defenses," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 385–399.

[28] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 161–176.

[29] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 401–416.

[30] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.

[31] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 121–138.

[32] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 227–242.

[33] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz *et al.*, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity." in *NDSS*, 2017.

[34] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 147–163.

[35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[36] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[37] N. Burow, X. Zhang, and M. Payer, "Shining light on shadow stacks," in *IEEE Security and Privacy Symp.* IEEE, 2019.

[38] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 555–566.

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

[40] 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 *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[41] ARM, "Trustzone for cortex-m," https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m, 2019.

[42] M. Electronics, "S32K148EVB-Q176," https://www.mouser.com/ProductDetail/NXP-Semiconductors/S32K148EVB-Q176?qs=W0yvOO0ixfEkuVrgK4IOLA%3D%3D, 2019, 2019.

[43] ——, "S32K146EVB-Q144," https://www.mouser.com/ProductDetail/NXP-Semiconductors/S32K146EVB-Q144?qs=sGAEpiMZZMtw0nEwywcFgJjuZv55GFNmk%252B1dyigtrYT9dHJtWqKRcQ%3D%3D, 2019, 2019.

[44] ——, "KDK350ADPTR-EVM," https://eu.mouser.com/ProductDetail/Texas-Instruments/KDK350ADPTR-EVM?qs=qSfuJ%252Bfl%2Fd7gVa9B0YeXTA==, 2019, 2019.

[45] ——, "EV-COG-AD4050LZ," https://eu.mouser.com/ProductDetail/Analog-Devices/EV-COG-AD4050LZ?qs=BZBei1rCqCCc%2Fxr7P1LvhQ==, 2019, 2019.

[46] ——, "MAX32660-EVSYS," https://www.mouser.com/ProductDetail/Maxim-Integrated/MAX32660-EVSYS?qs=sGAEpiMZZMtw0nEwywcFgJjuZv55GFNmU%252BdtNOmmq%252BDQXtc6gfhDiw%3D%3D, 2019, 2019.

[47] E. Sourcing, "Reversal of fortune for chip buyers: average prices for microcontrollers will rise," 2017, http://www.electronics-sourcing.com/2017/05/09/reversal-fortune-chip-buyers-average-prices-microcontrollers-will-rise/.

[48] R. York, "ARM Embedded segment market update," 2015, https://www.arm.com/zh/files/event/1_2015_ARM_Embedded_Seminar_Richard_York.pdf.

[49] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *Usenix Security Symposium*, 2010.

[50] "Control flow integrity clang 9 documentation - llvm," https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2019.

[51] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 577–587. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295

[52] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[53] EEMBC, "Coremark - industry-standard benchmarks for embedded systems," http://www.eembc.org/coremark.

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

[55] ARM, "Armv7-m architecture reference manual," https://developer.arm.com/docs/ddi0403/e/armv7-m-architecture-reference-manual, 2014.

[56] "Stm32479i-eval," http://www.st.com/resource/en/user_manual/dm00219329.pdf, 2018.

[57] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[58] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-oriented programming on a cortex-m processor," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 823–832.

[59] B. G. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, no. 3, pp. 216–226, 1979.

[60] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1470–1486.

[61] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 28–39. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274739

[62] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[63] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[64] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008, pp. 11–15.

[65] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," *ACM SIGPLAN*, vol. 42, no. 6, pp. 278–289, 2007.

[66] "CVE-2017-6956." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6956, 2017.

[67] "CVE-2017-6957." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6957, 2017.

[68] "CVE-2017-6961." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6961, 2017.

[69] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[70] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner, "Detecting format string vulnerabilities with type qualifiers." in *USENIX Security Symposium*, 2001, pp. 201–220.

[71] "Emerging Stack Pivoting Exploits Bypass Common Security," 2013, https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security/.

[72] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 16–26, 2014.

[73] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 276–291.

[74] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein, "Systematic analysis of defenses against return-oriented programming," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2013, pp. 82–102.

[75] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, "The dynamics of innocent flesh on the bone: Code reuse ten years later," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1675–1689.

[76] K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen, "Leakage-resilient layout randomization for mobile devices." in *NDSS*, 2016.

[77] J. Ganssle, *The art of designing embedded systems*. Newnes, 2008.

[78] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[79] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cfixx: Object type integrity for c++ virtual dispatch," in *Prof. of ISOC Network & Distributed System Security Symposium (NDSS). https://hexhive. epfl. ch/publications/files/18NDSS. pdf*, 2018.

APPENDIX A
HANDLING SPECIAL RECURSIVE FUNCTIONS

Special cases of recursion such as functions with multiple recursive calls require additional instrumentation. In the following we discuss two possible designs to support the RAI property. The first provides flexible handling of recursion by

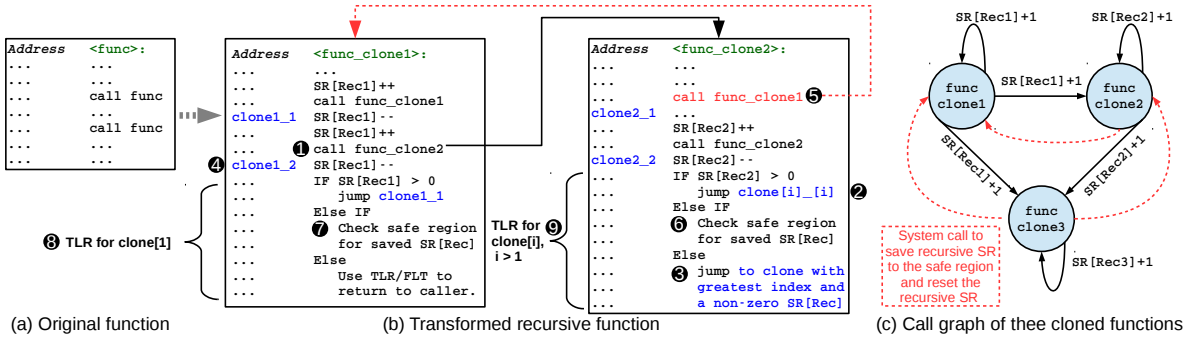

Fig. 13. Illustration of using SR segmentation to resolve multiple recursion. Red-dashed edges are backward edges (i.e., from higher indexed clones to lower indexed clones). that trigger a system call to save the SR to the safe region and reset SR.

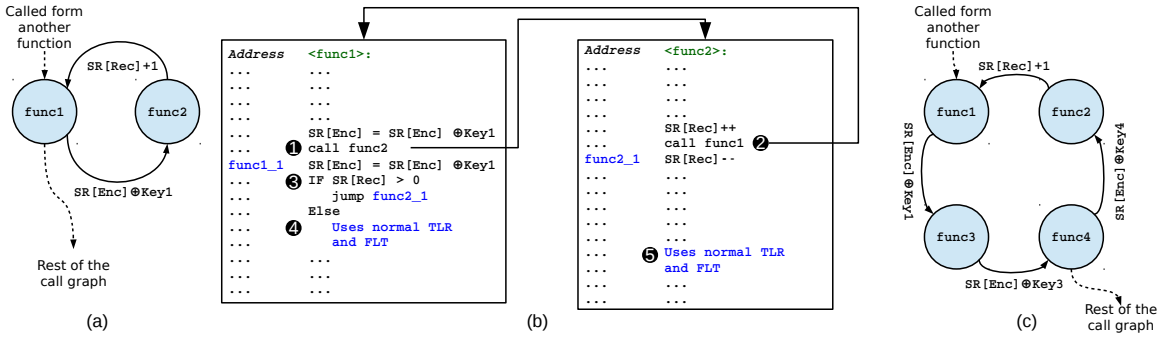

Fig. 14. Illustration of handling indirect recursion. Fig.(a) shows a call graph of two indirect recursive functions. Fig(b) shows a pseudo code of instrumenting indirect recursive functions. Fig(c) illustrates a cycle of four functions. Functions `func1` and `func2` are handled in the same method as the first case in(a).

mainly utilizing the safe region for special cases of recursion. However, this results in overhead since each access to the safe region requires a system call. Thus, we provide a sketch of a second design that uses function cloning to reduce the performance overhead of transitioning to the safe region. In the following we describe both options.

A. Safe Region Recursion

The goal of this design is to provide flexible support of special recursion cases (e.g., indirect recursion between two functions) while using the minimum number of bits in the SR. Instead of reserving a recursion counter for each recursive call, e.g., in case of multiple recursion as in Figure 13(a), this design only reserves the highest order bit of the SR to indicate the occurrence of special recursive calls. In addition, it reserves a special *recursion stack* in the safe region, where the bottom of the stack has a known value. Each special recursive call is instrumented with a system call. The system call sets the special recursive bit and stores the return address on the *recursion stack*. The subsequent recursive call is transformed into a direct call (i.e., to ensure it does not use `LR`, our SR) and is executed normally.

The special recursive function TLR is instrumented in the beginning to check if the state of the special recursion bit. If set, it executes a system call that will pop the saved return address from the *recursion stack* and return to it. In addition, it will reset the special recursion bit if the popped return address is at the end of the recursion stack. In essence, this technique implements a shadow stack for special recursion. An advantage of this design is its efficiency in utilizing the SR bits. It also does not limit the recursion limits the SR size. It is a more portable option for other architectures that do not share the pre-assumption of MCUS of defining the recursion limits a priori. Thus, μRAI implements this design for its prototype.

B. Handling Special Recursion Using Function Cloning

1) Handling Multiple Recursion Functions: This method aims to handle special cases of recursion while limiting the reliance of the safe region, and mainly utilize the recursion counters of SR through function cloning. Consider Figure 13(a), the original function is transformed by creating separate clones of the function. Each clone corresponds to one of the recursive calls in the original function. Each recursive call is directed to its corresponding clone, which uses a distinct segment of the recursion counter. This enables each clone to check only its respective recursion segment as shown in Figure 13(b). Without re-directing each recursive call to a separate clone of the recursive function, it is not possible to resolve which of the recursive calls is the correct return target. Function cloning solves a general case when the recursive calls within a function execute in *any ascending* order (e.g., `func_clone1` calls `func_clone2` in ❶). To return, a clone decrements its counter and returns to itself until its recursive SR segment reaches zero (e.g., `func_clone2` will return to `clone2_2` in ❷). For a clone[i] to return to its caller, it then checks the non-zero recursive counters in the SR as these indicate which previous clone is the caller of the current clone (i.e., clone[i] checks all $SR_{Rec}[j < i]$ where). The clone with the *highest index* in SR_{Rec} is the correct caller, and clone[i] return to the instruction following the call site of clone[i] in the resolved caller (i.e.,❸ will return to `clone1_2` at ❹) .

However, whenever multiple recursion executes out-of-order, as in `call func_clone1` in function `func_clone2` (❺), cloning alone cannot distinguish how many increments in SR_{Rec} happened before or after the

Algorithm 1 Multiple indirect recursion call procedure

1: **procedure** INDIRECT CALL[I] OUT OF N INDIRECT CALLS
2: **if** $SR_{Rec[i]} \geq SR_{Rec[j]}$, where $SR_{Rec[i]} \neq 0$ and $SR_{Rec[j]} \neq 0$, for any $j > i$ **then**
3: Save the SR and reset SR_{Rec}
4: **end if**
5: $SR_{Rec[i]} = SR_{Rec[i]} + 1$
6: Execute indirect call
7: $SR_{Rec[i]} = SR_{Rec[i]} - 1$
8: **if** All $SR_{Rec}counters = 0$ and Safe region contains saved SR **then**
9: Restore saved SR
10: **end if**
11: **end procedure**

Algorithm 2 Multiple indirect recursion return procedure

1: **procedure** RESOLVING CORRECT RETURN TARGET OF OUT OF N INDIRECT CALLS
2: // Check recursion counters in descending order
3: **if** $SR_{Rec[N]} \geq 0$ **then**
4: Return after indirect recursive call N
5: **end if**
6: **if** $SR_{Rec[N-1]} \geq 0$ **then**
7: Return after indirect recursive call N-1
8: **end if**
9: // Inline the check until the lowest index of recursion counters
10: // ...
11: // After inlining all checks
12: **if** All SR_{Rec} equal 0 **then**
13: //All indirect recursion have been resolved
14: Use the regular TLR
15: **end if**
16: **end procedure**

descending (i.e., to a clone with lower index) recursive call in (❺). Thus, any descending recursive call requires a system call to: (1) save the current recursive SR in the *safe region*; (2) reset the recursive SR. Upon returning, each clone function will check the safe region for saved recursive counters (❻ and ❼). Checking the safe region triggers a system call that will check the saved recursive counters and find clones with $SR_{Rec} > 0$. Again, the *highest index* in SR_{Rec} is the correct caller, and clone[i] returns to the instruction following the *descending* call site of clone[i] in the resolved caller in (i.e.,❼ will return to ❺). In general, a function with N recursive calls is transformed to into N cloned functions forming a complete digraph. Ascending edges are handled by incrementing the respective recursions counter and decrementing it afterwards. A descending edge (i.e., red-dashed edges in Figure 13(c)), are system calls to reset the recursive SR counters. A call graph representation of transforming a function with three recursive calls is shown in Figure 13(c). The first function (i.e., clone1) will use the TLR of clone[1] (❽), while other nodes will use the TLR clones > 1 (❾).

2) Handling Indirect Recursion: Indirect recursion is another special case that requires additional care. To understand the difference of indirect recursion, consider the conceptual call graph illustration shown in Figure 14(a). Indirect recursion causes a cycle in the call graph, as demonstrated by `func1` and `func2`. In such a case, it is not possible to handle indirectly recursive functions using XOR encoding as it results in FID collisions. Indirect recursion is handled by using the maximum recorded depth of each function in the call graph (see Section IV-D). When a call forms a cycle at the callee, this will always be a call from a function with higher maximum depth to a function with lower maximum depth, and identifies an indirect recursion. In such a case, the call causing the indirect recursion uses a recursion counter segment. However, the check for the recursive counter is done at the callee.

Consider the illustration shown in Figure 14(b), the call from `func1` to `func2` use the XOR encoding since it is from a lower depth to a higher depth (❶). However, the call from `func2` to `func1` uses one of the recursion counter segment to increment and decrement the counter before and after the call, respectively (❷). Checking the recursion counter however, is done in `func1` instead of `func2` (❸). Note that this requires iterating through the cycle between `func1` and `func2` until reaching a fixed set of the possible FIDs for both functions that result from the XOR encoding between `func1` and `func2`. To identify its caller, `func1` first checks its recursion counter. In case, it is greater than zero (❸), it returns to the call site causing the indirect recursion, otherwise it uses the regular TLR policy ❹. For `func2`, it uses the TLR policy directly (❺) as all FIDs are from the XOR encoding chain. Note that the same transformation is used for a cyclic call (i.e., call causing a cycle in the call graph). For example, `func1` and `func2` are handled in the same method in Figure 14(c).

Finally, in case of indirect recursion with multiple call sites, each call site uses a segment counter, as shown in Algorithm 1. In addition, each call site is given an index in the order they appear in the function. A check is inserted before each indirect recursive call ensure the indirect recursive calls execute in ascending order (i.e., line two in Algorithm 1). As discussed in Appendix A-B1, in case of calls executing out-of-order (e.g., indirect recursive call[2] executes after indirect recursive call[3] has already executed), a system call is needed to save the SR to the safe region and reset the recursion counters. The callee of the indirect recursive calls checks each segment counter in descending order (i.e., to enforce the return to the call sites ascending order), as shown in Algorithm Algorithm 2. This enables returning correctly to the correct call site in the caller (i.e., `func2`). Function `func2` checks the value of all recursion counters after returning from each call site. In case all recursion counters are zero, it checks the safe region for a saved SR, and rosters the SR value (i.e., line eight in Algorithm 1). The process repeats until no saved SR is found in the safe region, at which no indirect recursion is left and the function can return normally using its TLR.

APPENDIX B
MISCELLANEOUS

A. Imprecision and Irregular Control-Flow Transfers

μRAI handles imprecisions through separate encoding keys for each possible target in the target set of its type-CFI. For example, if a target set contains {`func1, func2`} and the indirect call is pointing to func1. Then μRAI will branch to the encoding routine specific `func1`. If the target is a recursive function then a recursive encoding is used. This ensures the function will always return to its exact caller and reduce the effect of complexity since we find a satisfying key for each target individually instead of the entire target set.

Our type-CFI implementation, although imprecise, has a maximum target set of five functions for an indirect call on our evaluation. μRAI is evaluated on `C`. Using `C++` code results in two challenges. The first is larger target set for type-CFI, thus larger FLT sizes. The second are attacks targeting the virtual table pointer such as COOP style attacks [78]. These however are an orthogonal problem to stack safety and may be protected through, e.g., OTI [79].

Supporting irregular control-flow transfers such as `setjmp` and `longjmp` requires custom system calls. For `setjmp`, the system call stores the return location along with the current SR value in the safe region. When executing `longjmp`, a system call restores the return location and SR from the safe region. This mechanism also enables the use of pre-compiled objects (e.g., `newlib`) since μRAI requires source code.

B. Scalability and Encoder Efficiency Discussion

μRAI scales its encoding scheme for larger applications by partitioning the call graph whenever no satisfying keys are found as shown in Figure 8. This frees up the SR to be reused, but results in some overhead since each transition between partitions requires a system call. For our experiments, partitioning the call graph was not needed. Table VII shows the number of call sites in our evaluation, as well as the number of nodes and edges handled by the encoder in the call graph. The number of nodes does not equal the number of functions as many are optimized by inlining in LLVM. In addition, the number of edges is different than the number of call sites as a result of imprecisions in the call graph. Our evaluation uses applications compare to and exceed the complexity of existing benchmarks (e.g., Animation compared to CoreMark).

We further evaluate μRAI's encoder and whether it populates the FLTs efficiently. μRAI's TLR requires the correct branch to be at a distance equal to the SR in FLT. Thus, unused FIDs in the FLT are filled with `JUMP ERROR` to ensure the FLT is correctly aligned as was shown in Figure 9. Table VIII shows the FLT efficiency for each application. We compute the FLT efficiency as the percentage of used FLT indices over the total FLT size. If the FLT is sparse (i.e., a large number of FIDs are filled with `JUMP ERROR`), then the efficiency is lower. Note that the encoder uses the information extracted by the call graph analyzer (i.e., Section V-A). The call graph analyzer is implemented as an LLVM IR pass. LLVM further optimizes the application after this pass. Such optimization may remove call sites (e.g., due to inlining) before μRAI instruments the firmware using its back-end pass (see Section V-C). As a result, some portions of the encoder's estimated FLT are unused. For example, if the encoder estimated FLT contains a branch instruction to a function that has been removed then this branch is ultimately replaced by a `JUMP ERROR`. This results in lower efficiency in the final binary when compared to the encoder generated FLT (e.g., 98.2% and 90.1% for average FLT efficiency in *Animation*). Averaged across all the applications, the encoder and binary demonstrate a high FLT efficiencies as 98.3% and 93.8%, respectively.

```
1 eor lr, #FK          ;Encode LR
2 b func               ;Direct call to func
3 eor lr, #FK          ;Decode LR
```

Listing 1. μRAI instrumentation for call instructions.

TABLE VI. ANALYSIS OF μRAI TRANSFORMATIONS AND ITS EFFECT ON RUNTIME OVERHEAD. N: NUMBER OF REGISTERS USED IN THE INSTRUCTION. P: PIPELINE REFILL. P CAN BE BETWEEN 1–3 CYCLES.

Instruction	**Baseline**	**μRAI**	
	# of cycles	Effect	# of cycles
`push {..,lr}`	1 + N	Remove `lr`	N
`pop {..,pc}`	1 + N + P	Remove `pc`	N
`eor`		Add 2	2
`mov`		Add 2	2
`add pc,<reg>`		Add 1	1 + P
`b <label>`		Add 1	1 + P
Total	2 + 2 N + P		4 + 2 N + 2 P
μRAI Overhead			2 + P

TABLE VII. SUMMARY OF THE NUMBER OF CALL SITES INSTRUMENTED BY μRAI, NUMBER OF NODES, AND EDGES IN THE CALL GRAPH FOR EACH APPLICATION.

App	**# of Call sites**		**# of Nodes**	**# of Edges**
	Direct	Indirect		
PinLock	26	0	22	26
FatFs_uSD	111	97	37	473
FatFs_RAM	29	94	25	373
LCD_uSD	157	52	44	343
Animation	152	52	46	349
CoreMark	91	0	21	94

TABLE VIII. SUMMARY OF μRAI FLT EFFICIENCY EVALUATION.

App	**Encoder**			**Binary**		
	Min.	Max.	Ave.	Min.	Max.	Ave.
PinLock	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
FatFs_uSD	83.3%	100.0%	97.1%	38.9%	100.0%	90.9%
FatFs_RAM	83.3%	100.0%	97.6%	75.0%	100.0%	95.8%
LCD_uSD	81.6%	100.0%	98.1%	44.4%	100.0%	94.0%
Animation	75.0%	100.0%	98.2%	22.2%	100.0%	90.1%
CoreMark	87.7%	100.0%	98.5%	43.9%	100.0%	92.0%

```
1 add pc,lr              ;First relative jump
2 b return_location_1    ;First FID
3 b return_location_2    ;Second FID
```

Listing 2. TLR instrumentation without SR segmentation.

```
1 ; N = 32 - function shift - segment bit size
2 mov r12,lr,lsl #N      ;r12 = lr<<N
3 ; M = 32 - segment bit size
4 mov r12,r12,lsr #M     ;r12 = r12>>M
5 add pc,r12             ;First relative jump
6 b return_location_1    ;First FID
7 b return_location_2    ;Second FID
8 ...
```

Listing 3. TLR with SR segmentation. N and M are constants calculated depending on the function and the start of its segment.

```
1 add lr,#0x01000000 ;increment counter
2 b func             ;direct call to func
3 sub lr,#0x01000000 ;decrement counter
```

Listing 4. An example of μRAI's instrumentation for recursive call sites. The recursion counter shown uses the higher eight bits of `LR`.

```
1  ; rd = destination register
2  ; rx = any available register other than rd
3  ;-------------------------------------
4  ; Condition flags are saved prior
5  movw rx,#0xE000; Higher word of MPU_RNR
6  movt rx,#0xED98; Lower word of MPU_RNR
7  sub rx,rd,rx    ; rx = rd - MPU_RNR (unsigned)
8  cmp rx,#8       ; If within MPU registers
9  bls ERROR       ; ERROR if less or equal
10 cmp rx, #0xd90  ; If points to VTOR
11 beq ERROR       ; ERROR if equal
12 bic rd, rd, 0x10000000 ; Safe region mask
13 ; restore condition flags and perform store
```

Listing 5. μRAI's exception handler SFI protection. The MPU Region Number Register (`MPU_RNR`) is middle address of the MPU.