INTERNET OF THINGS SYSTEMS SECURITY:

BENCHMARKING AND PROTECTION

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Naif Saleh Almakhdhub

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Saurabh Bagchi, Co-Chair

School of Electrical and Computer Engineering

Dr. Mathias Payer, Co-Chair

School of Computer Science

Dr. Milind Kulkarni

School of Electrical and Computer Engineering

Dr. Felix Xiaozhu Lin

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the School of Electrical and Computer Engineering

To my beloved parents, Saleh and Lyla, whose sacrifices and love are the reason for all my success; to my dear grandparents, Abdulaziz, Rashid, Dalal, Nora, and Aljohrah, who are the role models I aspire to be; and to my beloved wife Ghaida and my son Saleh (and his future siblings), who fill my heart with joy and happiness.

ACKNOWLEDGMENTS

This dissertation would not come to fruition without the countless sacrifices and love from my family. My dear father, who exemplified the value of hard work and instilled in our family the desire to achieve the highest levels of education. My beloved mother, whose dedication and rigorous standards of teaching enabled my siblings and I to succeed. My dear grandmother Dalal, whose faith in me never wavers.

I owe the world to my beloved wife Ghaida, and my dear son Saleh, who sacrificed so much to accompany me throughout this journey, and filled my heart with love and joy.

My heartfelt thanks to my siblings Dalal, Alanoud, Nora, Abeer, and Abdulaziz, who surround me with love and happiness.

I am grateful to my advisors, Prof. Mathias Payer and Prof. Saurabh Bagchi. I would not have reached this milestone without their careful reviews, generous feedback, and their guidance throughout my time at Purdue. They taught me invaluable skills that will guide my career for years to come. I would like to extend my thanks to the members my thesis committee, Prof. Milind Kulkarni and Prof. Felix Lin for their time and valuable feedback. I am grateful to Abraham Clements for his help, advice, and the long late nights we worked on papers together. I could not have asked for a better lab mate than him.

It was my pleasure to be a part of the HexHive group and the Dependable Computing Systems Laboratory (DCSL). I am grateful for all the reviews and feedback from my peers. I am proud of the direct and honest feedback culture within both groups that holds the quality of our work to high standards. Special thanks to my lab mates Mustafa Abdallah, Ashraf Mahgoub, Edgardo Barsallo Yi, and Manish Nagaraj for the joyful conversations we had together. My deepest thanks to all my friends, especially Abdulellah Alsaheel, Ali Alsahli, Abdulaziz Alshayban, and Mohammed Aljardan, for their support throughout my graduate studies.

I would like to express my gratitude to King Saud University, and the Saudi Arabian Cultural Mission for funding me throughout my graduate studies. My deepest thanks to my mentor, Prof. Saleh Alshebeili, who taught me how to preserve and work effectively under pressure. The lessons and skills I learned from him were invaluable to succeed in graduate school. I am also grateful to Prof. Basil AsSadhan, Prof. Saeed Aldosari, Prof. Aggelos Katsaggelos, and Prof. Goce Trajcevski for their generous help and guidance during my graduate studies.

My heartfelt gratitude to my extended family for their continuous love and support, especially my uncles, Abdulfattah, Mohammed, Hisham, Sami, Moath, Mohammed, Saleh, Abdulhameed, Fahd, and my aunts, Moneera, Latifa, Bodoor, Rihab, Thuraya, Albandree, Moneera, Rabeaa, and Bothaina.

I am most grateful to my beloved late grandfather Abdulaziz, may Allah (god) grant him paradise. No one was more instrumental after Allah in completing this PhD than him. I vividly remember his words inspiring me to pursue a PhD as an adolescent. Throughout my PhD, especially during tough times, his wise advice and encouraging words were the ones that lift my spirit, kept me going, and eventually helped me reach the other shore. His words will always be the shining stars enlightening my path.

Most importantly, I thank almighty Allah for the countless blessings upon me, I have been lucky beyond any means. I seek his mercy and forgiveness, and pray that he helps me do good with what I have learned.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Almakhdhub, Naif Saleh Ph.D., Purdue University, May 2020. Internet of Things Systems Security: Benchmarking and Protection. Major Professor: Saurabh Bagchi and Mathias Payer.

Internet of Things (IoT) systems running on Microcontrollers (MCUS) have become a prominent target of remote attacks. Although deployed in security and safety critical domains, such systems lack basic mitigations against control-flow hijacking attacks. Attacks against IoT systems already enabled malicious takeover of smart phones, vehicles, unmanned aerial vehicles, and industrial control systems.

The thesis introduces a systemic analysis of previous defense mitigations to secure IoT systems. Building off this systematization, we identify two main issues in IoT systems security. First, efforts to protect IoT systems are hindered by the lack of realistic benchmarks and evaluation frameworks. Second, existing solutions to protect from control-flow hijacking on the return edge are either impractical or have limited security guarantees. This thesis addresses these issues using two approaches.

First, we present BenchIoT, a benchmark suite of five realistic IoT applications and an evaluation framework that enables automated and extensible evaluation of 14 metrics covering security, performance, memory usage, and energy. BenchIoT enables evaluating and comparing security mechanisms. Using BenchIoT, we show that even if two security mechanisms have similarly modest runtime overhead, one can have undesired consequences on security such as a large portion of privileged user execution.

Second, we introduce Return Address Integrity (RAI), a novel security mechanism to prevent all control-flow hijacking attacks targeting return edges, without requiring special hardware. We design and implement μRAI to enforce the RAI property. Our results show μRAI has a low runtime overhead of 0.1% on average, and therefore is a practical solution for IoT systems.

This thesis enables measuring the security IoT systems through standardized benchmarks and metrics. Using static analysis and runtime monitors, it prevents control-flow hijacking attacks on return edges with low runtime overhead. Combined, this thesis advances the state-of-the-art of protecting IoT systems and benchmarking its security.

1. INTRODUCTION

1.1 Motivation

Embedded and IoT systems are deployed in security and privacy critical application such as health-care, industrial, Unmanned Ariel Vehicles (UAVs), or smart-home systems. Estimates project the number of IoT devices to exceed 22 billion devices by 2025 [1]. Unfortunately, this rapid growth is coupled with an alarming number of attacks against IoT devices. Such attacks caused some of the largest Distributed Denial of Service (DDoS) attacks to date (e.g., Mirai [2] and Hajime [3]), enabled tampering of critical infrastructure data [4], and allowed malicious takeover of UAVs [5,6] among others.

This thesis focuses on benchmarking the security of microcontroller-based IoT systems (IoT-MCUS) and protecting them from remote memory corruption attacks. Microcontroller systems (MCUS) are a significant portions of deployed IoT devices. UAVs, smart locks, Engine Control Unit (ECU) gateways, and WiFi System-on-Chip (SoC) are examples of such systems. MCUS are resource constrained systems: (1) they have few MBs of Flash and hundreds of KBs of RAM; (2) they run a single statically linked binary image; (3) they run either with a lightweight Operating System (OS), or as bare-metal (i.e., without an OS). The single binary image is responsible for application logic, security configuration, and accessing peripherals. As a result of their limited resources (e.g., hardware constraints), MCUS cannot use the same mechanisms from general desktop systems to deploy well known defenses such as randomization-based defenses (e.g., Address Space Layout Randomization–ASLR), or stack canaries [7]. However, even if such defenses are deployed, they offer weaker guarantees. The problem is exacerbated by the poor security practices of MCUS vendors and developers (e.g., not enforcing Data Execution Prevention or `W⊕X` [8–10]).

MCUS can be stand-alone devices, or a component of a larger system. Thus, vulnerabilities on MCUS are not confined to the device itself, but can risk the security of the entire system. For example, hijacking the control of a WiFi SoC can lead to malicious control of the underlying application processor of a smart phone as shown by Google's Project Zero [11]. These attacks gain arbitrary code execution on MCUS by hijacking the control-flow of the firmware.

As on desktop systems, control-flow hijacking attacks on MCUS originate from a memory corruption vulnerability violating memory safety or type safety. These attacks occur by corrupting a code pointer either on the forward edge (i.e., function pointers, and virtual table pointers) or backward edge (i.e., return addresses). However, unlike desktop systems, MCUS lack essential control-flow hijacking defenses as mentioned above. The absence of essential defense mechanisms, combined with their limited resources caused MCUS to become a prominent target for control-flow hijacking attacks.

This thesis introduces a systemic analysis of defense mitigations to secure IoT-MCUS. Building off this systematization, we find that despite the multiple proposed solutions to protect IoT-MCUS [8, 9, 12–18], such efforts suffer from two limitations. First, the lack of standardized and automated evaluation of security mechanisms. Second, existing solutions are either impractical (e.g., high runtime overhead) or have limited security guarantees.

First, IoT-MCUS lack a representative benchmarking suite of its IoT applications. Benchmarks from the server world [19, 20] or the embedded world [21–23] either are not applicable to IoT-MCUS (e.g., the require large memory) or do not represent the characteristics of IoT applications such as network connectivity and rich interactions with peripherals (e.g., sense, actuate). As a results, security evaluation on IoT-MCUS became limited and ad-hoc. This made comparisons between security techniques infeasible, as each are evaluated using different case studies or with different benchmarks that do not represent realistic IoT-MCUS applications. The problem is exacerbated by the tedious and manual evaluation process.

Second, current defenses to mitigate control-flow hijacking attacks on MCUS have limited security guarantees, incur high runtime overhead, or require extra hardware. The majority of MCUS lack special hardware features such as an additional processor, or Trusted Execution Environment (TEE). Moreover, many require adhering to strict real-time requirements. Thus, defenses relying on hardware extensions such as TEE (e.g., TrustZone [24]) or incurring high runtime overhead are not applicable to a large portion of MCUS. As a results, control-flow hijacking attacks such as Return Oriented Programming (ROP) remain a threat for MCUS.

1.2 Thesis statement

This dissertation addresses the limitation of benchmarking IoT systems and protecting them from remote control-flow hijacking attacks. Our first objective is to standardize the evaluation process of IoT-MCUS with realistic benchmark applications and metrics. The second objective is to prevent control-flow hijacking attacks on IoT-MCUS even in the presence of memory corruption vulnerabilities by using static analysis and runtime monitors.

The thesis statement is:

> *The creation of standardized benchmarks and metrics enables the measurement of security properties on heterogeneous embedded IoT systems. Customized static analysis and runtime monitors can prevent control-flow hijacking attacks on return edges at low runtime overhead.*

We prove the thesis statement using two approaches.

1.2.1 BenchIoT: A security benchmark for the Internet of Things

We achieve the first objective with BenchIoT, a set of five realistic IoT-MCUS applications and an evaluation framework to address the limitations of evaluating IoT-MCUS security. BenchIoT uses a software based approach to enable portable and extensible evaluation without relying on additional hardware extensions. The

BenchIoT benchmarks mimic realistic application by enabling rich interactions with peripherals (i.e., sense, process, and actuate). When such interactions require an external event (e.g., pushing a button), BenchIoT enables repeatable execution of such events without variation by triggering them at specific points in software.

The evaluation framework enables collecting 14 static and dynamic metrics covering security, performance, memory and energy. BenchIoT provides its evaluation with modest overhead of 1.2%. It enables comprehensive evaluation of security defenses, demonstrating unreported effects such as even though two defense mechanisms can have similar runtime overhead, one can have an adverse effect on energy consumption or have large portions of privileged user execution.

1.2.2 μRAI: Securing Embedded Systems with Return Address Integrity

To achieve our second objective, we introduce Return Address Integrity, a novel security mechanism to prevent all control-flow hijacking attacks targeting the return edges. We design and implement μRAI to enforce the Return Address Integrity property. To achieve its goals, μRAI removes return addresses from writable memory. Instead, μRAI resolves the correct return location using a combination of a single general purpose that is never spilled and a jump table in readable and executable only memory. We evaluate μRAI against the different control-flow hijacking attacks scenarios (i.e., buffer overflow, arbitrary write, and stack pivot) targeting the return edges and demonstrate μRAI prevents them all. Our results show μRAI enforces all its protections without requiring special hardware, and with low runtime overhead of 0.1%. Thus, it is both a practical and secure solution for IoT-MCUS.

1.3 Contributions and Work Publication

This thesis is mostly drawn from work which have been peer reviewed and published in high-quality security conferences. Our benchmarking framework BenchIoT [10] appeared at the 49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2019. Our control-flow hijacking protection mechanism, μRAI, appeared at the 27th Internet Society Network and Distributed System Security Symposium (NDSS) 2020 [25]. Our survey will be submitted to the 42nd IEEE Symposium on Security and Privacy (Oakland) 2021. To summarize our contributions are:

- Systematization of existing work to protect IoT-MCUS. We categorize prior work in terms of hardware requirements, security guarantees, and performance. More importantly, we discuss main issues in IoT-MCUS for future research.
- Design, implementation, and evaluation of BenchIoT, a set of five realistic IoT-MCUS benchmarks and an automated evaluation framework. BenchIoT enables measuring 14 metrics covering security, performance, memory usage, and energy consumption. We demonstrate BenchIoT effectiveness by evaluating three IoT-MCUS defense mechanisms and show although two defense mechanisms can have similar runtime overhead, one can have undesired consequences such a large portion of privileged user execution or susceptibility to code injection.
- Design of Return Address Integrity, a novel security mechanism to prevent all control-flow hijacking attacks targeting return edges. We implement μRAI to enforce the Return Address Integrity property, and demonstrate its effectiveness in preventing all control-flow hijacking attacks while maintain a negligible runtime overhead.

1.4 Summary and outline

This dissertation advances the state-of-the-art in securing IoT systems in two folds. First, it presents the design and implementation of a benchmarking framework to automate the evaluation and comparison of security mechanisms for IoT systems. Second, it presents a novel security mechanism enforcing the Return Address Integrity Property (RAI) that prevents all control-flow hijacking attacks on return edges with low runtime overhead, thus combining security and practicality.

The remainder of this thesis is organized as the following: Chapter 2 presents a systemic analysis of prior work to protect IoT systems form remote attacks. Chapter 3 presents BenchIoT, a benchmarking framework to evaluate and compare security mechanisms for IoT systems. In Chapter 4, we present the design, implementation, and evaluation of μRAI, a novel security mechanism enforcing the Return Address Integrity property on IoT systems. Chapter 5 concludes this dissertation.

2. THE LANDSCAPE OF MCUS SECURITY

2.1 Introduction

Memory corruptions are a well-known class of vulnerabilities that have plagued computer systems for decades [26]. The adoption of exploit mitigation techniques such as stack canaries [7] and, Address Space Layout Randomization (ASLR) enhanced the posture of security in general-purpose system. However, memory corruption attacks remain a threat despite the deployment of existing mitigations. As a result, memory exploitation mitigation remains an active research area.

Networked embedded systems, manifested in the Internet of Things (IoT), share the same risks of memory corruption vulnerabilities as in general-purpose systems. IoT systems are deployed in a multitude of applications, ranging from privacy-sensitive scenarios such as healthcare and smart home systems, to safety and security critical applications such as Industrial Control Systems (ICS), satellite systems, and autonomous vehicles. Unfortunately, IoT systems lack the basic hardening techniques already available in general-purpose systems due to a combination of constrained resources and implementation challenges to deploy the same exploit mitigation techniques. Consequently, an alarming number of attacks are targeting IoT systems [2,3,5,11,27], with some resulting in high-profile impacts such as power grid blackouts [28].

Micro-controller systems (MCUS) constitute a large portion IoT devices. However, MCUS are not only present as stand-alone devices in an IoT setting, but are often used as a part of a larger system. Thus, they can jeopardize the entire underlying system. For example, MCUS can be a WiFi System-on-Chip (SoC) on a smart phone or the Engine Control Unit (ECU) gateway in a vehicle, both have been used enable malicious takeover of the underlying system [11,27].

MCUS are a prominent target of attacks due to their limited resources and use of memory unsafe languages (e.g., C/C++). The problem is exacerbated by the poor security practices of MCU vendors. For example, deployed MCUS often lack Data Execution Prevention (DEP), a standard defense for over two decades to defend against code injection. As a result, MCUS are not only susceptible to memory corruption vulnerabilities, but also lack essential mitigations to defend against such vulnerabilities. Attackers leverage these vulnerabilities to hijack the control-flow of the application and gain arbitrary execution. Control-flow hijacking occurs by corrupting an indirect control-flow transfer (i.e., function pointer, virtual dispatch, or return address) towards the attacker malicious code. Alternatively, attackers can corrupt sensitive data to launch a Denial of Service (DoS) attack. In a Cyber-Physical System (CPS), this can cause the device to become unresponsive [12] leading to physical and safety hazards.

Memory corruption bugs on both general-purpose systems and MCUS originate from bugs that violate memory safety or type safety. However, MCUS are constrained systems lacking the same resource used to deploy many of the defenses on general-purpose systems. MCUS utilize a few MBs of Flash for code, and hundreds of KBs of RAM for data. MCUS lack virtual addressing (i.e., Memory Management Unit–MMU), and instead utilize a *single flat physical* address space. The application running on MCUS is a single binary image comprised of the application, libraries, and a lightweight OS (e.g., ARM's Mbed [29] or Amazon's FreeRTOS [30]) in a single address space. MCUS can also run as bare-metal without using a lightweight OS. As a result, MCUS need different defense mechanisms fitting their unique characteristics or an alternate design and implementation of existing defenses in order to be deployed on MCUS.

The vitality of securing MCUS, coupled with their susceptibility to memory corruption vulnerabilities has led to a large body of research to enhance their protection. This chapter systematizes the existing work to protect MCUS against memory corruption exploitation. We taxonomize existing defenses, and analyze their security guarantees, hardware requirements, and performance. More importantly, we identify open problems in securing MCUS for future research.

2.2 Scope and Background

The scope of this chapter focuses on *prior work protecting MCUS' software from memory corruption exploitation [26] at runtime, or ones applicable to MCUS.* It is not our goal to discuss memory corruption attacks and defenses in general, which have been surveyed by Szekers *et al.* [26]. In addition, detecting an already compromised device via static properties methods [31,32], or via network protocols [33,34] are out of scope as they are not the focus of this work.

Our goal is to compare MCUS defenses in terms of security guarantees, hardware requirements, and runtime performance. In order to achieve our goal, we first define the scope of our targeted system. We present a background of MCUS and discuss basic capabilities we assume on general MCUS along with ones we assume require special hardware or software features.

2.2.1 Scope

Embedded systems cover a vast and diverse space of devices, with varying degree of capabilities and processor speed. *The focus of this work is on micro-controller based embedded systems (MCUS).* These are different from high-end embedded systems (e.g., ARM Cortex-A [35,36]), which have clock speeds in GHz, use a Memory Management Unit (MMU), and have large storage space in GBs. Although these systems are special-purpose systems, they use a general-purpose OS (e.g., Linux) with retrofitted user space libraries (e.g., `busybox`). Many high-end embedded systems are already protected with the same defenses from general-purpose systems, such as ASLR [37], since they are equipped with the needed resources to enforce them (i.e., MMU). In contrast, MCUS are more constrained systems that do not posses the required resources to enforce similar mitigations, or cannot enforce them using the same mechanisms. For example, MCUS (e.g., STM32F479I [38] FRDM-K64F [39]), lack an MMU, and have small memory that is insufficient to enforce ASLR.

We define MCUS as an embedded system combining a micro-controller and standard peripherals, which can include network connectivity, on the same *device*. They run as bare-metal (i.e., without an OS) or can utilize a lightweight Real Time Operating System (RTOS) for multi-threaded applications. In either case, MCUS combine the application, libraries, and the OS into a single monolithic firmware image running on the device. In addition, MCUS use *physical addressing*. They enforce read, write, and execute permission by using a Memory Protection Unit (MPU), instead of an MMU. Thus, unlike high-end embedded systems, MCUS lack the memory isolation between the OS and application. The processor speed of MCUS have clock speeds of a few hundreds of MHz at most (e.g., ARM Cortex-M [40], ARM Cortex-R [41], AVR32 [42]). In terms of storage, higher-end MCUS utilize a few MBs of Flash and a few hundreds KBs of RAM.

Our discussion in remaining sections of this chapter are general to any MCUS architecture (e.g., ARMv7-M [40], AVR32 [42]). However, some architectural related background are needed when discussing some of the defenses and clarify some vulnerabilities of MCUS. In such case, we restrict our discussion to the ARMv7-M [40] as it is the most common architecture for 32-bit MCUS [43, 44].

2.2.2 Background and Target System

MPU and privilege levels: We assume the MPU present on the MCUS is capable of supporting two privilege level (i.e., privileged and unprivileged). This assumption is consistent with existing defenses for MCUS [8, 9, 12–16, 18, 45–47].

Trusted Execution Environment (TEE): Code present in a TEE runs with higher privileges than the OS or user code (i.e., hypervisor). Using a TEE can enable new methods of protecting or attesting MCUS. However, these have only been available recently (e.g., TrustZone [24]) for some MCUS. The majority of MCUS do not provide a TEE. In addition, existing TEE lack software support for MCUS (e.g., OS and library support). Thus, we assume requiring a TEE as an additional hardware requirement.

2.3 Defenses

While proposed defenses for MCUS are applied to various systems with varying capabilities and goals, these defenses can be divided into two parts: (1) remote defense and; (2) local defenses. Remote defenses require a remote *trusted* party in addition to the device itself to enforce their protection. Local defenses are self-contained protections that only apply the instrumentation to MCUS. In the following subsections, we describe each type separately along with the mechanisms they enforce.

2.3.1 Remote Defenses

Remote defenses are primarily attestation-based defenses that aim to *detect* an already compromised device. Attestation uses a challenge-response mechanism, where the *prover* (i.e., MCU) answers the challenge from the *verifier* (i.e., a trusted entity). The verifier considers the prover as trustworthy (i.e., not compromised) if the response is correct. The verifier uses a nonce value to ensure the response is fresh (i.e., the attacker is not using an old response to deceit the verifier).

Attestation mechanisms rely on a *trust anchor* on the prover to ensure the integrity of the prover's response. This can be done through software-based attestation [31], hardware-based attestation [48], or hybrid of both [16, 17]. Attestation schemes has been applied to a wide range of systems, many of which are not applicable to MCUS. Thus, we focus our discussion to literature relating protecting MCUS from runtime memory corruption exploitation as mentioned in subsection 2.2.1.

Control-flow attestation: To detect control-flow hijacking attacks (e.g., code reuse attacks) at runtime, control-flow attestation mechanisms [14, 49] use a runtime tracer as its trust anchor. To attest the device, the runtime tracer computes a hash chain of each control-flow transfer during the prover's response. The verifier checks the hash with a correct pre-computed hash (i.e., pre-computed in an offline phase) to verify the trustworthiness of the prover. C-FLAT [14] placed its runtime tracer in a TEE, and instrumented each control-flow transfer in the firmware. However, this leads to high-overhead and can break the compatibility of legacy code. Alternatively, LO-FAT [49] uses a special hardware extension that records control-flow transfer instruction in parallel with the processor to reduce the overhead of instrumenting each control-flow transfer. However, both are susceptible to Time of Check Time of Use (TOCTOU) attacks. That is, an attacker with physical access can alter devices memory between the time of attestation and time of execution. To overcome this limitation, ATRIUM [50] attests control-flow transfers in parallel with the processor. In general, control-flow attestation are targeted to detect control-flow hijacking attacks, but they can also detect cases of non-control data attacks (i.e., in case they alter the pre-computed control-flow). However, they do not detect non-control data where the control-flow is not altered.

Data-flow attestation: To detect all non-control data attacks (i.e., generalized by Data-Oriented Programming–DOP [51, 52]), LiteHAX [18] extends its hardware extensions to record all control-flow and data-flow events. However, such mechanism assumes a one-to-one attestation relation between the verifier and the prover, which might not be practical for collaborative MCUS such as autonomous systems or swarm robotics. Thus, DIAT [53] reduces the overhead of attesting all events by using modular attestation. That is, DIAT [53] relies on architecture support to ensure the underlying software is partitioned into isolated modules, and only attests control-flow and data-flow events between the isolated modules.

2.3.2 Local Defenses

In contrast to remote defenses, local defenses aim to *mitigate* memory corruption exploits rather than detect them. While many of the proposed policies are inspired from general purpose system, applying them to MCUS require different mechanisms and offer different guarantees than on general-purpose systems as discussed in the following.

Information hiding: Information hiding [54] techniques are probabilistic mitigations that aim at mitigating control-flow hijacking attacks by hiding the location of code pointers at randomized locations. Alternatively, these techniques randomize the code layout to break code reuse attacks such as Return Oriented Programming (ROP) [55], where the attacker chains snippets of instructions ending with a return to launch a control-flow hijacking attack. A primary example from general-purpose systems is ASLR, where the base address of each section is randomized at each run. However, MCUS use a single static binary image residing in Flash (see subsection 2.2.1). Thus, MCUS defenses [8,46,47,56] can only randomize the firmware layout once. The firmware is diversified either during compilation or through binary rewriting [56], but it will remain the same after the firmware is loaded to the MCU.

An alternative method to apply information hiding is encrypting the code on the MCU to protect Intellectual Property (IP). SCFP [15] uses a hardware extension to between the CPU's fetch and decode stage to decrypt the firmware and execute it.

Execute-only memory (XoM): To overcome information hiding, attackers resort to information leakage to disclose the code layout [57, 58]. An additional layer of mitigation is to place code in XoM. While MCUS lack hardware support for XoM, it can be applied in software [45] by means of Software-Fault Isolation (SFI) techniques [59, 60]. However, such technique when applied to MCUS lead to higher overhead [46]. To enable a more efficient XoM implementation for MCUS, uXoM [46] uses a load instruction variant of the Cortex-M [40] architecture that can only read unprivileged memory, even if the device is executing in privileged mode. Thus, uXoM [46] sets the code in privileged memory and transforms all load instruction into this special variant.

Stack canary: As on general-purpose systems, stack canaries [7] are deployed to mitigate stack buffer overflows. However, using the same mechanism directly on

MCUS can lead to synchronization errors [47] since MCUS use a single address space and lack privilege separation. As discussed in subsection 2.2.1, even MCUS with an OS supporting multiple threads can share the same code. The task of identifying which code is shared, and which specific to a single thread becomes non-trivial. As a result, μArmor [47] proposes a single global canary to avoid synchronization errors between threads.

Safe stack: Safe stack [61] splits the normal stack into two stacks: unsafe stack and regular stack. It uses static analysis to identify potentially *unsafe* local variables (e.g., array which might be indexed out of bounds), and places them into the unsafe stack. Return addresses and other local data are placed into the regular stack. Safe stack separation prevents stack buffer overflow vulnerabilities. However, adopting it to MCUS requires some changes. Safe stack relied on large address space or the segment register when applied to general-purpose systems in order to split the two stacks. Such features are not available on MCUS. To overcome these limitations, EPOXY [8] placed the regular and unsafe stacks at opposite ends of RAM, with each growing in opposite directions. In addition, EPOXY [8] placed an unmapped region to capture any stack buffer overflow vulnerability. EPOXY relies on randomization to protect the regular stack. A more secure design of safe stack is to isolate it using SFI, however this can lead to high overhead.

Shadow stack: The goal of a shadow stack [62, 63] mechanism is to prevent control-flow hijacking on the return edge (i.e., return addresses). To achieve this, a shadow stack isolates return addresses or duplicates them in a separated and protected stack (i.e., *shadow stack*). To enforce a shadow stack on MCUS, RECFISH places the shadow in a privileged region protected by the MPU. Thus, each return executes a system call to access the privileged shadow stack. This however leads to high runtime overhead (i.e., higher than 10% [26]). More importantly, it remains vulnerable to code executing in privileged mode as it can access the shadow stack. An alternative design is to use a TEE for the shadow stack [13]. While this prevents the attacks, it introduces high overhead to MCUS [13] as it requires switching into the TEE secure context at each return.

Control-Flow Integrity (CFI): CFI [64, 65] aims to mitigate control-flow hijacking attacks by restricting indirect control-flow transfers (i.e., indirect calls and returns) to a pre-computed *valid* target set that do not deviate from the application's Control-Flow Graph (CFG). That is, any indirect control-flow transfer that does not adhere to the CFG is considered malicious. However, generated CFGs are not fully precise to the aliasing problem [66]. Thus, the strength of CFI implementations depends of the precision of the CFG the pre-computed target sets. While the target set can enforced statically or dynamically, dynamic enforcement are not suited for MCUS as they incur high-overhead or require advanced hardware features [67] that are not commonly available for MCUS. While enforcing CFI can be applied to both forward edges (i.e., indirect calls) and backward edges (i.e., return addresses), it is more suitable to protect the forward edges. This is because the target set of backward edges are typically larger than those of forward edges [63, 65] since a backward edge CFI would consider all possible call sites within the valid target set. A coarse-grained CFI is susceptible to control-flow bending attacks [68]. Thus, existing techniques [9,13,15,56,69] use static analysis to compute the valid target sets to enforce CFI for forward edges. We discuss their implementation and precision in section 2.4.

Memory isolation: Memory isolation is a common and effective mechanisms to secure general-purpose systems, where kernel memory and individual processes' memory are isolated from each other. The single address space, lack of privilege separation, strict runtime requirement, and limited resources of MCUS renders the mechanisms to enforce memory isolation on general-purpose systems to be impractical when applied to MCUS. For example, enforcing kernel memory isolation, or requiring a system call at each access to Memory Mapped IO (MMIO) can break the real-time constraints on MCUS [12].

To overcome these limitations, Minion [12] uses static analysis (e.g., code reachability and data reachability from entry points in the firmware) to isolate code, data, and peripherals used by each thread to create *thread level* isolated partitions. At runtime, Minion's [12] memory view switcher dynamically configures the MPU according the currently executed thread. That is, each thread can only access its code, data, and peripherals. Other isolated threads are protected by the view switcher (i.e., using the MPU access permissions). Minion [12] controls the context switch handler and is the only code executing is privileged mode, thus it is the only one accessing the MPU. Note that unlike general-purpose systems, a thread can directly access its designated MMIO, without a system call transitioning to an isolated OS. However, Minion [12] requires an OS for its isolation. For bare-metal MCUS, ACES [9] enables intra-thread isolation by using Program Dependency Graph (PDG) to create isolated *compartments* of code, data, and peripherals. The policy used to create these compartments are flexible and can be defined by the developer.

Memory safety: Enforcing memory safety enables preventing memory corruption bugs. However, traditional methods [70–75] are not suitable for MCUS as it requires heavy checking for each pointer derefence. That is, it validates spatial memory safety by validating that the derefrence is within the valid bounds of the object. In addition, the object must be valid at the time of use (i.e., temporal safety). This leads to high overhead both in runtime and memory on MCUS.

To reduce this overhead, nesCheck [76] uses both static and dynamic analysis to identify memory accesses to check. Safe accesses that are proven to be safe statically are not instrumented. For other memory accesses, nesCheck [76] *conservatively* checks ones that can lead to memory corruption, but excludes ones can only lead to memory access errors. That is, nesCheck [76] provides sound memory safety. However, nesCheck [76] is only applicable to TinyOS [77], an operating system for wireless sensor networks and its `C` programming dialect `nesC`.

Table 2.1.
Overview of MCUS defenses. MCUS type defines whether the defense was used on bare-metal systems, systems with an OS, or both. Control-flow hijacking shows whether the defense protects the return edge, forward edge, or both. Non-control data shows what security guarantees the defense applies to it. Evaluation type shows whether the defense used any available benchmark, or customized applications. Using an MCUS evaluation platform indicates whether the proposed defense was evaluated on an MCUS device.

Legend

*: Defense is applicable, but not implemented. ◔: Only detects the attack. ◑: Mitigates the attack, attacks are still possible. ●: Prevents the attack. BM: Bare-metal.

Defense	MCUS Type		Requirements	Control-Flow Hijacking		Non-control data	Used Technique										Evaluation Type		Evaluated on MCUS Platform
	BM	OS		Return edge	Forward Edge		Control-Flow Attestation	Data-Flow Attestation	Information Hiding	Execute-only Memory	Safe Stack	Stack Canary	Shadow Stack	Control-Flow Integrity	Memory Isolation	Memory and Type Safety	Benchmark	Custom	
C-FLAT [14]	✓	*	TEE	◔	◔	◔	✓											✓	
LO-FAT [49]	✓	*	Customized hardware extension	◔	◔	◔	✓											✓	✓
LiteHAX [18]	✓	*	Customized hardware extension	◔	◔	◔	✓	✓									✓	✓	✓
ATRIUM [50]	✓	*	Customized hardware extension	◔	◔	◔	✓										✓		✓
DIAT [53]	*	✓	TEE	◔ (on critical modules)	◔ (on critical modules)	◔	✓	✓							✓			✓	✓(without TEE)
nesCheck [76]		✓	Specific programming language/OS	●	●	◑										✓		✓	
SCFP [15]	✓	*	Customized hardware extension	◑	◑				✓				✓				✓	✓	✓
CFI CaRE [13]	✓	*	TEE	●	◑								✓	✓			✓	✓	✓
Minion [12]		✓		◑	◑	◑									✓			✓	✓
ACES [9]	✓	*		◑	◑	◑								✓	✓			✓	✓
Symbiote [56]	*	*		◑	◑				✓					✓				✓	
EPOXY [8]	✓	*		◑					✓		✓						✓	✓	✓
μArmor [47]	*	✓		◑					✓			✓					✓		✓
RECFISH [69]	*	✓		◑	◑								✓	✓			✓		✓
LR^2 [45]	*	*		◑	◑				✓	✓									
uXoM [46]	*	✓	Specific to Cortex-M	◑	◑				✓	✓							✓		✓

2.4 Analysis

The defenses discussed in section 2.3 offer various guarantees with different requirements. This discussion aims to systematically analyze these defenses across four dimensions: (1) hardware and defense requirements; (2) security guarantees of remote and local defenses; (3) performance overhead; (4) effectiveness against control-flow hijacking ; and (5) evaluation type and platform. Table 2.1 shows a summary of the defenses across the various dimensions.

2.4.1 Hardware and Defense Requirements

Hardware requirements: For a defense to be widely adopted, it must limit the overhead needed to apply it [26]. This overhead can be either in runtime or resources required for the defense. In addition, it is preferable for a defense to be portable. That is, it does not rely on features specific to a single architecture. This is especially true for MCUS, which are low cost constrained systems.

However, a large portion of existing defense rely on special hardware features that are not available on widely available on currently deployed MCUS. This case is especially evident for remote defenses. As discussed in subsection 2.2.2, a TEE is not widely adopted on *currently deployed* MCUS. While it is better to minimize the hardware requirements of a defense, a TEE will be incorporated in newer MCUS architectures (e.g., TrustZone [24] in ARMv8-M [78]). Thus, requiring a TEE should become less of an issue in long term.

In contrast, requiring customized hardware extension significantly reduces the chances of wide adoption of a defense. These defenses require physical changes to the actual devices or replacing the device itself. Thus, they incur high cost for deployment. In addition, MCUS deploy diverese archititetures, thus such defenses are not easily integrated with existing MCUS without significant engineering and integration efforts.

Defense assumptions: While some defenses show promising results and provide various protections (e.g., control-flow attestation), we argue that these are built on unrealistic assumptions. Particularly, remote attestation techniques [14,18,49] assume a powerful verifier, able to compute and track all dynamic control-flow traces. There is no guarantee that such a verifier exists.

In addition, some defenses relying on information hiding [56] assume no versions of the same diversified firmware exists. This does not guarantee their security as even diversified firmware can share a ROP payload across different versions [8, 47]. Moreover, such assumption are not easily integrated into deployed systems since vendors have to keep track of each diversified version (i.e., for code signing and verification).

2.4.2 Security Guarantees of Remote and Local Defenses

The security guarantees of each mechanism depends mainly on its type (i.e., remote or local). The two types can be complemented with each other as each serves a different goal. Local defenses *mitigate or protect* an attack even the presence of memory corruption vulnerabilities. Remote defense *only detect* an attack, thus they offer *weaker guarantees*. That is, the attack can always happen with remote defenses. These defenses only notify the trusted verifier of a compromised device.

Remote defenses can *detect* a wider range of attacks by leveraging a more powerful remote verifier and customized or special hardware features. For example, remote defenses enable detecting control-flow bending attacks [68] since they trace all control-flow transfer (i.e., assuming the verifier pre-computed all dynamic control-flow traces). Some defenses use hardware extensions to track data-flow events [18].

However, the security guarantees of some remote defenses are not necessarily reflected in the actual implementation or evaluation. For example, rather than attesting the entire application, C-FLAT [14] attested *critical* parts of some of the firmware used in the evaluation. This opens a larger attack surface that will not be detected by such defenses. The actual guarantees of such defenses are not verified.

We argue the reason for such variance between the assumed design and actual implementation is the complexity introduced in remote defenses (e.g., tracking all transfers for any possible control-flow hijacking attack requires unrealistic verifier capabilities). We also argue that these can reduced by leveraging the proposed local defenses.

Local defenses can mitigate, and even prevent a wide range on attacks. The local defenses shown in Table 2.1 offer sound security guarantees, and many are applicable to existing MCUS without any special hardware requirements. A remote defense can therefore be complemented with a local defense. For example, a remote attestation mechanism can rely on a shadow stack mechanism such as CFI-CaRE [13] to prevent control-flow hijacking attacks. A remote attestation mechanism can be complemented with such mechanism to reduce the overhead of tracking backward edges, and instead focus on attesting the forward edges of the control-flow, or explicitly trace annotated *sensitive data*. That is, local defenses should be a base for remote defenses to build upon.

2.4.3 Performance Overhead

Table 2.2 shows the average runtime overhead of defenses as *reported* by each defense. We limit our comparison to local defenses as remote defenses depend on the frequency of attestations (e.g., once every minute), and the portion of protected code. These defenses reported the relation (e.g., linear [14, 53]) of attested firmware (e.g., with respect to number of control-flow transfer). In addition, we exclude defenses not evaluated on MCUS (e.g., LR^2 [45]) since the reported overhead does not necessarily apply when evaluated on MCUS as was shown by uXoM [46]. For ACES [9], the runtime overhead depends on the used policy for compartmentalization. ACES evaluated three separate policies with overheads ranging between 0% to over 400%.

Overall, with the exception CFI-CaRE [13] and RECFISH [69], proposed defenses demonstrate a practical average runtime overhead lower than 10%. However, these defenses offer weaker guarantees against control-flow hijacking attacks on the return edge compared CFI-CaRE [13] and RECFISH [69] as shown in Table 2.1. In the

Table 2.2.
A summary of theaverage runtime overhead for local MCUS defenses as a % over the baseline. These results shown here are the ones *reported* from the each paper respectively.

Legend

●: Higher than 100%. ◑: Higher than 10%, but below 50%. ○: Lower than 10%.

Defense	**Runtime Overhead**
SCFP [15]	○
CFI CaRE [13]	●
Minion [12]	○
ACES [9]	(overhead depends on application and applied policy)
EPOXY [8]	○
μArmor [47]	○
RECFISH [69]	◑
uXoM [46]	○

following section, we discuss the effectiveness of each defense in mitigating control-flow hijacking attacks.

2.4.4 Effectiveness Against Control-Flow Hijacking Attacks

As mentioned in subsection 2.4.2, remote defenses only *detect* an attack. That is, an attack can occur as in a normal application. Remote defenses also rely on the attestation frequency and time, making them vulnerable to TOCTOU attacks [79], with the exception of ATRIUM [50]. In addition, if the attack is detected, remote attacks only notify the trusted verifier. The verifier can then take additional measures (e.g., reset the device or disable it), these however are our of scope of this work. Our focus is on the strength of a mechanism to mitigate or prevent an attack, thus for our purpose remote attacks offer weaker guarantees than local defenses. Overall, the proposed mitigations shown in Table 2.1 improve the security of MCUS, however they do not prevent control-flow hijacking on MCUS.

Information hiding techniques, including ones relying on XoM, have been shown to be vulnerable on general-purpose systems to various information disclosure [57,58] and profiling attacks [80]. MCUS defenses using similar techniques [8,15,46,47,56] are even more susceptible to such attack as they have much lower entropy due to the small memory available. The same applies for applying a stack canary for MCUS [47], which is a weaker version of stack canaries applied on general-purpose systems.

Memory isolation mechanisms [9,12] only confine the vulnerability to the current isolated memory region. That is, the attacker can still divert the control-flow anywhere within the current isolated memory region.

Some defenses provide stronger guarantees against certain control-flow hijacking scenarios. EPOXY [8] adoption of a safe stack eliminates stack based buffer overflow exploitation on the return edge. It remains however vulnerable to other attack scenarios (e.g., arbitrary write). RECFISH [69] eliminates such attacks in unprivileged mode by placing a shadow stack in a privileged region. CFI-CaRE relies on a TEE to prevent attacks on the return edge both in privileged and privileged mode. However, both incur high runtime overhead as was shown in Table 2.2.

For forward edge protection, existing defenses support a coarse-grained CFI. For example, CFI-CaRE [13] allows indirect calls to target any function entry. Symbiote [56] randomly checks a portion of indirect calls. The most precise forward edge CFI is applied by ACES [9], which enforces a type-based CFI across isolated memory regions. Finally, nesCheck [76] offers promising guarantees, however its results were simulated and have not been verified on actual MCUS. Furthermore, it is specific to TinyOS [77] and the `nesC` programming language, which specific to wireless sensor network systems. MCUS however predominately use `C`.

2.4.5 Evaluation Type and Platform

Although the defenses shown in Table 2.1 target MCUS, the evaluation used for some defenses reveals a subtle issue. First, some defenses while applicable to MCUS, did not use an MCU platform to evaluate the defense [13], or used a platform lacking a required feature for the defense [53]. Thus, a conclusive judgement of the defense's performance cannot be made since the difference in evaluation platform can alter the performance of the defense. For example, uXoM [46] showed that applying a software-based XoM [81] resulted in significantly higher runtime overhead when evaluated on MCUS (i.e., 22.7% on MCUS compared to 5.0% on general-purpose systems [46]).

More importantly, the evaluation type (i.e., software application) is not standardized between the various defenses. A large number of defenses used customized applications, without any benchmark. Even for defenses using a benchmark, they often utilize different benchmarks. Furthermore, these benchmark are often simple, and do not reflect a realistic applications of IoT-MCUS (i.e., MCUS in an IoT setting). We discuss this in further detail in subsection 2.5.1.

2.5 Discussion and Future Research

Following our analysis of proposed defenses, we identify two main issues in MCUS security. We provide a discussion of each and our proposed suggestion for future research.

2.5.1 Benchmarking and Evaluation Frameworks

Experimental evaluation is essential for the progress of any field. However, existing MCUS defense evaluation suffers from multiple limitations. First, the evaluation process is tedious, relying heavily on hardware extensions and underlying boards, thus hindering researchers efforts. Second, existing benchmarks are too simple to evaluate the security guarantees of proposed defenses. Realistic MCUS applications in an IoT setting interact with a rich set of sensors and actuators. However, existing benchmarks remove such interactions with peripherals to ensure portability since peripherals are mapped differently in memory between different MCU manufacturers. Furthermore, the software APIs are different between the different MCUS board vendors. That is, an application must be written separately for each vendor, thus requiring significant engineering effort. This resulted in an ad-hoc evaluation of MCUS defenses, and a *quantitative* comparison between the different defense mechanisms became infeasible.

We propose developing an evaluation framework following a software-based approach. That is, it can be used by any MCUS sharing the same architecture. In addition, we propose that such a framework incorporate standardized metrics covering both security and performance metrics for MCUS. Developing such a framework to automate the evaluation process would enable researchers to effectively evaluate and compare the proposed defenses. Lastly, we propose developing a standardized benchmark applications mimicking IoT-MCUS, with rich interactions with peripherals and demonstrating networking capability as assumed in an IoT setting. The benchmarks must be built to be portable across various board manufactures with limited engineering efforts.

2.5.2 Control-Flow Hijacking Protection

Overall, the mitigations discussed in subsection 2.4.4 have a trade-off between performance and stronger security guarantees. That is, proposed defenses are either impractical or have limited security guarantees. Although proposed defenses enhance the state of MCUS security, they are still vulnerable to control-flow hijacking attacks.

For forward edge protection, a starting step is to utilize more precise analysis for CFI implementations. MCUS architectures are not supported by known CFI implementations [82], thus proposed defenses used a course-grained CFI (e.g., all function entries are in the allowed target set). Future research can apply stronger and more precise analysis [83] to reduce the target set of CFI implementations. Since MCUS have smaller code than general-purpose systems, the valid target set should be smaller, thus resulting in strong guarantees for the forward edge.

Protecting backward edges (i.e., return addresses) is more challenging for MCUS. Although some state-of-the-art solutions have been adopted to MCUS, they remain vulnerable to attacks or are impractical. Return addresses are more prevalent and are a more vulnerable target without strong defenses. Since MCUS demonstrate unique characteristics (e.g., rare use of recursion), a proposed defense might utilize such characteristics in designing specific defenses for MCUS to prevent attacks on return edges while maintaining low overhead.

2.6 Conclusion

Embedded and IoT system running MCUS are deployed in security critical domains. Unfortunately, MCUS are vulnerable to memory corruption attacks due the combination of constrained resources, use of low level languages, and lack strong security mechanisms. We surveyed proposed defenses for MCUS, and identified two main issues for MCUS security. First, the lack of standardized benchmarks and evaluation frameworks. Second, proposed defenses either impose substantial runtime overhead or have limited security guarantees against control-flow hijacking attacks on the return edge. In the following chapters, we address both concerns.

3. BENCHIOT: A SECURITY BENCHMARK FOR THE INTERNET OF THINGS

Attacks against IoT systems are increasing at an alarming pace. Many IoT systems are and will be built using low-cost micro-controllers (IoT-MCUS). Different security mechanisms have been proposed for IoT-MCUS with different trade-offs. To guarantee a realistic and practical evaluation, the constrained resources of IoT-MCUS require that defenses must be evaluated with respect to not only security, but performance, memory, and energy as well.

Evaluating security mechanisms for IoT-MCUS is limited by the lack of realistic benchmarks and evaluation frameworks. This burdens researchers with the task of developing not only the proposed defenses but applications on which to evaluate them. As a result, security evaluation for IoT-MCUS is limited and ad-hoc. A sound benchmarking suite is essential to enable robust and comparable evaluations of security techniques on IoT-MCUS.

This chapter introduces BenchIoT, a benchmark suite and evaluation framework to address pressing challenges and limitations for evaluating IoT-MCUS security. The evaluation framework enables automatic evaluation of 14 metrics covering security, performance, memory usage, and energy consumption. The BenchIoT benchmarks provide a curated set of five real-world IoT applications that cover both IoT-MCUS with and without an OS. We demonstrate BenchIoT's ability by evaluating three defense mechanisms. All benchmarks and the evaluation framework is open sourced and available to the research community [1].

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

3.1 Introduction

Experimental evaluation is integral to software systems research. Benchmarks play a pivotal role by allowing standardized and comparable evaluation of different software solutions. Successful benchmarks are realistic models of applications in that particular domain, easy to install and execute, and allow for collection of replicable results. Regrettably, there is no compelling benchmark suite in the realm of Internet of Things (IoT) applications, specifically in those that run on low-end platforms with either no operating system as a single binary image or with a lightweight OS like ARM's Mbed-OS [29]. As IoT applications become more ubiquitous and are increasingly used for safety-critical scenarios with access to personal user data, security solutions will take center stage in this domain. Therefore, IoT benchmarks will also be needed to evaluate the strength of the security provided by the security solutions.

The IoT domain that we target has some unique characteristics, which make it challenging to directly apply existing benchmarks either from the server world or even the embedded world, to our target domain. These IoT systems run on low-end micro-controllers (MCUS), which have frequencies of the order of tens to a few hundreds of MHz's, e.g., ARM's 32-bit Cortex-M series. They have limited memory and storage resources, of the order of hundreds of KBs and a few MBs respectively. These applications typically have tight coupling with sensors and actuators that may be of diverse kinds, but using standard interfaces such as UART and SPI. Finally, the applications have the capability for networking using one or more of various protocols. In terms of the software stack that runs on these devices, it is either a single binary image that provides no separation between application and system level (and thus is a "bare-metal" or no OS system) or has a light-weight real time OS (e.g., ARM's Mbed-OS), which supports a thin application-level API. We refer to our target domain for the remainder of the chapter as IoT-MCUS.

Existing benchmarks from the server world are not applicable because they do not reflect applications with characteristics mentioned above and frequently rely on functionality not present on IoT-MCUS. For example, SPEC CPU2006 [19] targets desktop systems and requires e.g., standardized I/O. Many IoT applications on the other hand have non-standard ways of interacting with IO devices such as through memory-mapped IO. In addition, their memory usage is in the range of hundreds of MBs [84]. Several benchmarks [21–23, 85] are designed specifically for comparing performance on MCUS. However, they do not exercise the network connectivity and do not interact with the physical environment in which the devices may be situated (i.e., they do not use peripherals). Moreover, these benchmarks lack the complexity and code size of realistic applications and as result make limited use of relatively complex coding constructs (e.g., call back event registration and triggering). From a security perspective, control-flow hijacking exploits rely on corrupting code pointers, yet these benchmarks make limited use of code pointers or even complex pointer-based memory modification. Thus, they do not realistically capture the security concerns associated with IoT-MCUS.

The lack of security benchmarks for IoT applications inhibits disciplined evaluation of proposed defenses and burdens researchers with the daunting task of developing their own evaluation experiments. This has resulted in ad-hoc evaluations and renders comparison between different defenses infeasible as each defense is evaluated according to different benchmarks and metrics. Table 3.1 compares the evaluations of several recent security mechanisms for IoT-MCUS, and only two of them use the same benchmarks to evaluate their defenses, and even these two target different architectures, making a comparison hard. Out of all the defenses, only four used any benchmarks at all and they were from the embedded world and not representative of IoT applications as identified above. The other solutions relied solely on micro-benchmarks and case studies. These are unique to the individual papers and often exercise only a single aspect of a realistic application (e.g., writing to a file).

Requirements for IoT benchmarks.

Benchmarks for IoT-MCUS must meet several criteria. First, the applications must be realistic and mimic the application characteristics discussed above. While an

Table 3.1.
A summary of defenses for IoT-MCUS with the evaluation type.

Defenses	**Evaluation Type**	
	Benchmark	Case Study
TyTan [16]		✓
TrustLite [17]		✓
C-FLAT [14]		✓
nesCheck [76]		✓
SCFP [15]	Dhrystone [22]	✓
LiteHAX [18]	CoreMark [85]	✓
CFI CaRE [13]	Dhrystone [22]	✓
ACES [9]		✓
Minion [12]		✓
EPOXY [8]	BEEBS [23]	✓

individual benchmark need not satisfy *all* characteristics, the set of benchmarks in a suite must cover all characteristics. This ensures security and performance concerns with real applications are also present in the benchmarks. IoT devices are diverse, therefore the benchmarks should also be diverse and cover a range of factors, such as types of peripherals used, and being built with or without an OS. Finally, network interactions must be included in the benchmarks.

Second, benchmarks must facilitate repeatable measurements. For IoT applications, the incorporation of peripherals, dependence on physical environment, and external communication make this a challenging criterion to meet. For example, if an application waits for a sensed value to exceed a threshold before sending a communication, the time for one cycle of the application will be highly variable. The IoT-MCUS benchmarks must be designed to both allow external interactions while enabling repeatable measurements.

A third criterion is the measurement of a variety of metrics relevant to IoT applications. These include performance metrics (e.g., total runtime cycles), resource usage metrics (e.g., memory and energy consumption), and domain-specific metrics (e.g., fraction of the cycle time the device spends in low-power sleep mode). An important goal of our effort is to enable benchmarking of IoT security solutions and hence the benchmarks must enable measurement of security properties of interest. There are of course several security metrics very specific to the defense mechanism but many measures of general interest can also be identified, such as the fraction of execution cycles with elevated privilege ("root mode") and number of Return-Oriented Programming (ROP) gadgets.

Our Contribution: BenchIoT

This chapter introduces the BenchIoT benchmark suite and evaluation framework that fulfills all the above criteria for evaluating IoT-MCUS. Our benchmark suite is comprised of five realistic benchmarks, which stress one or more of the three fundamental task characteristics of IoT applications: sense, process, and actuate. They also have the characteristics of IoT applications introduced above. The BenchIoT bench-

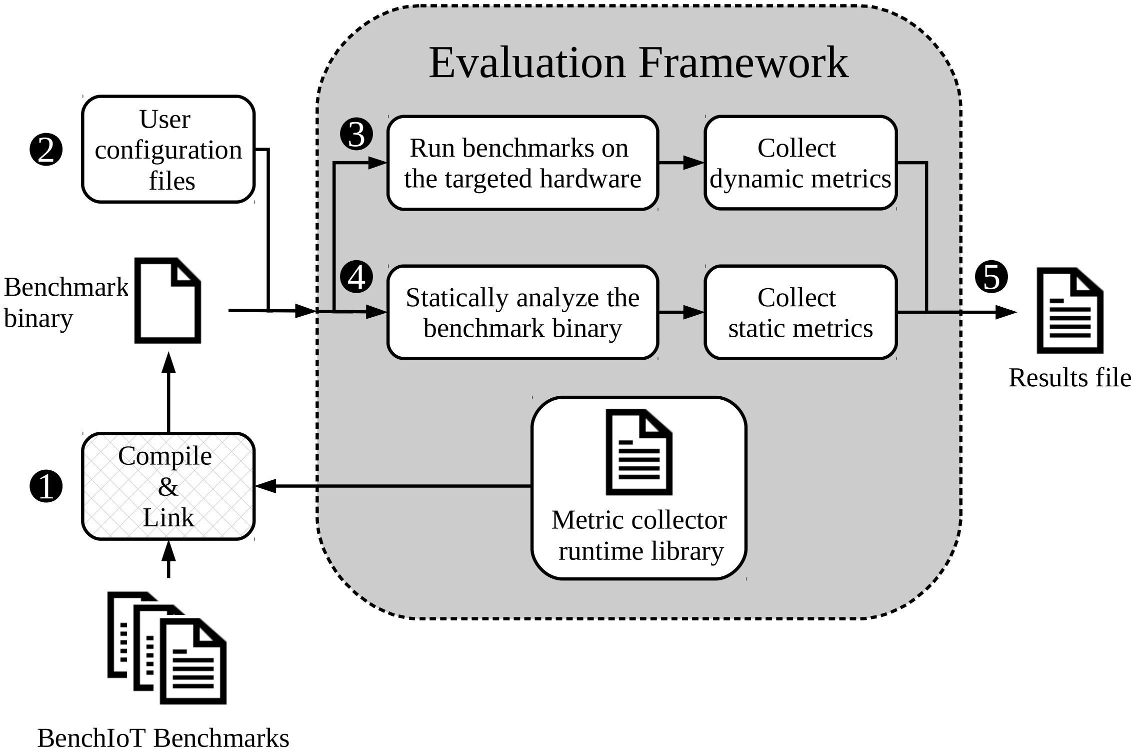


Fig. 3.1. An overview of the evaluation workflow in BenchIoT.

marks enable deterministic execution of external events and utilize network send and receive. BenchIoT targets 32-bit IoT-MCUS implemented using the popular ARMv7-M architecture. Each BenchIoT benchmark is developed in C/C++ and compiles both for bare-metal IoT-MCUS, and for ARM Mbed-OS. Our use of the Mbed API (which is orthogonal to the Mbed-OS) enables realistic development of the benchmarks since it comes with important features for IoT-MCUS such a file system.

BenchIoT enables repeatable experiments while including sensor and actuator interactions. It uses a software-based approach to trigger such events. The software-based approach enables precise control of when and how the event is delivered to the rest of the software without relying on physical environment. This approach has been used previously for achieving repeatability as a means to automated debugging [86,87].

BenchIoT's evaluation framework enables automatic collection of 14 metrics for security, performance, memory usage, and energy consumption. The evaluation framework is a combination of a runtime library and automated scripts. It is extensible to include additional metrics to fit the use of the developer and can be ported to other applications that use the ARMv7-M architecture. An overview of BenchIoT and the evaluation framework is shown in Figure 3.1. The workflow of running any benchmark in BenchIoT is as follows: (1) The user compiles and statically links the benchmark with a runtime library, which we refer to as the *metric collector library*, to enable collecting the dynamic metrics ❶; (2) The user provides the desired configurations for the evaluation (e.g., number of repetitions) ❷; (3) To begin the evaluation, the user starts the script that automates the process of running the benchmarks to collect both the dynamic ❸ and static ❹ metrics; (4) Finally, the benchmark script produces a result file for each benchmark with all its measurements ❺.

To summarize, this chapter makes the following contributions: (1) This is the first realistic benchmark suite for security and performance evaluation of IoT-MCUS. It enables the evaluation of IoT-MCUS with realistic benchmarks representing characteristics of IoT applications such as connectivity and rich interactions with peripherals; (2) It enables out-of-the-box measurements of metrics for security, performance, memory usage, and energy consumption; (3) It provides a deterministic method to simulate external events enabling reproducible measurements; (4) It demonstrates the effectiveness of BenchIoT in evaluating and comparing security solutions where we apply three standard IoT security defenses to the benchmarks and perform the evaluation. Our evaluation brings out some hitherto unreported effects, such as, even though defense mechanisms can have similarly modest runtime overhead, they can have significantly different effects on energy consumption for IoT-MCUS depending on their effect on sleep cycles. The benchmark suite along with the evaluation scripts is open sourced and available to the research community [88].

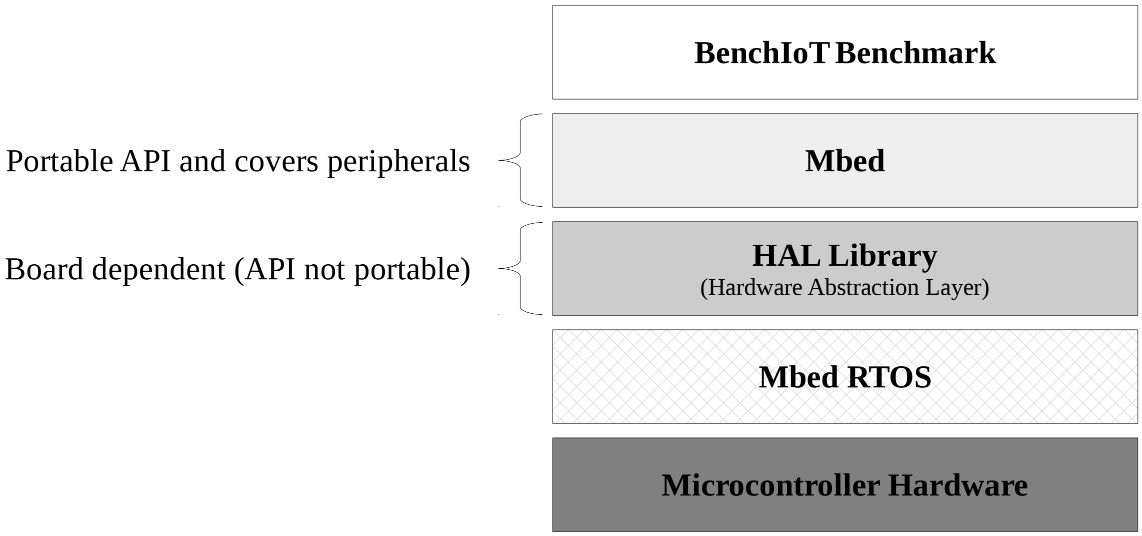

BenchIoT Benchmark
Mbed
HAL Library
(Hardware Abstraction Layer)
Mbed RTOS
Microcontroller Hardware
Portable API and covers peripherals
Board dependent (API not portable)

Fig. 3.2. Illustration of software layers used in developing BenchIoT benchmarks. BenchIoT provides portable benchmarks by relying on the Mbed platform.

3.2 Scoping and Background

3.2.1 Scoping and Target Systems

The goal of this work is to enable security evaluation and comparison for different security defenses on IoT-MCUS devices through: (1) comprehensive, automatically measured metrics and (2) benchmark suite representing realistic IoT applications. It is *not* the goal of this work to propose new security mechanisms. However, we believe that our benchmark suite will be vital for continued innovation and reproducibility of security research on IoT-MCUS.

We define an IoT-MCU device as an embedded system that executes software on a microcontroller (μC), and has network connectivity. That is, the notion of *device* includes the μC and the standard peripherals packaged on the same board. As such, all of BenchIoT's benchmarks utilize IP communication. MCUS have clock speeds of a few MHz topping out under 200MHz, unlike higher-end embedded systems (e.g., ARM Tegra 2) which operate at clock speeds in the range of GHz. Our target systems have a few hundreds KBs of RAM and few MBs of Flash. These constraints mean they have limited software executing on them. It is common practice to have these devices run a single application in a dedicated mode and therefore all our benchmarks also provide a single functionality. They operate either with a lightweight Real Time Operating System (RTOS), enabling multiple threads of execution, or a single threaded application without an OS (i.e., bare-metal). In both cases, a single statically linked binary is the only code that executes on the system.

The MCUS typically lack security hardware commonly available on server-class systems (e.g., MMUs). However, they commonly have a Memory Protection Unit (MPU) [89]. A MPU enforces read, write, and execute permission on physical memory locations but does not support virtual memory. The number of regions an MPU supports is typically quite small (8 in the ARM v7-M architectures). MPUs in general support two privilege levels (i.e., privileged and unprivileged). These differences in capabilities and software development make many security mechanisms for desktop systems inapplicable for IoT-MCUS (e.g., ASLR). ASLR relies on virtual memory to randomize the layout of the application.

To implement the benchmarks and demonstrate rich and complex IoT-MCUS applications, BenchIoT targets 32-bit IoT-MCUS using the ARM Cortex-M(3,4,7) μCs, which are based on the ARMv7-M architecture [40]. ARM Cortex-M is the most popular μC for 32-bit MCUS with over 70% market share [43, 44]. This enables the benchmarks to be directly applicable to many IoT devices being built today. As shown in Figure 3.2, hardware vendors use different HAL APIs depending on the underlying board. Since ARM supplies an ARM Mbed API for the various hardware boards, we rely on that for portability of BenchIoT to all ARMv7-M boards. In addition, for applications requiring an OS, we couple those with Mbed's integrated RTOS—which is referred to as Mbed-OS. Mbed-OS allows additional functionality such as scheduling, and network stack management. To target other MCUS, we will have to find a corresponding common layer or build one ourselves—the latter is a significant engineering task and open research challenge due to the underlying differences between architectures.

3.2.2 Background

Cortex Microcontroller Software Interface Standard: The Cortex Microcontroller Software Interface Standard [90] (CMSIS) is a standard API in `C` provided by ARM to access the Cortex-M registers and low level instructions. CMSIS is portable across Cortex-M processors and is the recommended interface by ARM. Note that unlike Mbed, CMSIS does not cover peripherals (e.g., UART). Mbed however uses CMSIS to access Cortex-M registers.

Privilege modes: ARMv7-M supports two privilege levels: (1) privileged mode, where all memory regions are accessible and executable. Exception handlers (e.g., interrupts, system calls) always execute in privileged mode. (2) user mode, where only unprivileged regions are accessible depending on the MPU access control configuration. To execute in privileged mode, unprivileged code can either execute Supervisor call (SVC), a system call in ARMv7-M, or be given elevated privileges through the system's software.

Software Trigger Interrupt Register: The STIR register provides a mechanism to trigger external interrupts through software. An interrupt is triggered by writing the interrupt number to the first nine bits of STIR. BenchIoT utilizes the STIR register to ensure reproducibility of experiments and avoid time variations of external interrupts arrival.

Data Watchpoint and Trace Unit: ARM provides the Data Watchpoint and Trace (DWT) unit [40] for processor and system profiling. It has a 32-bit cycle counter that operates at the system clock speed. Thus, BenchIoT uses it for making runtime measurements in the system.

3.3 Benchmark Metrics

The goal of the BenchIoT metrics is to enable quantifiable evaluation of the security and practicality of proposed defenses for IoT-MCUS. While security defenses are diverse and use various metrics to evaluate their effectiveness, the metrics proposed by BenchIoT are chosen based on the following criteria: (1) enable evaluating established security principles for IoT-MCUS (e.g., principle of least privilege); (2) enable evaluating performance effects of defenses on IoT-MCUS.

BenchIoT provides 14 metrics spanning four categories, namely: (1) Security; (2) Performance; (3) Memory; (4) Energy. Figure 3.3 shows a summary of the metrics. We note that metrics cannot cover all attack and security aspects. BenchIoT is designed to provide a generalized framework for researchers. Thus, we avoid metrics specific to a security technique.

3.3.1 Security Metrics

Total privileged cycles: An important challenge in securing IoT-MCUS is reducing the number of privileged cycles. Privileged execution occurs during (1) Exception handlers and (2) User threads with elevated privileges. By default, applications on IoT-MCUS execute completely in privileged mode. We measure the total number of execution cycles in privileged mode. A lower number is better for the security of the system.

Privileged thread cycles: Though measuring total privileged cycles can help assess the security risks of the system, they include exception handlers that are asynchronous and may not always be executable directly by a malicious adversary. However, a malicious user can exploit privileged thread cycles as these occur during the normal execution of the user code. Thus, this metric is a direct quantification of security risks for normal application code.

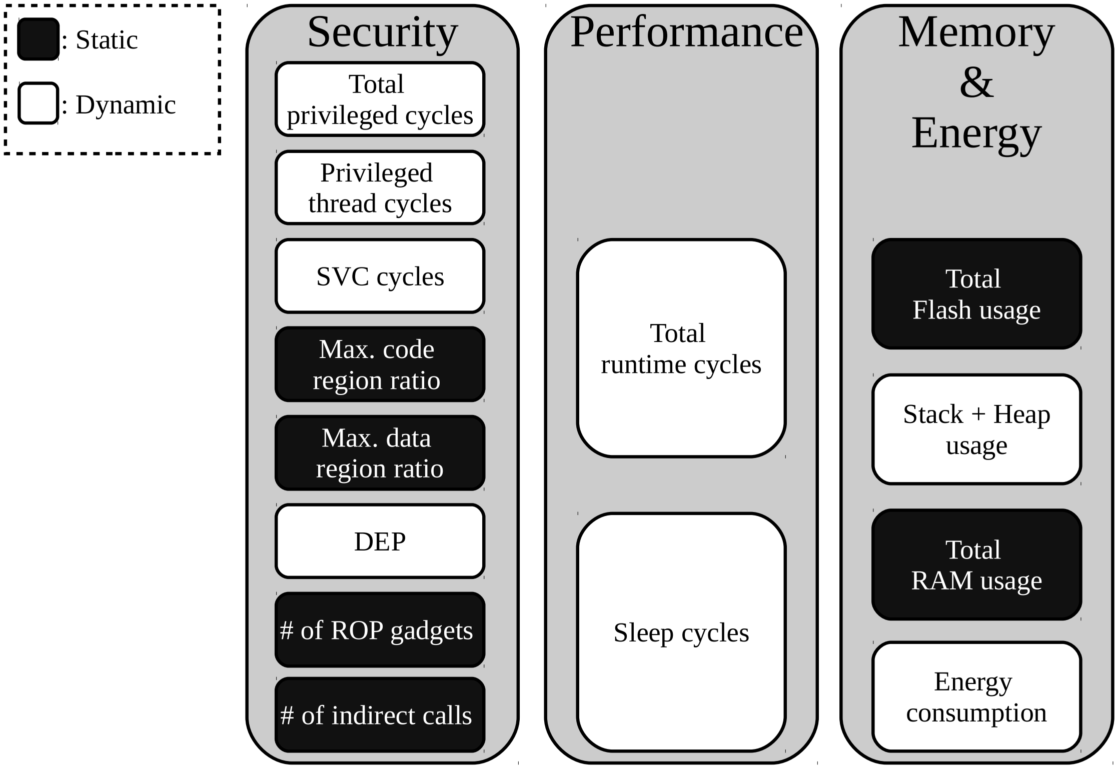


Fig. 3.3. A summary of the BenchIoT metrics.

SVC cycles: We single out SVC cycles (i.e., system call cycles) from other exception handlers as these are synchronous and can be called by unprivileged threads to execute privileged code, therefore is a possible attack surface.

Maximum code region ratio: Control-flow hijacking like code reuse attacks depend on the available code to an attacker to reuse. Memory isolation limits an attacker's capabilities by isolating segments of code within the application. This metric aims to measure the effectiveness of memory isolation by computing the size ratio of the maximum available code region to an attacker with respect to the total code size of application binary. A lower value is better for security.

Maximum global data region ratio: Another attack vector are data-only attacks [51, 52]. Such attacks target sensitive data of the application rather than hijacking the control-flow. These sensitive data can be security critical and lead to command injection or privilege escalation (e.g., by setting a boolean `is_admin` to true). This metric aims to measure the effectiveness of data isolation by computing the size ratio of the maximum available global data region to an attacker with respect to the total data section size of the application binary. A lower value is again better for security.

Data Execution Prevention: DEP is a well-known defense mechanism to stop code injection attacks by making memory regions either writable (data) or executable (code), but not both. Unfortunately, DEP is not commonly deployed in IoT-MCUS [8, 12]. As a result, DEP has been added to the BenchIoT evaluation framework to raise awareness of such an important defense for IoT-MCUS developers. This metric requires using the BenchIoT API when changing the MPU configuration to validate DEP is always enforced (i.e., every MPU region always enforces $\mathtt{W} \oplus \mathtt{X}$).

Number of available ROP gadgets: Return Oriented Programming (ROP) [55] is a type of code reuse attack used to hijack the control-flow of the application. It is performed by exploiting a memory corruption vulnerability to chain existing code snippets–that end with a return instruction (i.e., ROP gadgets)–together to perform arbitrary execution. Hence, reducing the number of possible ROP gadgets reduces the attack surface available to the attacker, and helps quantify the effectiveness of defense mechanisms such as randomization or CFI. The number of ROP gadgets is measured using the ROPgadget compiler [91] on the benchmark binary.

Number of indirect calls: Control-flow hijacking occurs through corrupting indirect control-flow transfers. ROP resiliency covers control-flow hijacking of backward edges (i.e., function returns). Another attack scenario is using indirect calls (i.e., forward edges). Thus, reducing the number of indirect calls (e.g., function pointers) or protecting them through control-flow hijacking defenses like Control-Flow Integrity (CFI) [65] is a desired security property. We measure the number of indirect calls by parsing the benchmark binary.

3.3.2 Performance Metrics

Total runtime cycles: This metric measures the total runtime of a benchmark from establishing a remote connection to the end. Including execution cycles prior to establishing a connection results in variance in measuring the benchmarks (e.g., wait for client) and thus they are excluded. While many IoT-MCUS run forever, defining a start and an end to a benchmark is important to enable analyzing the impact of security mechanisms on the overall performance.

Sleep cycles: Popular embedded OSes [29, 30] use a sleep manager that automates entering sleep mode to minimize energy consumption of IoT-MCUS. For example, the application can enter sleep mode while blocking on network receive or writing a file to uSD card. Since energy consumption during sleep mode is typically two to three orders of magnitude lower compared to active or idle mode [92], many IoT applications spend a large portion of their execution in sleep mode. A security technique can reduce sleep cycles and this is important to capture to measure the effects on energy conservation.

3.3.3 Memory and Energy Metrics

Total Flash usage: Measures the application code and read-only data.

Stack and heap usage: This metric enables analyzing the overhead of memory used at runtime. The obtained measurement is the maximum usage of the stack and the heap.

Total RAM usage: In addition to the stack and heap usage, this metric measures the statically allocated data sections by parsing the benchmark binary image.

Total energy consumption: This metric is measured physically (e.g., with a logic analyzer) and is the only metric in BenchIoT that depends on external hardware. The user connects a current sensor to the micro-controller power supply in series to measure the power. The energy is calculated by multiplying the average power with total runtime. A General Purpose Input Output (GPIO) signal is instrumented to signal the beginning and the end of the measurement.

3.4 Benchmark Design

To develop a realistic benchmarking suite for IoT-MCUS, we designed BenchIoT to satisfy the following criteria: (1) Enables deterministic execution of external events; (2) Performs different types of tasks to increase coverage of application behaviors and domains; (3) Utilizes various peripherals; (4) Maximizes portability and reproducibility across hardware vendors; (5) Minimizes the effect of surrounding environment on network connectivity. In the next sections, we provide the explanation and need for each dimension.

3.4.1 Deterministic Execution Of External Events

Including external events (e.g., user pushing a button) is necessary to represent realistic IoT-MCUS applications. However, these external events lead to a wide variance across benchmarks, thus rendering the evaluation non-repeatable. An important aspect in the design of BenchIoT is that it allows deterministic execution of external events. We define external events as any interrupt caused by an action not performed by the underlying device. For example, sending a pin code for a smart locker application from a nearby PC is considered an external event. Note that this mechanism does not cover network functionality (e.g., `send`, `recv`) since these must demonstrate actual connection with a remote client to represent an IoT application.

External events execute as interrupt requests, thus, BenchIoT leverages the `STIR` register to trigger the interrupt at specific points in the benchmark. The Interrupt Request (IRQ) is triggered by writing the IRQ number to the `STIR` register at specific points of the benchmark. Thus, instead of having a variance in the waiting time to enter a pin code, the interrupt is triggered at a specific point, enabling reproducibility.

Triggering the interrupt in software allows the BenchIoT benchmarks to control the IRQ execution and the input dataset. The device may execute the IRQ on the hardware and after finishing, the benchmark overwrites the provided value with the read value from the dataset that is being used to drive the experiment. For example, developers may use different temperature sensors in different settings. After the triggered IRQ executes, BenchIoT replaces the measured value with the read temperature to make the benchmark execution independent of the physical environment.

3.4.2 Application Characteristics

To increase the coverage of IoT-MCUS application domains, the BenchIoT benchmarks were designed to have characteristics typical of IoT applications. The characteristics can be categorized into three classes: (1) Sensing: the device *actively* records sensory data from one or more on-board sensors; (2) Processing: the device performs some computation or analysis (e.g., authentication); (3) Actuation: the device performs some action based on sensed data and local processing or remote processing. A benchmark may perform one or more of these task types.

The attack surface is influenced by the type of the task. For example, applications with sensing tasks (e.g., smart meter) often sample physical data, and thus their communication might be limited since they act as sensor nodes to a more powerful server that aggregates and analyzes their data. However, data tampering becomes a prominent attack vector in such applications. An example of such an attack is tampering smart meter data to reduce electricity bills [4]. An attack on an application with actuation can impose a cyber-physical hazard. For example, an attack hijacking the control-flow of an industrial robot poses physical risks to humans [93].

3.4.3 Peripherals

The BenchIoT benchmarks are designed to include various peripherals to represent realistic interactions of IoT-MCUS. In addition, peripherals can increase the attack surface for an application [94, 95], and thus their security evaluations differ. For example, attacks against autonomous vehicles target vulnerabilities in the Controller Area Network (CAN) bus [96]. The runtime performance is also effected by these peripherals. For example, a μSD that uses the Secure Digital Input Output (SDIO) results in faster data transfer than the Synchronous Peripheral Interface (SPI) since SDIO utilizes more data lines.

While BenchIoT is designed to stress as many peripherals as possible, we focus on the most commonly available peripherals across different hardware targets to allow portability. These are; UART/USART, SPI, Ethernet, timers, GPIO, Analog-to-Digital Converter (ADC), Real-time Clock (RTC), and Flash in-application programming (IAP). In addition, in case a non-common peripheral is used in a benchmark (e.g., Display Serial Interface (DSI)) and such peripheral is not present on the developer targeted board, BenchIoT allows the developer to configure and still run the main benchmark while excluding the missing peripheral.

3.4.4 Portability

BenchIoT aims to enable security evaluation for IoT-MCUS across a wide set of hardware platforms. Since IoT-MCUS cover both systems with and without an OS, we develop BenchIoT to support both. Therefore, the BenchIoT benchmarks were developed in `C/C++` using the popular Mbed platform [29,97]. Unlike other RTOSs [98], Mbed is integrated with essential features for the IoT "things" (e.g., networking, cryptographic library) and allows developing benchmarks for systems with an RTOS as well as bare-metal systems. As shown earlier in Figure 3.2, Mbed provides an abstraction layer above each vendor's HAL library, thus allowing benchmarks to be portable across the various ARMv7-M targets supported by Mbed.

3.4.5 Network Connectivity

In keeping with the fundamental characteristic of IoT applications that they perform network communication, we design our benchmarks to do network send and receive. However, wireless communication can introduce significant non-determinism in the execution of a benchmark. Therefore, BenchIoT uses Ethernet as its communication interface since it maintains the application's IoT-relevant characteristic to remain unchanged, while minimizing noise in the measurements.

3.5 Benchmark Applications

In this section we describe the BenchIoT benchmarks and highlight the notable features of IoT applications that each demonstrates. Table 3.2 shows the list of BenchIoT benchmarks with the task type and peripherals it is intended to stress. While the bare-metal benchmarks perform the same functionality, their internal implementation is different as they lack OS features and use a different TCP/IP stack. For the bare-metal applications, the TCP/IP stack operates in polling mode and uses a

Table 3.2.
A summary of BenchIoT benchmarks and their categorization with respect to task type, and peripherals.

Benchmark	**Task Type**			**Peripherals**
	Sense	Process	Actuate	
Smart Light	✓	✓	✓	Low-power timer, GPIO, Real-time clock
Smart Thermostat	✓	✓	✓	Analog-to-Digital Converter (ADC), GPIO, μSD card
Smart Locker		✓	✓	Serial(UART/USART), Display, μSD card, Real-time clock
Firmware Updater		✓	✓	Flash in-application programming
Connected Display		✓	✓	Display,μSD card

different code base. As a result the runtime of bare-metal and OS benchmarks are different.

Smart Light: This benchmark implements a smart light that is controlled remotely by the user. The user can also send commands to automate the process of switching the light on and off to conserve energy. Moreover, the smart light periodically checks if a motion was detected, and if no motion is detected it turns the light off to conserve energy. Conversely, it will turn on once a motion is detected. From a performance prescriptive, the benchmark demonstrates an event-driven application that will spend large portion of cycles in sleep mode, wake up for short periods to perform a set of tasks. It is important to measure the energy overhead, which may happen due to reduction of the sleep cycle duration. From a security perspective, attacks on smart light can spread to other smart lights and cause widespread blackout [99].

Smart Thermostat: The smart thermostat enables the user to remotely control the temperature and inquire about the current temperature of the room. In addition, the device changes temperature when the desired temperature is requested through a UART peripheral, with the display showing the responses from the application. Temperature monitoring is a common part of industrial applications (e.g., industrial motors monitoring [100, 101]). Attacks on a smart thermostat can target corrupting sensitive data (e.g., temperature value), thus leading to physical risks (e.g., overheating motor) or can use the compromised device as a part of botnet [102].

Smart Locker: This benchmark implements a smart mail locker, such as for large residential buildings. The benchmark demonstrates delivering and picking up the mail from various lockers. Upon delivering a package to one of the lockers, a random pin is generated and is sent to the server to notify the user. The device only saves the hash of the random pin to compare it upon picking up a package. Moreover, the benchmark maintains a log file of actions (i.e., pickup/drop package). The server also sends commands to inquire if the smart locker contain a certain package. The user uses the serial port to enter the input (e.g., random pin), and the application uses a display (if present) to communicate with the user. In addition, the benchmark uses a cryptographic library and stores sensitive data (e.g., hashes of generated pins). This is an event-driven benchmark.

Firmware Updater: This benchmark demonstrates a remote firmware update. On power up, the firmware updater starts a receiver process. It receives the firmware size followed by the firmware, after writing the firmware to flash it is executed. Practical security mechanisms need to be compatible with firmware updates, as vulnerabilities targeting firmware updates have been a target of attacks [103].

Connected Display: The connected display receives a set of compressed images from a remote server. It decompresses the images and shows them on the display. It also saves each image to file. The large code present in such application (e.g., networking library, image compression library) makes measuring the maximum code region ratio and ROP resiliency more relevant.

3.6 Evaluation Framework

The goal of the evaluation framework is: (1) to enable measuring the metrics explained in section 3.3; (2) to automate the evaluation process of the BenchIoT benchmarks.

Automating the evaluation of IoT-MCUS is important since evaluating IoT-MCUS is often a tedious task as it relies on manual measurements. Another option is the use a commercial hardware debugger. To avoid the limitations of both options, the BenchIoT framework follows a software based approach to collect its metrics. That is, BenchIoT does not require any extra hardware to collect its metrics (except for the energy measurement). This is achieved by only relying on the ARMv7-M architecture.

The BenchIoT evaluation framework consists of three parts: (1) a collection of `Python` scripts to automate running and evaluating the benchmarks; (2) a collection of `Python` scripts to measure the static metrics; (3) a runtime library—which we refer to hereafter as the *metric collector library*—written in C, that is statically linked to every benchmark to measure the dynamic metrics.

3.6.1 Static Metrics Measurements

We collect the static metrics explained in Figure 3.3 by parsing the binary image of each benchmark. To collect static RAM usage and Flash usage we use the `size` utility and the results are measured according to the address space and name of the region.

Unlike the previous static metrics, security static metrics require different tools. First, for the number of ROP gadgets we use the ROP gadget compiler [91]. For the second static security metric, the number of indirect calls, we parse the output of `objdump` and count the static number of indirect branch instructions (i.e., `BX` and `BLX`). However, in the ARMv7-M architecture the same instruction can be used for a return instruction (i.e., backward edges) by storing the return address in the link register (`LR`). Thus, to measure the indirect calls (i.e., forward edges) we count branch instructions that use a register other than the link register.

3.6.2 Metric Collector Runtime Library

The goal of the metric collector library is to allow transparent and automatic measurement of dynamic metrics to the user. That is, it should not limit the resources available to the user (e.g., using a timer) or limit the functionality of the system. The metric collector uses the DWT unit cycle counter to measure the execution cycles for dynamic metric in Figure 3.3, such as the total privileged cycles or the sleep cycles. The DWT cycle counter provides precise measurement since it runs at the system clock speed. The metric collector library uses a global data structure that contains a dedicated variable for each of its collected metrics. Each variable is updated by reading the DWT cycle counter.

In order to provide transparent evaluation for the developer, the static library remaps some of the CMSIS library functionality to the BenchIoT library. The remapped CMSIS functions are instrumented to collect the metrics automatically at runtime. As an example, since the `WFI` instruction puts the processor to sleep till it is woken up by an interrupt, the remapped function intercepts the `WFI` call to track sleep cycles. A user can call the CMSIS functions normally, and the metric collector library will transparently collect the metrics. Another option for such instrumentation is to use a binary re-writer [104]. However, such a method relies on commercial non-open source software (i.e., IDA-Pro) and is thus not compatible with the goals of BenchIoT.

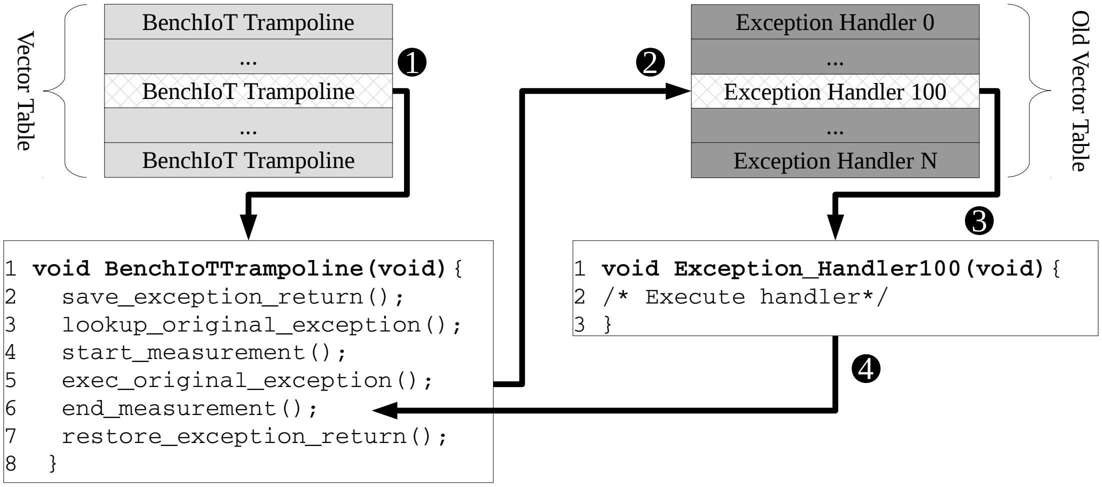
The second goal for the metric collector library is to automatically measure dynamic metrics such as the total privileged cycles. To achieve this, the metric collector automatically tracks: (1) privileged thread cycles; (2) all cycles of exception handlers. Measuring privileged thread cycles is done by instrumenting the `__set_CONTROL()` CMSIS call to track changes between privileged and unprivileged user modes. Measuring execution cycles of exception handlers poses several challenges.

First, while some exception handlers like SVC (i.e., system calls) can be measured by manual instrumentation to the SVC handler, other exception handlers like interrupt requests vary in the number of handler present and API used depending on the underlying hardware. Hence, they cannot be manually instrumented. Second, the hardware handles exception entry and exit, and there is no software path that can be instrumented. When an exception occurs, the hardware looks up the exception handler from the vector table, pushes the saved stack frame, and redirects execution to the exception handler. When the exception finishes, the hardware handles returning using the saved stack frame and special values stored in the link register `LR`.

To overcome these limitations, the metric collector library controls the vector table in order to track exception handlers. As shown in Figure 3.4, before the main application the metric collector library switches the vector table to point to the one controlled by itself. With this setup, when an exception occurs (e.g., `Exception_Handler100`), the hardware will redirect the execution to the `BenchIoTTrampoline` function ❶. To remember the special value for exception exit, `BenchIoTTrampoline` saves the value of `LR`. Next, it resolves the actual handler by reading the Interrupt Control and State Register (`ICSR`). It initiates the measurement and redirects the execution to the original handler ❷, which invokes `Exception_Handler100` ❸. After the original handler has finished, execution returns to `BenchIoTTrampoline` ❹, which ends the measurement and returns normally by using the saved `LR` value. The dynamic metrics measured by the metric collector are sent at the end of the benchmark through a UART to the user's machine. These are received automatically by the evaluation framework and stored to a file.

3.7 Evaluation

To demonstrate how the metrics described in section 3.3 enable evaluation of proposed security defenses, we evaluated the BenchIoT benchmarks with three defense mechanisms. We also compare the complexity of our benchmarks to that of existing embedded benchmarks.



```
void BenchIoTTrampoline(void){
  save_exception_return();
  lookup_original_exception();
  start_measurement();
  exec_original_exception();
  end_measurement();
  restore_exception_return();
 }
```

```
void Exception_Handler100(void){
/* Execute handler*/
}
```

Fig. 3.4. Exception handlers tracking with BenchIoT.

3.7.1 Defense Mechanisms

The first defense is ARM's Mbed-μVisor [105]. The μVisor is a hypervisor that enforces the principle of least privilege by running all application code in non-privileged mode. Only μVisor's code and parts of the OS run in privileged mode.

The second is a remote attestation mechanism drawn from well established attestation defenses [14, 106–108], and is purposed to authenticate the integrity of the code residing on the device. The remote attestation mechanism uses a real-time task that runs every 25ms in a separate thread to read the code in blocks, hash it, then send the hashed block to the server to verify the code integrity. At initialization, the remote attestation configures the MPU to save the code for reading and hashing the application in a special region in flash that is only accessible in privileged mode.

The final defense mechanism is a data integrity mechanism we draw from [12, 105, 109, 110] that provides data integrity through memory isolation. Our implementation moves sensitive data from RAM to a `SECURE_DATA_REGION` in Core Coupled RAM (CCRAM) at compile time. CCRAM is an optional memory bank that is isolated from RAM. It provides faster access to its data than RAM but has smaller size. The secure data region is accessible in privileged mode only. It is enabled before accessing the sensitive data and is disabled afterwards. The sensitive data depends on the underlying benchmark (e.g., Flash IAP in firmware updater). It is important to note that the goal of our security evaluation is to demonstrate how BenchIoT metrics can help evaluate existing defense mechanisms with respect to security benefits and performance overhead. It is *not* to propose new security mechanisms. The BenchIoT benchmarks are built with the standard configuration of IoT-MCUS and popular OSes to reflect real security challenges of current systems. For example, the baseline is evaluated using the default configuration of Mbed-OS, which means the MPU is not enabled and DEP is not supported.

We evaluated both the baseline and defense mechanisms on the STM32F479I-Eval [38] board. Measurements were averaged over five runs for each benchmark.

Note that since Mbed-μVisor and remote attestation require an OS (i.e., remote attestation requires a separate thread), it was only evaluated for the OS benchmarks.

3.7.2 Performance and Resource Usage Evaluation

Figure 3.5 shows the performance evaluation. For the OS benchmarks, the total runtime shows a modest runtime overhead for all mechanisms. The highest overhead occurs for remote attestation at 2.1% for firmware updater. Thus, from the viewpoint of runtime overhead, all the security mechanisms appear feasible for all the benchmarks running on IoT platforms. However, the story is more nuanced when we look at the effect of the security mechanisms on sleep cycles. The μVisor has no sleep cycles, which has an adverse effect on energy consumption. The μVisor disables sleep because of incompatibility issues [111] since implementation of sleep function differs depending to the underlying hardware (i.e., HAL library). Some HAL implementations break the privilege separation enforced by the μVisor, and as a result the μVisor goes into idle loop instead of entering sleep mode. The remote attestation mechanism decreases sleep cycles since it runs in a separate thread with a real-time task every 25ms, thus, the OS will run the remote attestation mechanism instead of entering sleep mode. On the other hand, the data integrity mechanism shows negligible change for sleep cycles.

Note that the reduction in runtime overhead for the connected display benchmark with μVisor occurs because the benchmark was configured to exclude the display. Porting the display functionality to the μVisor is a manual effort and is orthogonal to our goals of evaluating the security characteristics of the μVisor. Thus, to limit our efforts we utilize the option available by BenchIoT to run a benchmark without the Display Serial Interface (DSI) as mentioned in subsection 3.4.3.

For the bare-metal benchmarks, the data integrity mechanism shows similar overhead for the total runtime cycles as its OS counterpart. Moreover, in all bare-metal benchmarks, there is no sleep cycles because it is lacking the sleep manager provided by the Mbed-OS.

In order to collect the metrics in Figure 3.5 and all other dynamic results, the evaluation framework used the metric collector runtime library. As shown in Figure 3.5(c), the metric collector library has a low average overhead of 1.2%.

Figure 3.6 shows a comparison of the memory usage overhead. The μVisor and remote attestation mechanisms show an increase in memory usage overall. The remote attestation shows a large increase heap and stack usage (over 200%) since it requires an additional thread with a real-time task. However, it shows less than 30% increase in RAM since the larger portion of of RAM usage is due to the global `data` and `bss` regions. The μVisor requires additional global data and thus show a larger increase in RAM. Both require additional code and thus increase Flash usage. The data integrity mechanism for both the OS and bare-metal benchmarks change some local variables to globals and moves them to CCRAM. Thus, they show negligible effect on memory overall. Notice that data integrity mechanism is different between the bare-metal and the OS benchmarks. The bare-metal benchmarks are consistently smaller than their OS counterparts. As mentioned earlier in section 3.5, the bare-metal benchmarks differ in their implementation although they provide the same functionality. These differences are also manifested in the Flash metrics.

3.7.3 Security Evaluation

Minimizing privileged execution: Minimizing privileged execution is a desired security property (subsection 3.3.1). However, as shown in Figure 3.7, the remote attestation and data integrity mechanisms (for both OS and bare-metal) share the risk of over-privileged execution that are present in the insecure baseline, since they do not target minimizing privileged execution. Even with these defenses applied, almost the entire application runs in privileged mode (e.g., 98.9% for Smart-light using remote attestation in Figure 3.7(a)). The μVisor, however, shows the highest

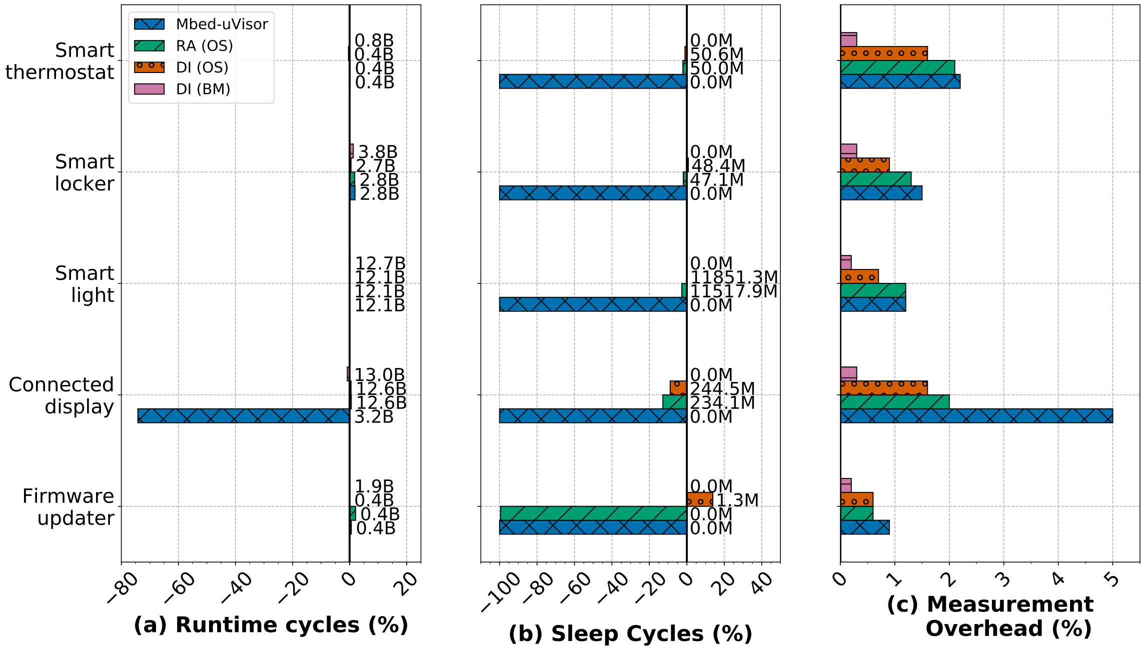


Fig. 3.5. Summary of BenchIoT performance metrics for μVisor, Remote Attestation, (RA) and Data Integrity (DI) defense mechanisms overhead as a % of the baseline insecure applications. BM: Bare-Metal.

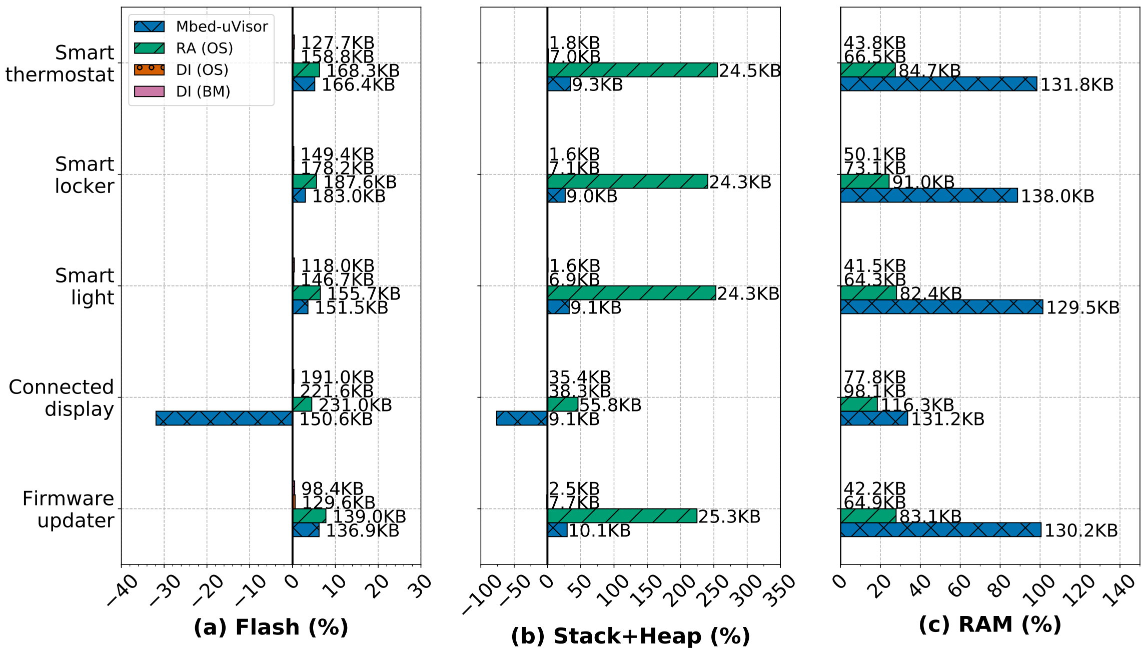


Fig. 3.6. Summary of BenchIoT memory utilization metrics for μVisor, Remote Attestation (RA), and Data Integrity (DI) defense mechanisms overhead as a % over the baseline applications. The size in KB is shown above each bar.

Table 3.3.
Summary of BenchIoT memory isolation and control-flow hijacking metrics for Mbed-μVisor, Remote Attestation (RA) and Data Integrity (DI) defense mechanisms overhead as a percentage of the baseline insecure applications. BM: Bare-metal

Defense Type	**Benchmark**	**Metric**				
		Max. Code Reg. ratio	Max. Data Reg. ratio	# ROP gadgets	# Indirect calls	DEP
μVisor (OS)	Smart-light	0.0%	0.0%	10,990 (29.9%)	207 (14.4%)	✗
	Smart-thermostat	0.0%	0.0%	12,087 (31.1%)	199 (11.8%)	✗
	Smart-locker	0.0%	0.0%	12,318 (28.6%)	211 (13.4%)	✗
	Firmware Updater	0.0%	0.0%	10,364 (32.8%)	190 (11.8%)	✗
	Connected display	0.0%	0.0%	11,478 (-4.3%)	198 (-63.5%)	✗
RA (OS)	Smart-light	-0.2%	0.0%	8,833 (4.4%)	195 (7.7%)	✓
	Smart-thermostat	-0.2%	0.0%	9,765 (5.9%)	197 (10.7%)	✓
	Smart-locker	-0.2%	0.0%	10,408 (8.6%)	205 (10.2%)	✓
	Firmware Updater	-0.2%	0.0%	8,556 (9.7%)	189 (11.2%%)	✗
	Connected display	-0.1%	0.0%	12,603 (5.1%)	561 (3.5%)	✓
DI (OS)	Smart-light	0.0%	-0.1%	8,398 (-0.8%)	181 (0.0%)	✗
	Smart-thermostat	0.0%	-0.1%	9,231 (0.1%)	178 (0.0%)	✗
	Smart-locker	0.0%	-0.8%	9,567 (-0.1%)	186 (0.0%)	✗
	Firmware Updater	0.0%	-0.01%	7,878 (1.0%)	170 (0%)	✗
	Connected display	0.0%	-1.8%	12,082 (0.8%)	542 (0%)	✗
DI (Bare-metal)	Smart-light	0.0%	-0.1%	6,040 (0.4%)	103 (0.0%)	✗
	Smart-thermostat	0.0%	-0.2%	6,642 (1.0%)	98 (0.0%)	✗
	Smart-locker	0.0%	-1.1%	7,015 (0.3)	108 (0.0%)	✗
	Firmware Updater	0.0%	-0.01%	5,332 (0.4%)	90 (0.0%)	✗
	Connected display	0.0%	-2.6%	9,697 (2.2%)	462 (0.0%)	✗

reduction in privileged execution. For example, only 1.4% of Smart-light runs in privileged mode. The Firmware-updater shows the lowest reduction for μVisor (i.e., 55.4%) since it requires privileges to execute correctly (i.e., writing to Flash and running the new firmware). However, the μVisor still reduces the total privilege cycles by 44%. These improvements are expected since the μVisor runs all-application code in non-privileged mode, except for the μVisor and OS code. The increase in SVC cycles in all defenses is because they utilize system calls to execute their code. For example, the highest increase in SVC cycles is remote attestation that uses an SVC every 25ms to hash the firmware. Since the Smart-light application is the longest-running benchmark, it will intuitively have the largest increase in SVC cycles (i.e., 2,173.7%). The percentage of the increase is not shown in bare-metal benchmarks since the baseline does not use SVC calls.

Enforcing memory isolation: The insecure baseline application allows access to all code and data, thus its maximum code region ratio and maximum data region ratio are both `1.0`. Enforcing memory isolation reduces both ratios. The remote attestation mechanism isolates its own code in a separate region using the MPU. Thus, the maximum code region is the rest of the application code other than the remote attestation code—the improvement in the maximum code region ratio is 0.2% in Table 3.3. Similarly, the data integrity mechanism improves the maximum data region ratio. However, for both mechanisms 99% of the code and data is still always accessible to the normal application. The μVisor enables manual data isolation between threads using special API calls. However, we did not use this feature since we aim to evaluate the general characteristics of defenses and not develop our own.

Control-flow hijacking protection: As shown in Table 3.3, none of the mechanisms reduce the attack surface against code reuse attacks (i.e., ROP gadgets and indirect calls). The μVisor and remote attestation mechanisms increase the code size of the application, intuitively they will increase the number of ROP gadgets and indirect calls. The largest increase in the number of ROP gadgets occurs with the μVisor at an average of 23.6% for the five benchmarks since it requires larger code to be added. The data integrity mechanism on the other hand only instruments the benchmark with small code to enable and disable the secure data region, and thus causes limited increase in the number of ROP gadgets and indirect calls. The reduction in the number of ROP gadgets and indirect calls for the connected display application of the μVisor is because the display driver was disabled, and thus its code was not linked to the application. An option to improve these defenses against code reuse attacks is to couple them with established mechanisms such as CFI. Moreover, an important

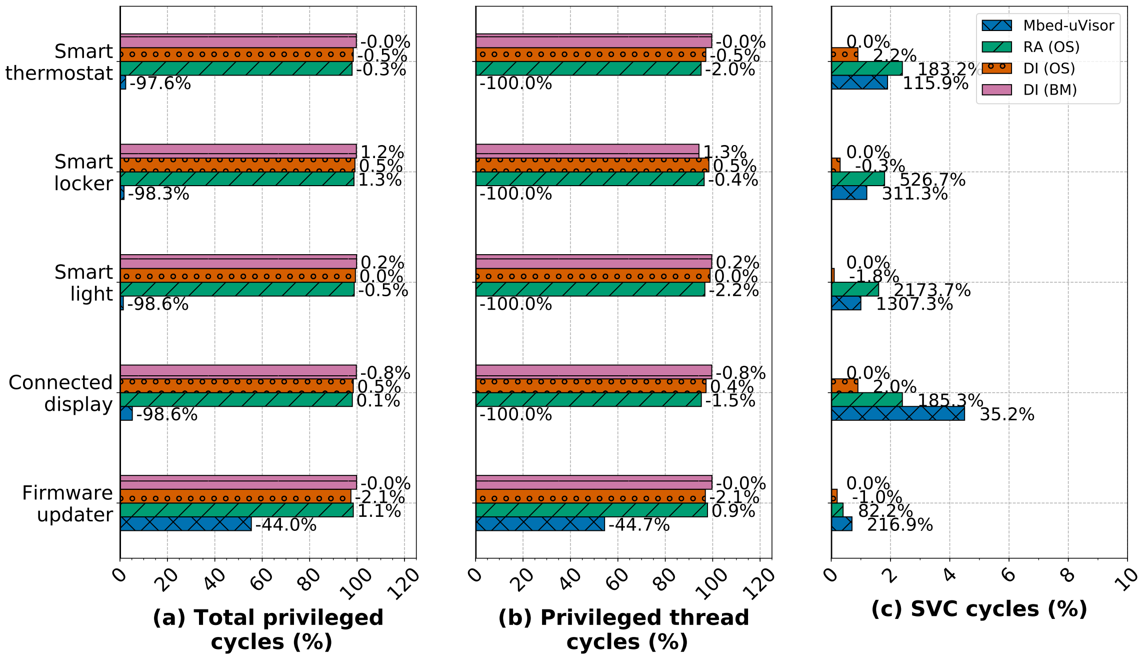


Fig. 3.7. Summary of BenchIoT comparison of minimizing privileged execution cycles for Mbed-μVisor, Remote Attestation (RA) and Data Integrity (DI) defense mechanisms as a % w.r.t the total runtime execution cycles. The overhead as a % of the baseline insecure applications is shown above each bar. BM: Bare-Metal

aspect in defending against control-flow hijacking is enabling DEP. However, only the remote attestation configures the MPU to enforce DEP. The μVisor does not enforce DEP on heap. A similar observation was made by Clements *et al.* [9]. The data integrity mechanism enables all access permissions to the background region (i.e., all the memory space) then configures the MPU for various regions it is interested in. However, regions that are not configured remain with the writable and executable permissions, thus breaking DEP.

3.7.4 Energy Evaluation

Now we look at the energy implication of the benchmarks (Figure 3.8). While all mechanisms showed similar runtime overhead, the energy consumption for the

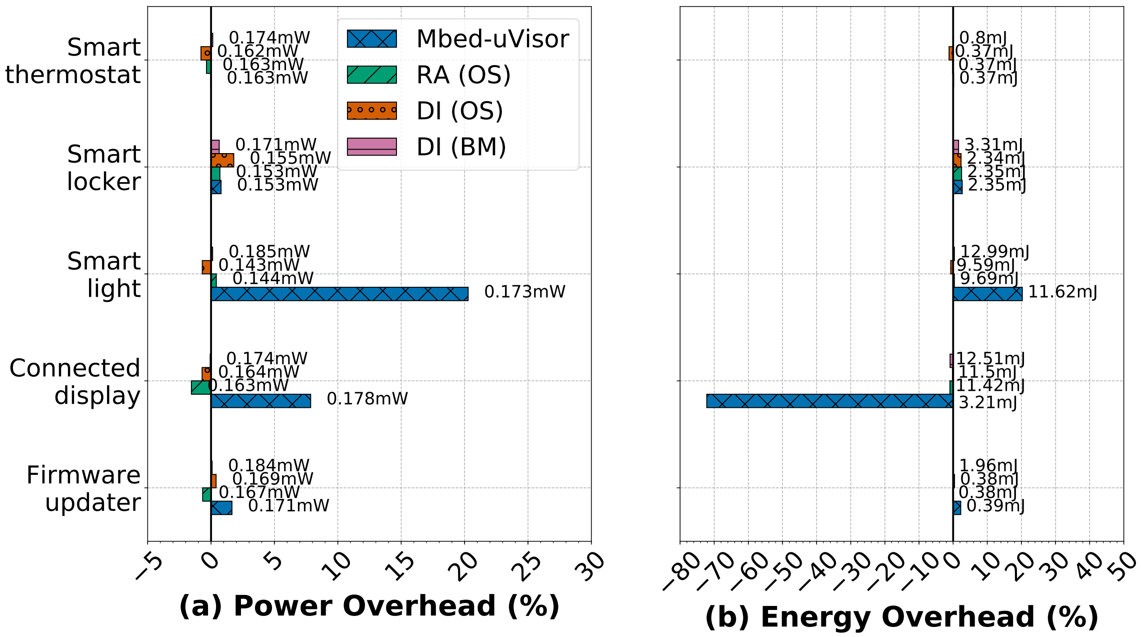
μVisor mechanism increases significantly for the Smart-light benchmark. The Smart-light benchmark spends large amounts of time in its sleep cycle, and since the μVisor disables sleep cycles, the increase is pronounced in this application. Since the μVisor disables sleep cycles, it consistently has an adverse effect on energy consumption for all applications and this correlates to a drop in sleep cycles as shown in Figure 3.5. Even if security mechanisms provide similar runtime overhead (e.g., data integrity and μVisor for Smart-light), the difference in energy consumption can vary widely, with an increase of 20% for μVisor. Such a conclusion cannot be obtained simply from the metric of the total runtime overhead, but only when used in conjunction with our metric of sleep cycles or energy consumed.

For the bare-metal benchmarks, the lack of an OS results in a lack of the sleep manager, and thus the device keeps polling and drawing the same average power all throughout. This can be noticed by the lack of sleep cycles in Figure 3.5 for the bare-metal benchmarks. Thus, difference in energy consumption is caused by the increase in total runtime due to the defense mechanism.

3.7.5 Code Complexity Comparison to BEEBS

To measure the complexity of BenchIoT benchmarks, we measure the cyclomatic complexity [112] and compare to the BEEBS [23] benchmarks. BEEBS has been used for security evaluation in embedded systems by EPOXY [8] and for energy evaluation by many prior works [113–115]. We exclude HAL libraries from the measurements for both benchmark suites as they differ based on the vendor and the hardware as discussed in subsection 3.4.4. This provides a consistent comparison without the influence of the underlying hardware.

Table 3.4 shows the comparison by computing the minimum, maximum, and median cyclomatic complexity and lines of code across all benchmarks. BenchIoT shows much larger numbers—median complexity is higher by 162X (bare-metal) and 343X (Mbed OS). The results are expected since BEEBS is designed to evaluate the energy



(a) Power Overhead (%)

(b) Energy Overhead (%)

Fig. 3.8. Summary of power and energy consumption with the BenchIoT benchmarks for the defense mechanisms as a % overhead of the baseline insecure applications. Power and energy values are shown above each bar in mW and mJ, respectively. BM: Bare-metal

Table 3.4.
Comparison of code complexity between BenchIoT and BEEBS.

Benchmark Suite	**Cyclomatic Complexity**			**Lines of Code**		
	Minimum	Maximum	Median	Minimum	Maximum	Median
BEEBS	3	1,235	16	22	6,198	97
BenchIoT (Bare-metal)	2,566	3,997	2,607	17,562	23,066	17,778
BenchIoT (OS)	5,456	6,887	5,497	31,828	37,331	32,038

efficiency of the architecture, and not meant to provide stand-ins to real IoT applications. For example, BEEBS excludes peripherals and does not incorporate network functionality.

3.8 Related Work

Numerous benchmarking suites have been proposed by the systems research community. However we focus our discussion on benchmarks targeting MCUS and IoT-MCUS. Table 3.5 shows a comparison between BenchIoT and other benchmarks.

Desktop benchmarks: Soteria [118] is a static analysis system targeting IoT platforms (e.g., Samsung's SmartThings market), which are assumed to be connected to the cloud. IoTAbench [119] and RIoTBench [120] are benchmarks for large-scale *data analysis* of IoT applications. BenchIoT however targets IoT-MCUS.

High-end embedded systems benchmarks: Mibench [21] is a set of 35 general purpose applications targeting embedded systems that are deigned to evaluate the performance of the system. The benchmarks are user-space applications, with some of the benchmarks assuming the presence of an OS and file system. ParMiBench [121] is an extension of Mibench targeting multi-core embedded processors. Other benchmarks target specific applications of embedded systems. MediaBench [122] targets multimedia applications. DSP-stone [123] evaluates compilers for Digital Signal Processing (DSP) applications for embedded systems. BenchIoT benchmarks differ from

Table 3.5.
A Comparison of benchmarks and their categorization with respect to task type, networking communication, and peripherals between BenchIoT and other benchmarking suites.

Benchmark	**Task Type**			**Network**	**Peripherals**
	Sense	Process	Actuate	**Connectivity**	
BEEBS [23]		✓			
Dhrystone [22]		✓			
CoreMark [85]		✓			
IoTMark [116]	✓	✓		partially (bluetooth only)	only I^2C
SecureMark [117]		✓			
BenchIoT	✓	✓	✓	✓	✓

the above since we focus on IoT benchmarks, enabling security evaluations of IoT-MCUS, and incorporating networking.

MCUS Benchmarks: The Worst-Case Execution Time (WCET) [124] evaluates the worst execution time for real-time systems. Dhrystone [22] is a synthetic benchmark to evaluate integer performance. BEEBS [23] is a collection of benchmarks from Mibench, WCET, DSP-stone, and the Livermore Fortran kernels [125] to evaluate energy consumption for bare-metal systems. CoreMark [85] targets evaluating processor performance. However, all target a specific metric, do not utilize peripherals, and do not show most of the characteristics of IoT applications. In contrast, BenchIoT is aimed to enable security evaluation, it incorporates IoT application characteristics, and covers both bare-metal and OS benchmarks.

IoT-MCUS benchmarks: IoTMark [116] evaluates the energy overhead of wireless protocols such as Bluetooth. SecureMark [117] measures performance and energy for implementing TLS on IoT edge nodes, it does not however demonstrate connectivity. BenchIoT on the other hand demonstrates TCP/IP connectivity as well as security, performance, memory, and energy evaluation.

3.9 Discussion

Extending BenchIoT: The flexible design of BenchIoT enables users to extend it with their customized metrics or benchmarks. For example, a user interested in cycles spent executing function `foo` can extend the global data structure of the metric collector library, instrument `foo` with the BenchIoT API at the beginning and at the end of `foo`, and add the metric to the result collection interface. Only 10 LoC are needed for this customized metric. Moreover, users can evaluate their customized benchmarks using the BenchIoT evaluation framework. The users customized benchmark can use external peripherals (e.g., BLE) that were not included in core BenchIoT benchmarks. We note that the reason for excluding external peripherals from the five benchmarks is portability. For example, to add BLE users will need to buy an additional hardware module for BLE and use its non-portable software libraries. Thus, external peripherals were excluded from the core benchmark suite. Since users can easily add their own applications and evaluate them, we leave the choice of adding external peripherals to the users. More details are available at [88].

Portability of BenchIoT: We believe BenchIoT can be extended to ARMv8-M, as it shares many of the characteristics of ARMv7-M (i.e., include the TrustZone execution). ARMv8-M however is a fairly new architecture, and a limited number of boards are available at the time of writing. We leave this as future work. For other architectures, the concepts of BenchIoT are applicable. However, since BenchIoT follows a software-based approach, the main task is porting the metric collector runtime library, since it handles exception entry and exit. These are architecture dependent (e.g., calling conventions, registers) and require architecture dependent implementation.

3.10 Conclusion

Benchmarks are pivotal for continued and accelerated innovation in the IoT domain. Benchmarks provide a common ground to evaluate and compare the different security solutions. Alternatively, the lack of benchmarks burdens researchers with measurements and leads to ad-hoc evaluations. For IoT-MCUS, the problem is exacerbated by the absence of commonly measured evaluation metrics, the tedious measurement process, and the multi-dimensional metrics (performance, energy, security).

Concerned by the rising rate of attacks against IoT devices and the ad-hoc evaluation of its defenses, we developed BenchIoT, a benchmarking suite and an evaluation framework for IoT-MCUS to enable evaluating and comparing security solutions. The suite is comprised of five representative benchmarks, that represent salient IoT application characteristics: network connectivity, sense, compute, and actuate. The applications run on bare-metal or a real-time embedded OS and are evaluated through four types of metrics—security, performance,memory usage, and energy. We illustrate how the evaluation metrics provide non-trivial insights, such as the differing effects of different defenses on consumed energy, even though both show a similar runtime overhead. BenchIoT benchmarks are open sourced freely available to the research community [88].

4. μRAI: SECURING EMBEDDED SYSTEMS WITH RETURN ADDRESS INTEGRITY

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, such as 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 re-purposes 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.

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

4.1 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 [5, 6], and resulted in power grid blackouts [28] among others.

A significant, yet particularly vulnerable portion of embedded devices are micro-controller -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 [29, 30]). Existing solutions to protect MCUS [8, 9, 12–18, 46, 47, 126], 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 [8, 9, 12], such as Data Execution Prevention (DEP), stack canaries [7], 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 [11]. 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) [65] can be applied to protect forward edges as was done in desktop systems [64, 127]. 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) [55] 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 [7, 47], and randomization [8].

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 mechanisms. 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 reflashing 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 [8,46,47], memory isolation [9,12,46], or CFI [13,14]. However, these defenses only reduce the attack surface, but remain bypassable by ROP style attacks [57, 68, 128–130]. 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 [68]. An alternative approach is to rely on information hiding. However, information hiding based defenses [8, 46, 47] are vulnerable to information disclosure [57, 58] and profiling [80] 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 [8, 61] only results in a few bits of entropy. A safe stack protected through Software Fault Isolation (SFI) [59,60] removes the need for information hiding, but will incur high overhead [63].

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) [47]. In order to enforce stronger guarantees to protect return addresses one option is to use a shadow stack [62, 63]. However, a shadow stack requires hardware isolation to protect it from information disclosure [63, 131]. 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 [69]. The other option is to rely on a TEE such as ARM's TrustZone [13, 24]. Both will result in high overhead (e.g., 10–500% [13]). More importantly, TEEs are not commonly available on MCUS [47]. 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 [132–136], requiring protections via software updates [43, 44]. 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 [65], 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 [69]) 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% [63, 69, 81]). 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.

Contribution: This chapter 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.

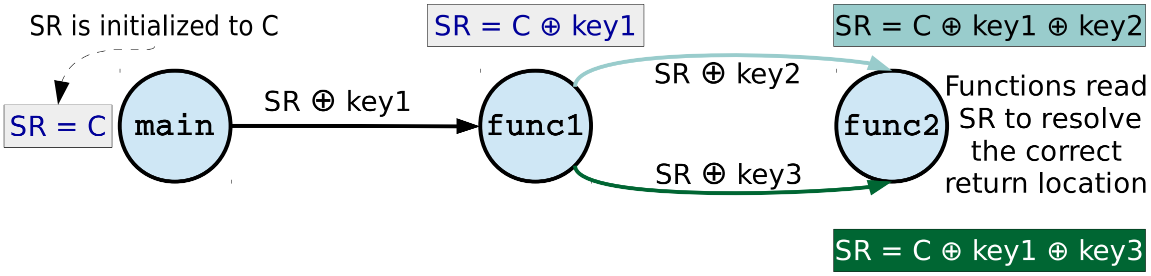


Fig. 4.1. Illustration of encoding SR through an `XOR` chain. Arrows indicate a call site in the call graph. SR is XORed each time an edge is walked.

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 4.1, 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 and interrupts). Exception handlers execute in privileged mode, and can execute asynchronously (i.e., interrupts). As shown

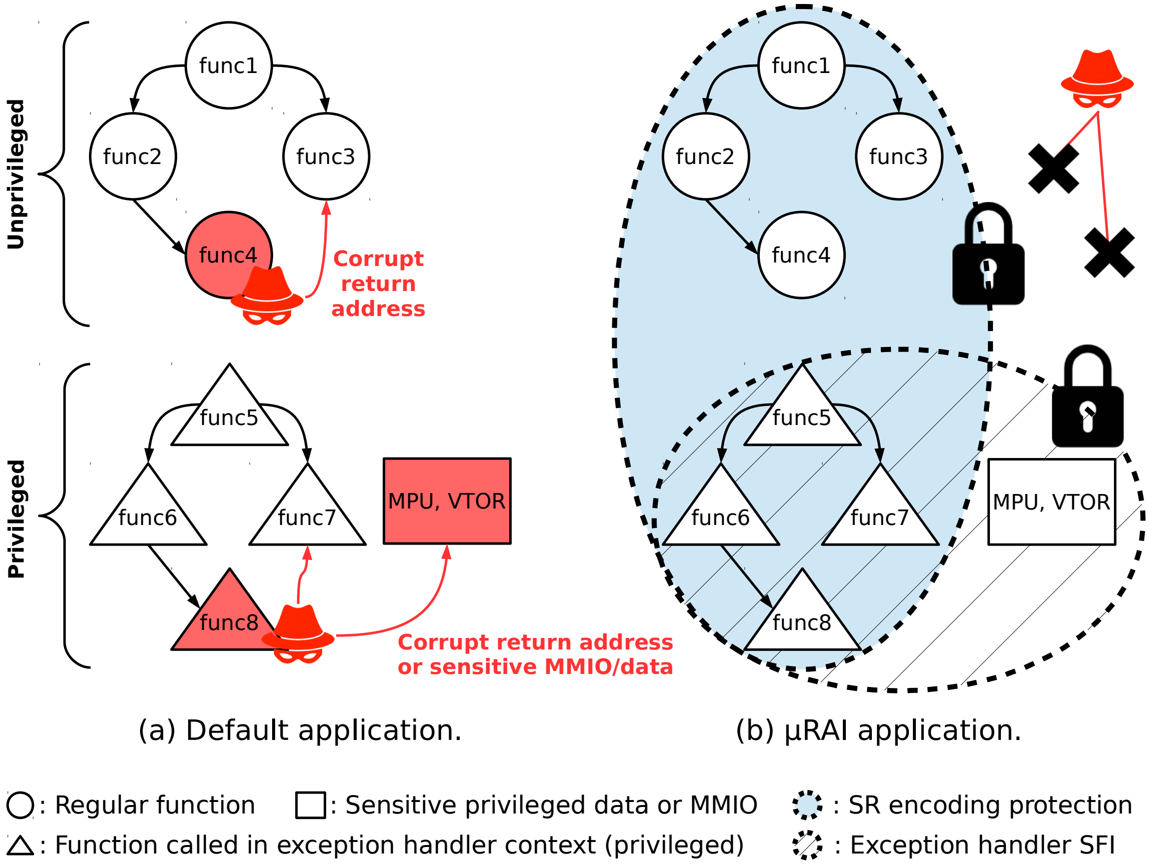


Fig. 4.2. Illustration μRAI's protections. μRAI prevents exploiting a vulnerable function (e.g., `func8`) to corrupt the return address or disable the MPU in privileged execution by coupling its SR encoding with exception handler SFI.

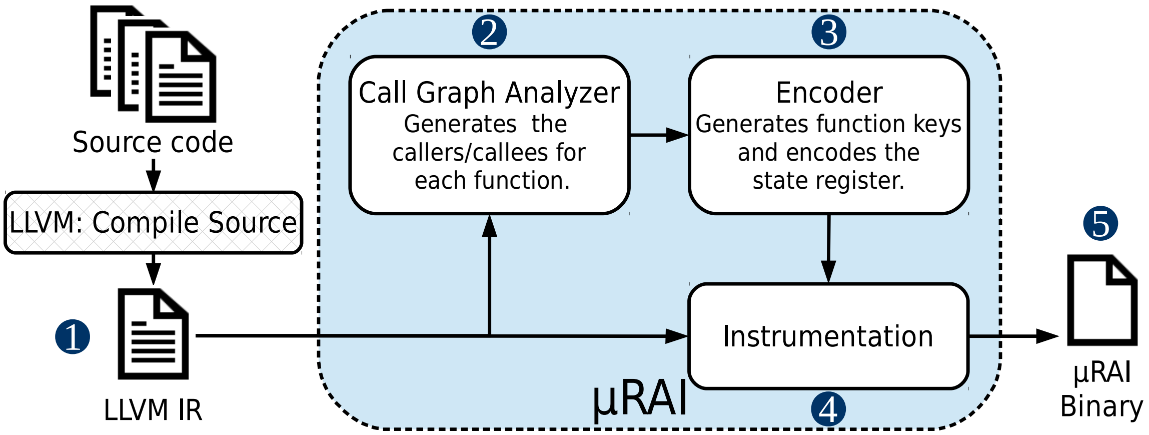

Source code
LLVM: Compile Source
1
LLVM IR
2
Call Graph Analyzer
Generates the callers/callees for each function.
3
Encoder
Generates function keys and encodes the state register.
Instrumentation
4
µRAI
5
µRAI Binary

Fig. 4.3. An overview of μRAI's workflow.

in Figure 4.2(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 4.2(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 4.3, 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 [82,137,138] 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 [85]. μRAI shows an average overhead of 0.1%. In summary, our contributions are:

- **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.
- **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.
- **μRAI:** We design and implement a prototype that enforces the RAI property on MCUS. We evaluate our implementation on CoreMark [85], representative MCUS applications, and proof-of-concept attacks. Our results show that μRAI enforces the RAI property with a runtime overhead of 0.1%.

4.2 Threat Model

We assume an attacker with arbitrary read and write primitives aiming to hijack the control-flow (e.g., via ROP [55]) 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 [82, 137, 138] 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 [51].

4.3 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 [40]. ARMv7-M is applied to Cortex-M (3,4,7) processors, the most widely-deployed processor family for 32-bit MCUS [43, 44].

Memory layout: As shown in Figure 4.4, 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 [38] has only 2MB Flash for its code region, and only 320KB for RAM.

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 [40]).

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.

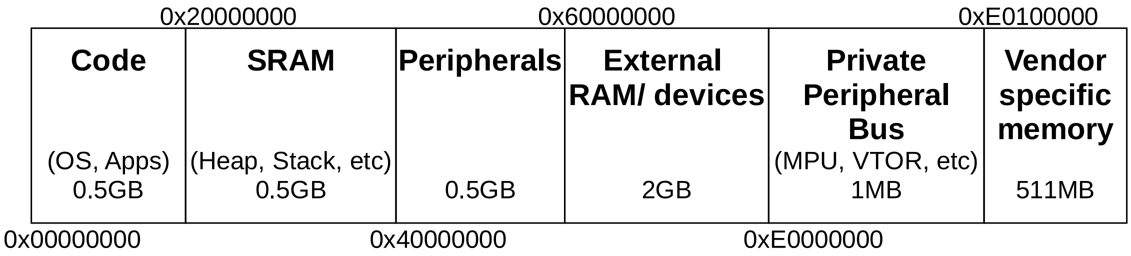


Fig. 4.4. ARMv7-M memory layout.

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.

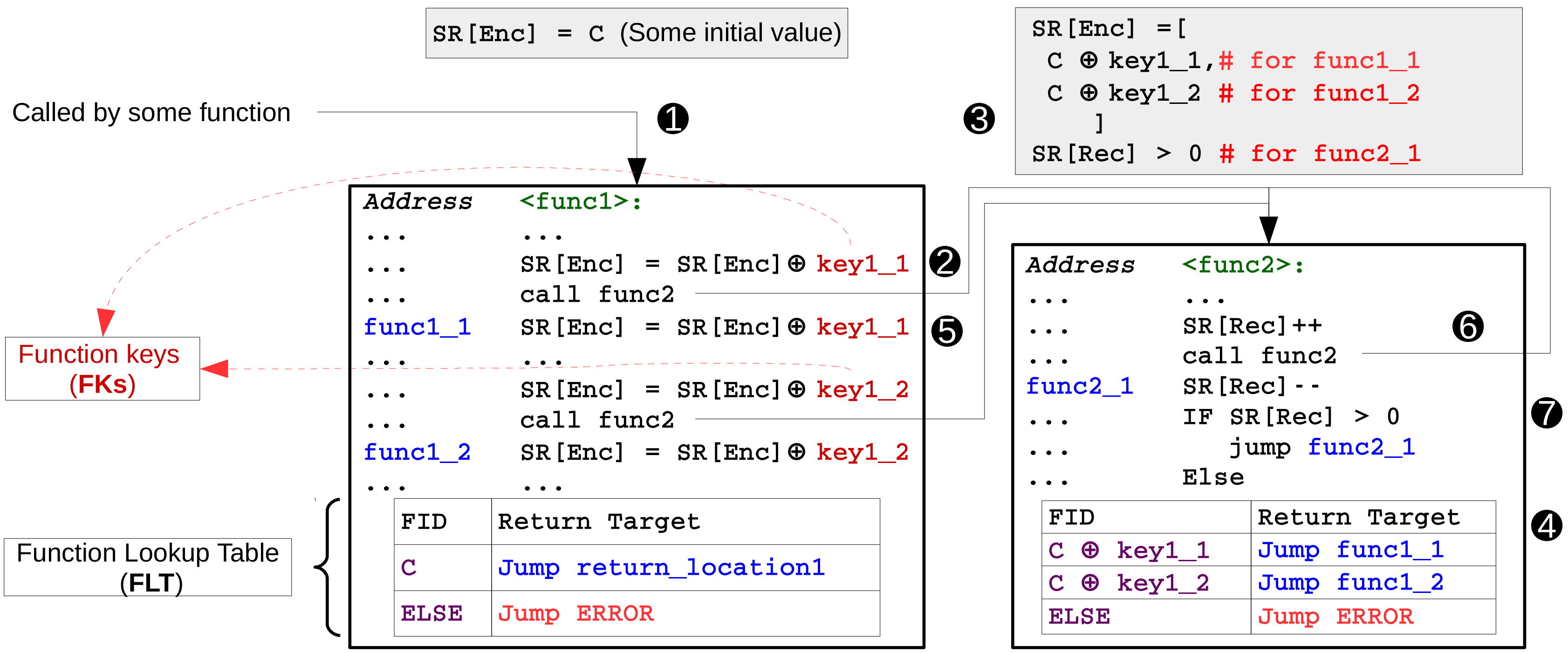
Call instruction types: A call instruction in ARMv7-M has four possible types, shown in Table 4.1. Both direct and indirect calls can automatically update `LR` to hold the return address (i.e., the instruction following the call site) by using any of the branch and link instructions in Table 4.1. Subsequent functions push `LR` on the stack to store the return address in case they use another branch and link instruction to call another function.

4.4 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

Table 4.1.
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>`



```
SR[Enc] = C (Some initial value)
```

```
Address    <func1>:
...        ...
...        SR[Enc] = SR[Enc]⊕ key1_1
...        call func2
func1_1    SR[Enc] = SR[Enc]⊕ key1_1
...        ...
...        SR[Enc] = SR[Enc]⊕ key1_2
...        call func2
func1_2    SR[Enc] = SR[Enc]⊕ key1_2
...        ...
```

FID	Return Target
C	Jump return_location1
ELSE	Jump ERROR

```
SR[Enc] =[
 C ⊕ key1_1,# for func1_1
 C ⊕ key1_2 # for func1_2
    ]
SR[Rec] > 0 # for func2_1
```

```
Address    <func2>:
...        ...
...        SR[Rec]++
...        call func2
func2_1    SR[Rec]--
...        IF SR[Rec] > 0
...           jump func2_1
...        Else
```

FID	Return Target
C ⊕ key1_1	Jump func1_1
C ⊕ key1_2	Jump func1_2
ELSE	Jump ERROR

Fig. 4.5. Illustration of μRAI's design. Enc: Encoded SR. Rec: Recursive SR.

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 [139,140]), 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 [141] 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 [65], 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 [65], while dynamic methods [67] 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 [68,142]. 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 4.5. 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.

4.4.1 μRAI Terminology

Before discussing the details of μRAI's design, we first define its main components, which are illustrated in Figure 4.5.

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.

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.

4.4.2 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 4.5, 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 ⊕ 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 ⊕ key1_1 → 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 ⊕ 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 4.5 is a recursive function. Assume `func2` is called from the first call site in `func1` (i.e., `SR = C ⊕ key1_1`). Then, `func2` calls itself twice at ❻ (i.e., `SR = C ⊕ key1_1 ⊕ any_key ⊕ any_key`), the value of the SR will be `C ⊕ 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 [8, 69].*

μRAI allows bits reserved for the recursion counter to be adaptable according to the underlying application.

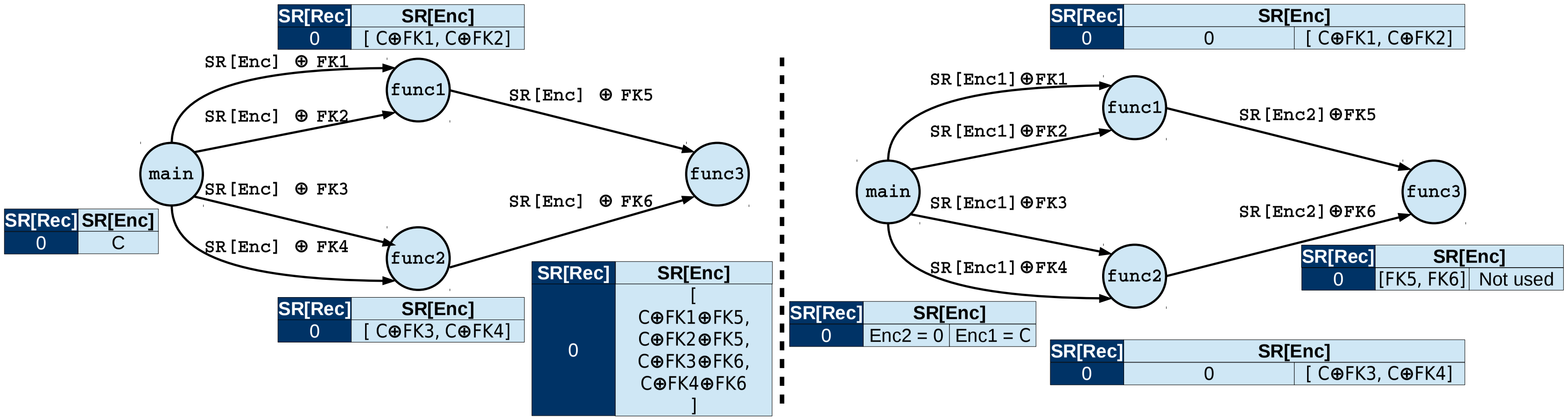
Using the SR as a single value however is prone to path-explosion and thus large increases in the FLT. It also cannot handle corner cases for recursive calls (e.g., indirect recursion). μRAI resolves these issues by partitioning the SR.

4.4.3 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 4.3) is to construct the firmware's call graph [141]. μRAI uses the call graph to obtain the possible return targets for each function in the FLT. As mentioned previously in section 4.4, 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 4.6(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 4.6(a), since it can be reached by multiple paths (i.e., two paths from `main`→`func1` or two paths from `main`→`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 4.6(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 `func1` or `func2` call `func3`,



(a) SR without segmentation

(b) Segmented SR

Fig. 4.6. Illustration of using SR segmentation to reduce path explosion. Segmentation reduced the possible SR values for `func3` by half.

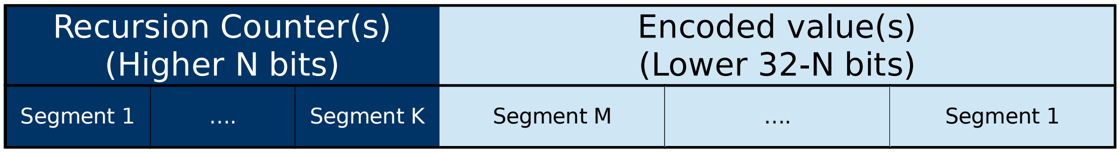


Fig. 4.7. Illustration segmenting the state register.

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 subsection 4.4.2, recursion is rare in MCUS [8,69]. Since recursion is discouraged on MCUS, we provide the details for handling special cases of recursion in Appendix A.

Figure 4.7 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.

4.4.4 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 4.6, 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 4.8, 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 4.5, `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.

4.4.5 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 4.8; 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 [59, 60] 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 4.8, 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 for only functions called within an exception handler context, enables

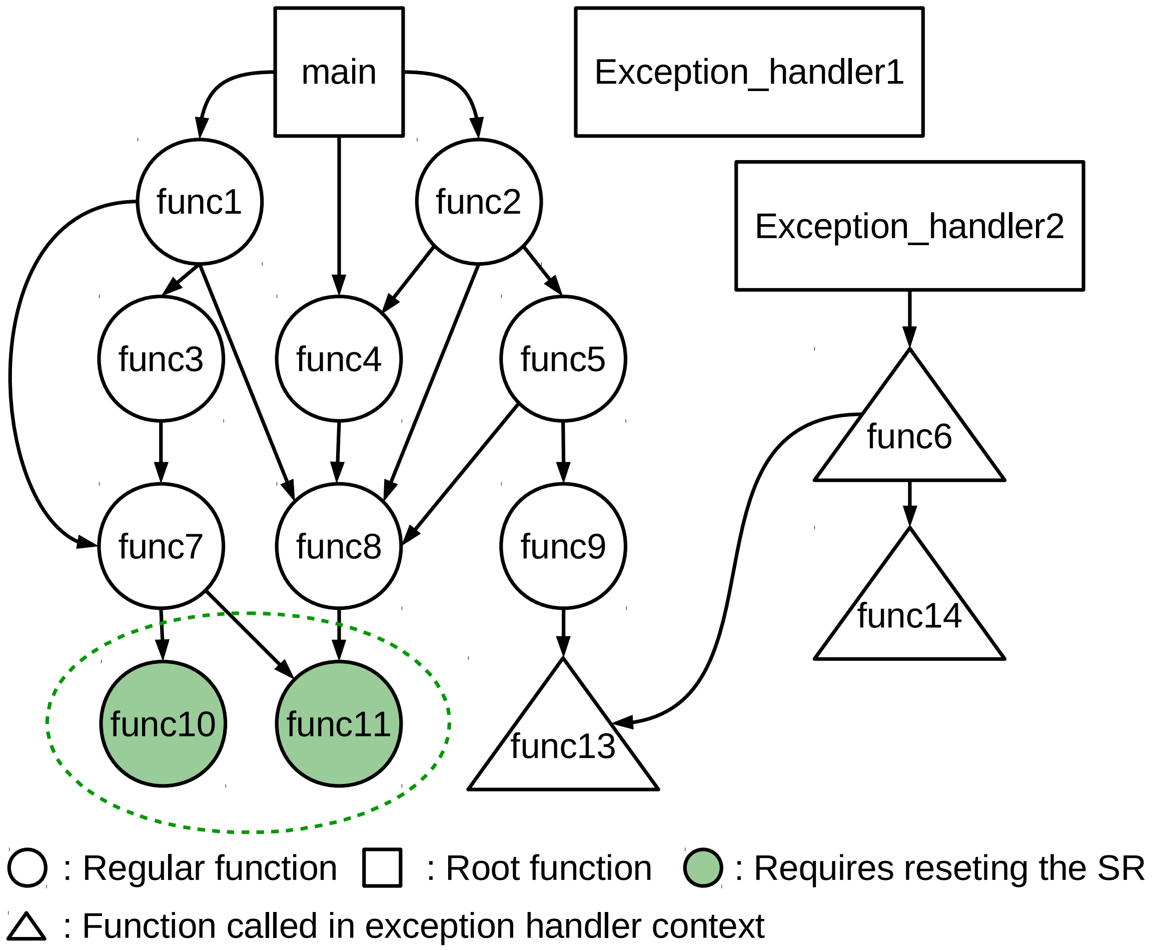


Fig. 4.8. Conceptual illustration of μRAI's call graph analysis.

an efficient implementation and limit the effect on the runtime overhead since many of the instrumented functions execute for a limited portion of application.

4.4.6 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 subsection 4.4.3, 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.

4.4.7 Target Lookup Routine Policy

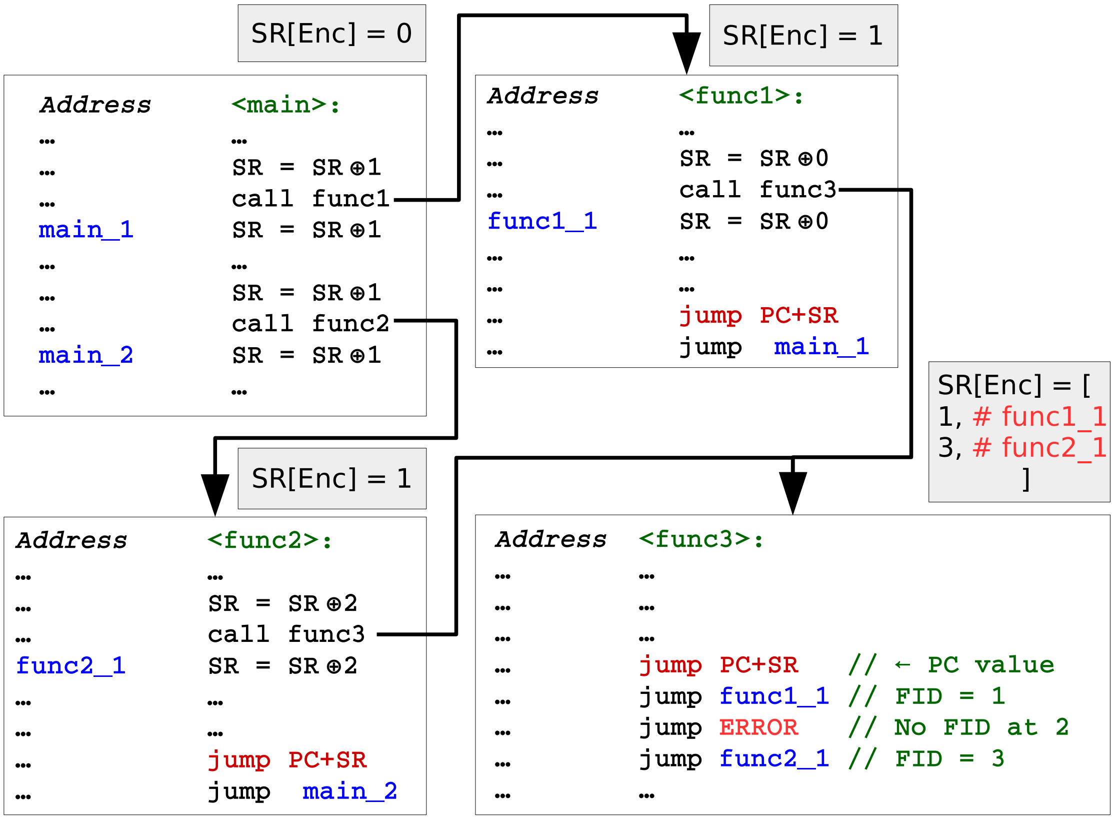
Enforcing the RAI property is important, however, it is equally important to maintain an acceptable performance overhead [26]. One simple TLR policy to resolve the correct return location is to use a switch statement and compare the value of the SR sequentially to the FID values in the FLT, and return 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. 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 4.9 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.

4.5 Implementation

μRAI is comprised of four components (see Figure 4.3). 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 [143]. The encoder is implemented as a Python script to leverage the graph libraries [144]. 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.



```
Address        <main>:
…              …
…              SR = SR⊕1
…              call func1
main_1         SR = SR⊕1
…              …
…              SR = SR⊕1
…              call func2
main_2         SR = SR⊕1
…              …
```

```
Address        <func1>:
…              …
…              SR = SR⊕0
…              call func3
func1_1        SR = SR⊕0
…              …
…              …
…              jump PC+SR
…              jump  main_1
```

```
Address        <func2>:
…              …
…              SR = SR⊕2
…              call func3
func2_1        SR = SR⊕2
…              …
…              …
…              jump PC+SR
…              jump  main_2
```

```
Address   <func3>:
…         …
…         …
…         …
…         jump PC+SR    // ← PC value
…         jump func1_1  // FID = 1
…         jump ERROR    // No FID at 2
…         jump func2_1  // FID = 3
…         …
```

Fig. 4.9. Illustration μRAI's relative-jump TLR policy.

4.5.1 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 [145] 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 subsection 4.8.1 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.

4.5.2 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 [40]. 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 subsection 4.4.4. 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 (subsection 4.4.4).

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 subsection 4.4.4, 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 4.3). 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 (subsection 4.4.7), thus increasing the memory overhead.

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

Fig. 4.10. μRAI instrumentation for call instructions.

```
; check the counter bits do not overflow before incrementing
add lr ,#0x01000000 ;increment counter
b func              ;direct call to func
sub lr ,#0x01000000 ;decrement counter
```

Fig. 4.11. An example of μRAI's instrumentation for recursive call sites. The recursion counter shown uses the higher eight bits of `LR`.

4.5.3 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.

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 4.1). A detailed disassembly is shown in Figure 4.10 and Figure 4.11.

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

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

Fig. 4.12. TLR instrumentation without SR segmentation.

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 (see Figure 4.12 and Figure 4.13 for a detailed disassembly of both cases). 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.

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

Fig. 4.13. TLR with SR segmentation. N and M are constants calculated depending on the function and the start of its segment.

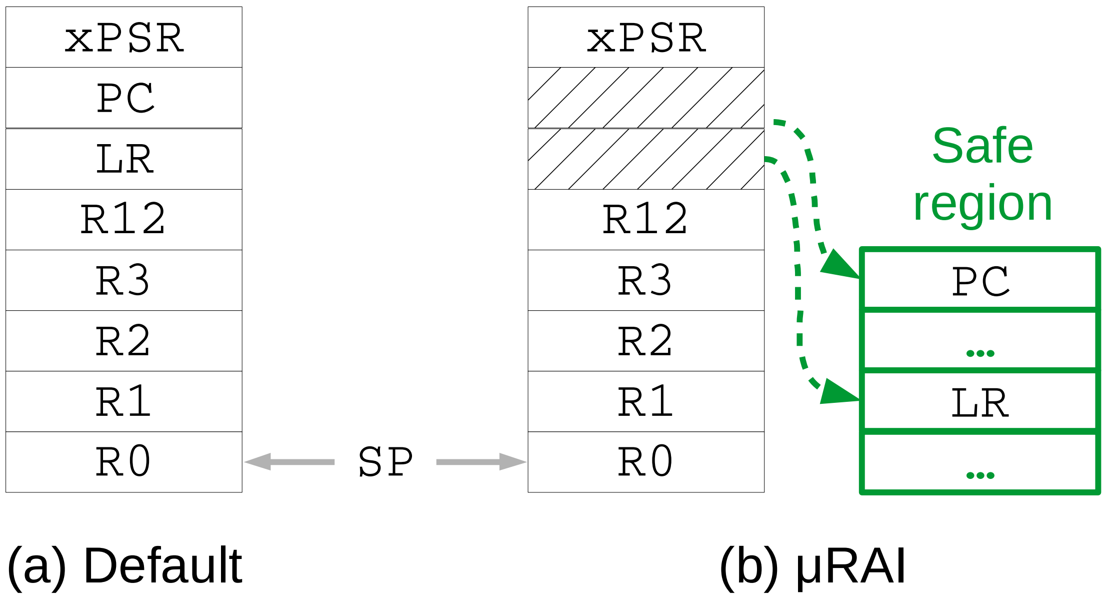


Fig. 4.14. Illustration of exception stack frame for ARMv7-M.

4.5.4 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 subsection 4.5.2. 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 subsection 4.5.5.

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

Fig. 4.15. μRAI's exception handler SFI protection. The MPU Region Number Register (`MPU_RNR`) is middle address of the MPU.

4.5.5 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 4.14(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 4.14(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 4.3). 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 [38]). 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 [59]. For our evaluation, we leverage this more efficient bit-masking approach. See Figure 4.15 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.

4.6 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 [85]. *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 [85] to measure MCUS performance. The evaluation is performed using the STM32f479-EVAL [38] board and includes the cost of type-CFI.

4.6.1 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 [146–148], 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 [149], 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 [150]), where the attacker directly over-writes the return address, without needing to corrupt other memory locations. *Third*, a stack pivot [151], 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 4.2, 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 Figure 4.12 or Figure 4.13, 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 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 Figure 4.12 or Figure 4.13, 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.

4.6.2 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 [82, 138, 142]. We build our prototype on top of ACES' [9] type-based CFI as

Table 4.2.
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

it provides a more precise target set than other existing work for MCUS. Intuitively, the chance of a control-flow bending style attack [68] 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 4.2 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 4.2, 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 4.2. 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 [65].

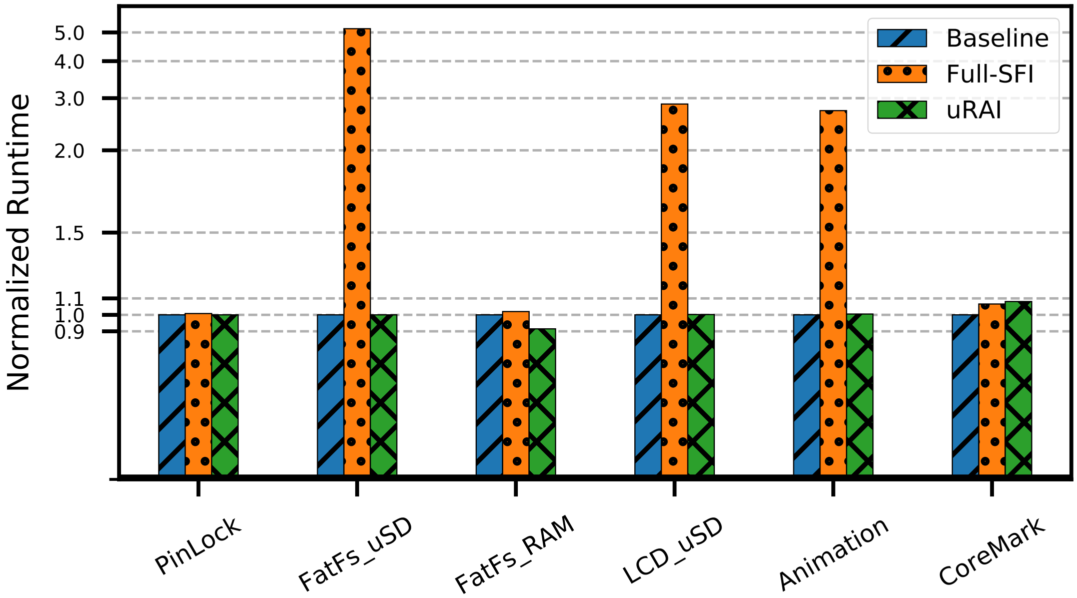


Fig. 4.16. Comparison of runtime overhead for μRAI, Full-SFI, and baseline.

4.6.3 Runtime Overhead

For defense mechanisms to be deployable, they must result in low performance overhead [26]. 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 4.16 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 4.3 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.

Table 4.3.
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**	μ**RAI**	
	# 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
μ**RAI Overhead**			2 + P

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 4.4 shows both the 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.

4.6.4 FLT Encoding Analysis

A central component of μRAI is its encoder (see subsection 4.5.2) and how efficiently it configures and populates the FLTs to reduce the effects of path explosion on the memory overhead. As discussed in subsection 4.5.2, 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

Table 4.4.
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

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 4.5 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 subsection 4.4.3. To demonstrate this effect, we show the FLT sizes both with and without segmentation in Table 4.6. Averaged across all applications, segmentation reduces FLT sizes by 78.1%. Intuitively, PinLock can be instrumented without segmentation. As mentioned in subsection 4.5.3, 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.

Table 4.5.
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

Table 4.6.
Summary of the segmentation effect on FLT size.

App	Without Segmentation			Segmented			Ave.
	Min.	Max.	Ave.	Min.	Max.	Ave.	Reduction
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%

4.6.5 Encoder Efficiency Discussion

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 4.9. Table 4.7 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., subsection 4.5.1). 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 subsection 4.5.3). 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.

Table 4.7.
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%

4.6.6 Scalability Analysis

μRAI scales its encoding scheme for larger applications by partitioning the call graph whenever no satisfying keys are found as shown in Figure 4.8. 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 4.8 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).

4.6.7 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 4.17(a) shows the

Table 4.8.
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

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.

The Flash increase of μRAI's instrumentation, exception handler SFI (EH-SFI), and type-based CFI is shown in Figure 4.17(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).

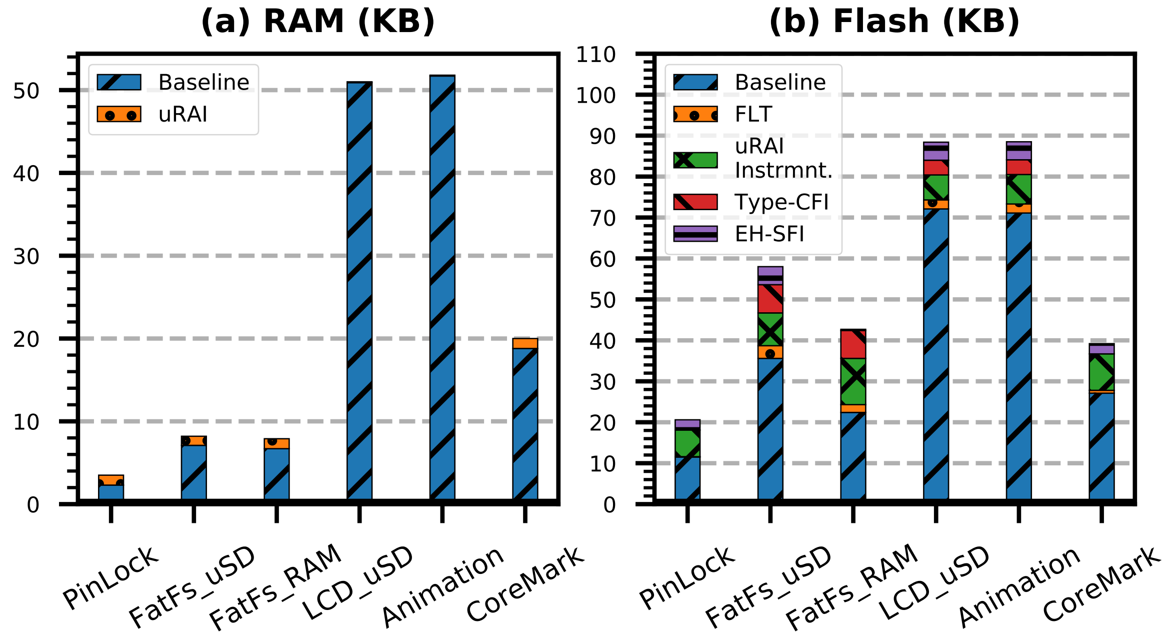


Fig. 4.17. Illustration of μRAI's memory overhead.

4.7 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 [26, 54, 55, 65, 152–154] 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 [14] attests the integrity of the control-flow by calculating a hash chain of the executed control-flow. LiteHAX [18] 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 [12] 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 [9] automates compartmentalizing the firmware on a finer-grained intra-thread level. TyTan [16] and TrustLite [17] 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 [45] 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 uXOM [46] which enables efficient implementation of XoM for Cortex-M processors. EPOXY [8] uses a modified and randomized location of a SafeStack [61] 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 [15] 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 [68]). CFI-CaRE [13] 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 [69] 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% [13, 69]). μRAI enforces the RAI property without requiring a TEE and with a modest runtime overhead.

4.8 Discussion

4.8.1 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 [155]. These however are an orthogonal problem to stack safety and may be protected through, e.g., OTI [156].

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.

4.8.2 Miscellaneous

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 [8]. 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 [8].

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 [157]. 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.

4.9 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.

5. CONCLUSION

Internet of Things systems are omnipresent, ranging smart home systems and smart cities, to critical infrastructure such as industrial control systems. Although deployed in security and safety critical domains, such systems lack basic mitigations against control-flow hijacking attacks. As a result, IoT systems have become a prominent target of remote attacks, causing some of the largest Distributed Denial-of-Service attacks, and resulting in power grid blackouts.

To improve the security posture of IoT systems, we survey previous defense mitigations to secure IoT systems in Chapter 2. Building off our survery, We identify two main issues in IoT systems security. First, efforts to protect IoT systems are hindered by the lack of realistic benchmarks and evaluation frameworks. Second, existing solutions to protect from control-flow hijacking on the return edge are either impractical or have limited security guarantees.

This thesis address the first issue in Chapter 3. We present BenchIoT, a benchmark suite of five realistic IoT applications and an evaluation framework that enables automated and extensible evaluation of 14 metrics covering security, performance, memory usage, and energy. BenchIoT enables automated evaluation and comparison of security mechanisms.

Chapter 4 introduces μRAI, a compiler mitigation enforce the Return Address Integrity property. μRAI prevents all control-flow hijacking attacks on the return edges with negligible runtime overhead. μRAI does not require special hardware extensions and thus is practical and secure solution for IoT systems.

This thesis demonstrates that standardized benchmarks and metrics enable measuring security properties on heterogeneous embedded IoT systems. Moreover, it proves the effectiveness of customized static analysis and runtime monitors in preventing control-flow hijacking attacks on return edges at low runtime overhead. Combined, this thesis advances the state-of-the-art of protecting IoT systems and benchmarking its security.

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] X. Li, X. Liang, R. Lu, X. Shen, X. Lin, and H. Zhu, "Securing smart grid: cyber attacks, countermeasures, and challenges," *IEEE Communications Magazine*, vol. 50, no. 8, 2012.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] N. S. Almakhdhub, A. A. Clements, M. Payer, and S. Bagchi, "Benchiot: A security benchmark for the internet of things," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 234–246.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[20] M. Larabel and M. Tippett, "Phoronix test suite," *Phoronix Media,[Online]. Available: http://www. phoronix-test-suite. com/.[Accessed June 2016]*, 2011.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on.* IEEE, 2001, pp. 3–14.

[22] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, 1984.

[23] J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: open benchmarks for energy measurements on embedded platforms," *CoRR*, vol. abs/1308.5174, 2013. [Online]. Available: http://arxiv.org/abs/1308.5174

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

[25] N. S. Almakhdhub, A. A. Clements, S. Bagchi, and M. Payer, "μRAI: Securing embedded systems with return address integrity," in *Proc. of ISOC Network & Distributed System Security Symposium (NDSS). https://hexhive. epfl. ch/publications/files/20NDSS. pdf*, 2020.

[26] 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.

[27] K. S. Lab, "Car hacking research: Remote attack tesla motors," https://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars, 2016.

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

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

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

[31] Y. Li, J. M. McCune, and A. Perrig, "Viper: verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 3–16.

[32] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "{VRASED}: A verified hardware/software co-design for remote attestation," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1429–1446.

[33] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "Seda: Scalable embedded device attestation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 964–975.

[34] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "Darpa: Device attestation resilient to physical attacks," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016, pp. 171–182.

[35] ARM, "cortex-a53," https://developer.arm.com/products/processors/cortex-a/cortex-a53.

[36] raspberrypi, "raspberry-pi-3-model-b," https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[37] "Kernel source tree for Raspberry Pi Foundation," https://github.com/raspberrypi/linux/blob/e2d2941326922b63d722ebc46520c3a2287b675f/Documentation/features/vm/ELF-ASLR/arch-support.txt, 2020.

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

[39] NXP, "FRDM-K64F," https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F, 2020, 2020.

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

[41] ——, “Arm architecture reference manual,” https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf, 2014.

[42] Microchip, “Avr32 architecture document,” http://ww1.microchip.com/downloads/en/devicedoc/doc32000.pdf, 2011.

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

[44] 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/.

[45] 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.

[46] 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

[47] 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.

[48] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, “Remote attestation to dynamic system properties: Towards providing complete system integrity evidence,” in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 115–124.

[49] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, “Lo-fat: Low-overhead control flow attestation in hardware,” in *Proceedings of the 54th Annual Design Automation Conference 2017*, 2017, pp. 1–6.

[50] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, “Atrium: Runtime attestation resilient under memory attacks,” in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 384–391.

[51] 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.

[52] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block oriented programming: Automating data-only attacks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1868–1882.

[53] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, “Diat: Data integrity attestation for resilient collaboration of autonomous systems,” in *Network and Distributed Systems Security Symp.(NDSS)*, 2019.

[54] 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.

[55] 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.

[56] A. Cui and S. J. Stolfo, “Defending embedded systems with software symbiotes,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 358–377.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

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

[64] 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.

[65] 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.

[66] G. Ramalingam, “The undecidability of aliasing,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[67] 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.

[68] 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.

[69] 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.

[70] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 263–277.

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

[72] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[73] Z. Liu and J. Criswell, "Flexible and efficient memory object metadata," *ACM Sigplan Notices*, vol. 52, no. 9, pp. 36–46, 2017.

[74] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[75] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 381–392.

[76] D. Midi, M. Payer, and E. Bertino, "Memory safety for embedded devices with nescheck," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 127–139.

[77] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*. Springer, 2005, pp. 115–148.

[78] ARM, "Armv8-m architecture reference manual," https://static.docs.arm.com/ddi0553/a/DDI0553A_e_armv8m_arm.pdf, 2020.

[79] M. Bishop, M. Dilger *et al.*, "Checking for race conditions in file accesses," *Computing systems*, vol. 2, no. 2, pp. 131–152, 1996.

[80] 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.

[81] 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.

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

[83] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[84] J. L. Henning, "Spec cpu2006 memory footprint," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 84–89, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1241601.1241618

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

[86] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster, "Tardis: software-only system-level record and replay in wireless sensor networks," in *Proceedings of the 14th International Conference on Information Processing in Sensor Networks.* ACM, 2015, pp. 286–297.

[87] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with envirolog," in *Proceedings of 25TH IEEE International Conference on Computer Communications (INFOCOM).*

[88] "BenchIoT," https://github.com/embedded-sec/BenchIoT.

[89] ARM, "Optional memory protection unit," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/BIHJJABA.html.

[90] ——, "Cortex microcontroller software interface standard," https://developer.arm.com/embedded/cmsis.

[91] J. Salwan, "ROPgadget - Gadgets Finder and Auto-Roper," http://shell-storm.org/project/ROPgadget/, 2011.

[92] J. Ko, K. Klues, C. Richter, W. Hofer, B. Kusy, M. Bruenig, T. Schmid, Q. Wang, P. Dutta, and A. Terzis, "Low power or high performance? a tradeoff whose time has come (and nearly gone)," in *European Conference on Wireless Sensor Networks.* Springer, 2012, pp. 98–114.

[93] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *Security and Privacy (SP), 2017 IEEE Symposium on.* IEEE, 2017, pp. 268–286.

[94] T. Does, D. Geist, and C. Van Bockhaven, "Sdio as a new peripheral attack vector," 2016.

[95] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, 2015.

[96] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *Security and Privacy (SP), 2010 IEEE Symposium on.* IEEE, 2010, pp. 447–462.

[97] ARM, "Mbed-SDK," https://os.mbed.com/handbook/mbed-SDK.

[98] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016.

[99] E. Ronen, A. Shamir, A.-O. Weingarten, and C. OFlynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *Security and Privacy (SP), 2017 IEEE Symposium on.* IEEE, 2017, pp. 195–212.

[100] "Smart sensors for electric motors embrace the IIoT," https://www.engineerlive.com/content/smart-sensors-electric-motors-embrace-iiot, 2018.

[101] "Industrial Sensors and the IIoT," https://www.motioncontrolonline.org/content-detail.cfm/Motion-Control-Technical-Features/Industrial-Sensors-and-the-IIoT/content_id/1716, 2016.

[102] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart nest thermostat: A smart spy in your home."

[103] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation." in *NDSS*, 2013.

[104] T. Kim, C. H. Kim, H. Choi, Y. Kwon, B. Saltaformaggio, X. Zhang, and D. Xu, "Revarm: A platform-agnostic arm binary rewriter for security applications," 2017.

[105] ARM, "Mbed-uVisor," https://github.com/ARMmbed/uvisor.

[106] H. Tan, W. Hu, and S. Jha, "A remote attestation protocol with trusted platform modules (tpms) in wireless sensor networks." *Security and Communication Networks*, vol. 8, no. 13, pp. 2171–2188, 2015.

[107] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 1–16.

[108] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *Security and privacy, 2004. Proceedings. 2004 IEEE symposium on.* IEEE, 2004, pp. 272–282.

[109] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.* ACM, 2017, pp. 193–204.

[110] FreeRTOS-MPU, "FreeRTOS-MPU," https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html, 2019.

[111] Mbed-uVisor, "mbed OS can't sleep when uVisor is enabled," 2017, https://github.com/ARMmbed/uvisor/issues/420, 201.

[112] T. J. McCabe, “A complexity measure,” *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[113] J. Pallister, S. J. Hollis, and J. Bennett, “Identifying compiler options to minimize energy consumption for embedded platforms,” *The Computer Journal*, vol. 58, no. 1, pp. 95–109, 2013.

[114] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder, “Static analysis of energy consumption for llvm ir programs,” in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2015, pp. 12–21.

[115] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, “Exploiting dynamic timing margins in microprocessors for frequency-overscaling with instruction-based clock adjustment,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015, pp. 381–386.

[116] EEMBC, “Iotmark - an eembc benchmark,” https://www.eembc.org/iot-connect/about.php.

[117] ——, “Securemark - an eembc benchmark,” https://www.eembc.org/securemark/.

[118] Z. B. Celik, P. McDaniel, and G. Tan, “Soteria: Automated iot safety and security analysis,” in *USENIX Annual Technical Conference (USENIX ATC)*, Boston, MA, 2018.

[119] M. Arlitt, M. Marwah, G. Bellala, A. Shah, J. Healey, and B. Vandiver, “Iotabench: an internet of things analytics benchmark,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 133–144.

[120] A. Shukla, S. Chaturvedi, and Y. Simmhan, “Riotbench: An iot benchmark for distributed stream processing systems,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.

[121] S. M. Z. Iqbal, Y. Liang, and H. Grahn, “Parmibench-an open-source benchmark for embedded multiprocessor systems,” *IEEE Computer Architecture Letters*, vol. 9, no. 2, pp. 45–48, 2010.

[122] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” in *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*. IEEE, 1997, pp. 330–335.

[123] V. Zivojnovic, “Dsp-stone: A dsp-oriented benchmarking methodology,” *Proc. of ICSPAT’94*, 1994.

[124] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The mälardalen wcet benchmarks: Past, present and future,” in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[125] F. H. McMahon, “The livermore fortran kernels: A computer test of the numerical performance range,” Lawrence Livermore National Lab., CA (USA), Tech. Rep., 1986.

[126] 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

[127] 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.

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

[129] 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.

[130] 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.

[131] 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.

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

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

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

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

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

[137] 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

[138] 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.

[139] 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.

[140] 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.

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

[142] 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

[143] 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.

[144] 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.

[145] 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.

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

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

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

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

[150] 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.

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

[152] 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.

[153] 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.

[154] 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.

[155] 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.

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

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

A. HANDLING SPECIAL RECURSIVE FUNCTIONS

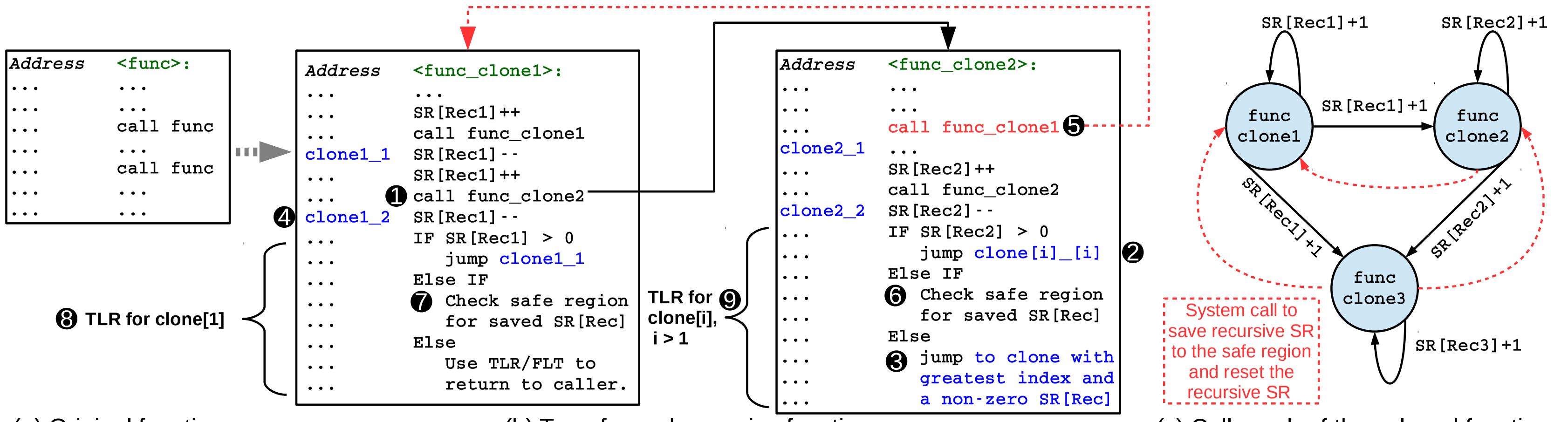
Special cases of recursion such as functions with multiple recursive calls require additional instrumentation. In the following we discuss two possible designs to support the RAI property. The first provides flexible handling of recursion by mainly utilizing the safe region for special cases of recursion.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

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 A.1(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.

Handling Special Recursion Using Function Cloning: 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 A.1(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 A.1(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 *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}



```
Address   <func>:
...       ...
...       ...
...       call func
...       ...
...       call func
...       ...
...       ...
```

(a) Original function

```
Address      <func_clone1>:
...          ...
...          SR[Rec1]++
...          call func_clone1
clone1_1     SR[Rec1]--
...          SR[Rec1]++
...        ❶ call func_clone2
❹ clone1_2   SR[Rec1]--
...          IF SR[Rec1] > 0
...             jump clone1_1
...          Else IF
...          ❼ Check safe region
...             for saved SR[Rec]
...          Else
...             Use TLR/FLT to
...             return to caller.
```

```
Address      <func_clone2>:
...          ...
...          ...
...          call func_clone1 ❺
clone2_1     ...
...          SR[Rec2]++
...          call func_clone2
clone2_2     SR[Rec2]--
...          IF SR[Rec2] > 0
...             jump clone[i]_[i]      ❷
...          Else IF
...          ❻ Check safe region
...             for saved SR[Rec]
...          Else
...          ❸ jump to clone with
...             greatest index and
...             a non-zero SR[Rec]
```

(b) Transformed recursive function

(c) Call graph of thee cloned functions

Fig. A.1. 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.

Algorithm 1 Multiple indirect recursion call procedure

```
procedure INDIRECT CALL[I] OUT OF N INDIRECT CALLS
    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
        Save the SR and reset $SR_{Rec}$
    end if
    $SR_{Rec[i]} = SR_{Rec[i]} + 1$
    Execute indirect call
    $SR_{Rec[i]} = SR_{Rec[i]} - 1$
    if All $SR_{Rec}counters = 0$ and Safe region contains saved $SR$ then
        Restore saved $SR$
    end if
end procedure
```

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 A.1(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 A.1(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).

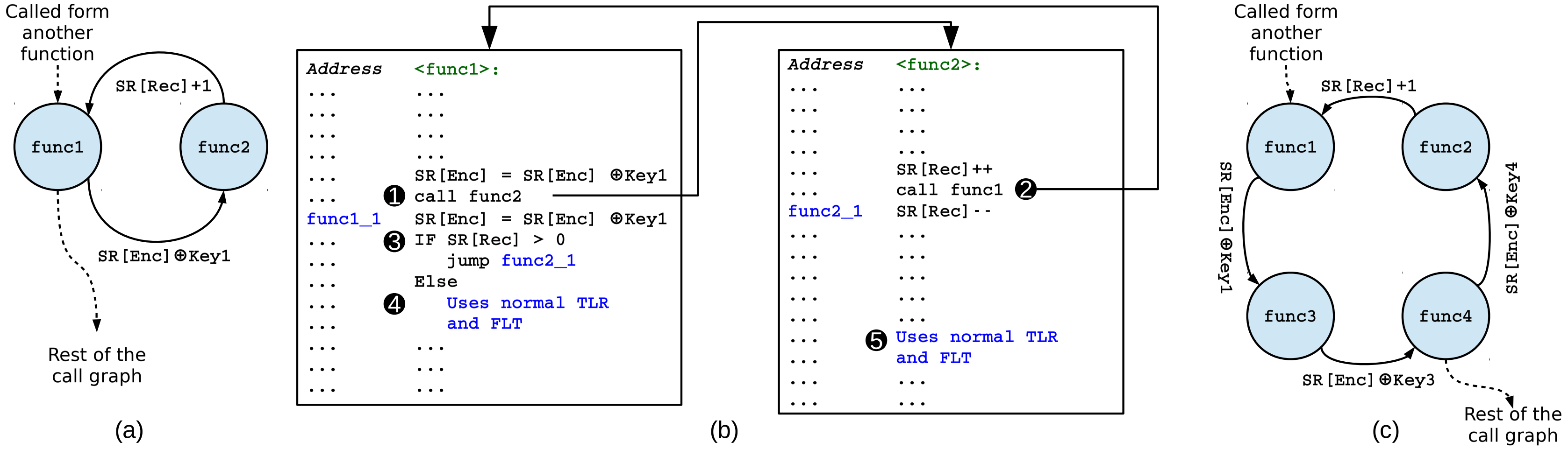
Indirect recursion is another special case that requires additional care. To understand the difference of indirect recursion, consider the conceptual call graph illustration shown in Figure A.2(a). Indirect recursion causes a cycle in the call graph, as demonstrated by `func1` and `func2`. In such a 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 subsection 4.4.4). 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

Algorithm 2 Multiple indirect recursion return procedure

```
procedure Resolving correct return target of out of N indirect calls
// Check recursion counters in descending order
    if $SR_{Rec[N]} \geq 0$ then
        Return after indirect recursive call N
    end if
    if $SR_{Rec[N-1]} \geq 0$ then
        Return after indirect recursive call N-1
    end if
// Inline the check until the lowest index of recursion counters
// ...
// After inlining all checks
    if All $SR_{Rec}$ equal 0 then
        //All indirect recursion have been resolved
        Use the regular TLR
    end if
end procedure
```

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 A.2(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



```
Address    <func1>:
...        ...
...        ...
...        ...
...        ...
...        SR[Enc] = SR[Enc] ⊕Key1
...      ❶ call func2
func1_1    SR[Enc] = SR[Enc] ⊕Key1
...      ❸ IF SR[Rec] > 0
...           jump func2_1
...        Else
...      ❹    Uses normal TLR
...           and FLT
...        ...
...        ...
...        ...
```

```
Address    <func2>:
...        ...
...        ...
...        ...
...        ...
...        SR[Rec]++
...        call func1 ❷
func2_1    SR[Rec]--
...        ...
...        ...
...        ...
...        ...
...        ...
...      ❺ Uses normal TLR
...        and FLT
...        ...
...        ...
```

Fig. A.2. 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).

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 A.2(c).

Finally, in case of indirect recursion with multiple call sites, each call site uses a segment counter, as shown in 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 above, 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 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 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.

VITA

Naif Saleh Almakhdhub is a PhD candidate in the electrical and computer engineering department at Purdue University, where he is advised by Prof. Mathias Payer and Prof. Saurabh Bagchi. His research focuses on protecting systems software against control-flow hijacking attacks, with an emphasis on applying these protections to embedded and IoT systems. He holds a Bachelor of Science (BSc) degree in electrical engineering from King Saud University, and a Master of Science (MS) degree from the electrical engineering and computer science department at Northwestern University.