Security for Software on Tiny Devices

Saurabh Bagchi

1 Introduction

At over 9 billion embedded processors in use today, the number of embedded devices has surpassed the number of humans. With the rise of the "Internet of Things" (IoT), the number of embedded devices, their complexity, and their connectivity is exploding. These smart *"things"* include fitness trackers, smart light bulbs, smart thermostats, Amazon's Dash Button, utility smart meters, smart locks, and smart TVs. Microcontrollers executing bare-metal software have been embedded deeply into larger systems. These embedded microcontrollers are often overlooked but they control vital components of our systems, e.g., network cards, wireless controllers, hard drive controllers, SD memory cards, or near field communication in cellphones. Many of these devices (or components in devices) are low cost with software running directly on the hardware, known as "bare-metal systems." In such systems, the application runs as privileged low-level software with direct access to all processor registers, the entire available memory, and all peripherals. This is in contrast to systems with an operating system that provides isolation and manages access to security-sensitive resources. These bare-metal systems must satisfy strict execution timing guarantees, while running on constrained hardware platforms with power and dollar constraints.

Society relies on these systems to provide secure and reliable computation, communication, and data storage. Yet, they are built with security paradigms that have been obsolete for several decades. Embedded systems are generally deployed without any active defenses or mitigations and do not follow common design criteria to enforce least privileges and restricted access. Defenses that are well known for

This article summarizes work done jointly with Mathias Payer (ETH Zurich) and also reflects further discussions with Stephen Checkoway (Oberlin College) and Mathias.

S. Bagchi (✉)
Purdue University, West Lafayette, Indiana, USA
e-mail: sbagchi@purdue.edu

L. Wang et al. (eds.), *System Dependability and Analytics*, Springer Series in Reliability Engineering, https://doi.org/10.1007/978-3-031-02063-6_7

desktops to protect against code injection, control-flow hijacking, or data corruption attacks are missing on embedded systems.

1.1 Ravi's Contributions on This Topic

Ravi has made some fundamental contributions on this topic. These have come more recently in the problem areas of security for teleoperated surgical robots [8, 11] and customized malware (and its countermeasure) for cyber-physical systems [10], with expanded focus on autonomous vehicular systems [4, 17] and the smart grid [13, 19]. We have learned of some design elements from these works and the community has adopted many principles and techniques for rigorous evaluation from them.

1.2 Why can't We "just" Adopt Defenses from the Server World to the Embedded World?

Protecting embedded devices in the presence of vulnerabilities poses unique challenges that are fundamentally different from desktop or server systems. Therefore, simply porting existing defenses is not an option. First, embedded systems often run directly on the hardware without an intermediate operating system or virtualization layer. The program itself is responsible for mediating access to all resources, including security-critical ones, among all the tasks. Second, due to the lack of a Memory Management Unit (MMU), embedded systems have a single flat address space where all memory locations (e.g., the locations of I/O ports) are static. Third, embedded systems are custom tailored to a specific purpose. Each type of system may have a specific hardware configuration where some I/O ports are security sensitive while others are not. Orthogonally, note that desktop defenses are incomplete and cannot defend against all code reuse attacks or information leaks, as shown through any recent attack that bypasses all existing defenses [23, 25]. Leveraging buffer overflows, use-after-free bugs, integer overflow, or type confusion vulnerabilities, adversaries can leak information and compromise software running on desktop systems despite all currently used defenses.

Protecting software against control-flow hijacking, code reuse attacks, and information disclosure is challenging for embedded systems. However, the embedded system environment also provides some unique opportunities that enable strong, novel defenses. First, whole program analysis on these systems is feasible. Due to cost, power, and environmental constraints, the software code base running on these systems is usually kept small. Best coding practices result in limited stack depth, restricted use of indirect control-flow, limited use of recursion, and fixed memory allocation. These bound the exploration space so that static analysis can be applied to entire programs. Second, the source code of each component is generally available to the developers as all components are developed by the same company. Even when libraries are used, all code is compiled to a monolithic binary and combined through Link Time Optimization (LTO). Third, both the software running on an embedded system and the underlying hardware are single purpose, further simplifying the analysis. The software running on embedded systems has limited functionality, often with a single purpose—compared to desktop systems with hundreds of parallel processes. Similarly for the hardware, each hardware unit is dedicated to a single executing process, whereas on desktop systems, the device is shared among multiple processes. The combination of these opportunities enables us to scale static and dynamic analysis techniques to full embedded systems and to devise strong protection mechanisms that respect the above-mentioned domain-specific constraints.

Our solution approach: RESIN. In our prior and ongoing work, we seek to solve the problem of protecting embedded systems against a wide variety of attacks, without the need to rearchitect the entire application. Our approach has 3 inter-dependent high-level tasks. We show a schematic of our overall system in Fig. 1. The parts where the user/developer need to provide input are shown in the salmon colored boxes according to the legend.

1. **Task I: Guided IoT exploration**. In this task, we develop targeted static analysis to identify the control and the data flow in the program. This is augmented with an

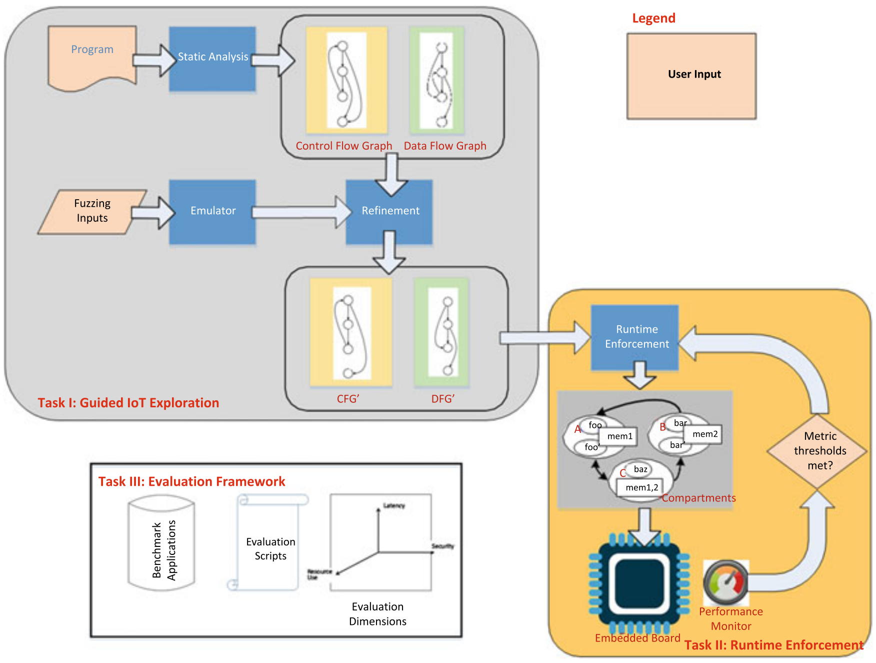

Fig. 1 Overview of the main components and their placement in the overall system RESIN

IoT emulator which emulates the runtime behavior under fuzzed and controlled inputs to discover information that is outside the scope of static analysis.

2. **Task II: Runtime enforcement**. Here we develop runtime enforcement techniques for enforcing the principle of least-privilege execution, which is considered standard security practice, but is absent in embedded system execution. We will operate with feedback provided by the constraints of the hardware and the performance impact due to the isolation of multiple compartments of code and data.
3. **Task III: Evaluation framework**. We develop a rich set of bare-metal system applications to stress different functionalities in representative use cases. We develop these benchmarks for a set of embedded boards that provide different hardware capabilities (e.g., access to different sensors) and develop scripts for evaluating different aspects of security and performance. The security and performance metrics combine the domain-agnostic as well as the domain-specific ones. The latter includes an understanding that performance needs to be deterministic for our target domain.

Target domains. We demonstrate the benefits of RESIN through realistic applications developed in five security-critical, target domains on real hardware.

1. **Smart homes**. Devices such as the Amazon Dash button, smart light bulbs, smart door locks, and per-room temperature sensors increase convenience in an modern home but, in the hands of an adversary, can result in safety and privacy hazards.
2. **Wearables**. We are increasingly tracking different aspects of our lives through heart rate monitors, activity trackers, smart shoes, or smart watches. These devices have access to highly personal data and may need to communicate urgent and critical health indicators.
3. **Smart cities**. Modern cities are increasingly connected with smart, battery-powered sensors placed in side walks and streets to detect pedestrians, bikes, and cars. These devices are low-powered, embedded, run real time, and communicate wirelessly. Protecting these devices is crucial for roadside safety.
4. **Connected transportation and infrastructure**. A modern car contains dozens of safety–critical, connected embedded devices that communicate over shared buses as well as over a variety of wireless interfaces including Bluetooth and 4G LTE. Vehicle-to-vehicle and vehicle-to-infrastructure communications are starting to be built and deployed.
5. **Industrial control systems**. Physical processes in industrial settings are computer-controlled. These safety–critical systems can be exploited to cause great harm.

Our work focuses on the sort of low-powered, embedded devices that are ubiquitous in these domains.

2 Background and Related Work

2.1 Embedded System Development

An embedded system is often meant to perform a dedicated function in contrast to a general purpose computer. Frequently this is a component within some larger system. In a vehicle, for example, there are multiple dedicated embedded computing systems, e.g., to control the anti-lock brakes, to monitor and control the vehicle's emissions, or to display information on the dashboard. The number of embedded systems has risen rapidly and today, less than 2 percent of microcontrollers manufactured are used in general-purpose computers [5]. An important sub-class of embedded systems have real-time requirements and therefore security mechanisms proposed by us or others, cannot afford to perturb the timing of the software to any significant extent. Importantly, for such systems, it is crucial to guarantee the timing properties and thus our security mechanisms must also minimize the variance in the timing that they introduce.

Certain common hardware constraints on embedded development are: (i) processing power—these devices are typically 16-or 32-bit, running at up to a few hundred MHz-s, driven by requirement of low dollar cost and low power consumption (say, 1 mW/MIPS to 10 mW/MIPS); (ii) memory—the RAM is typically up to a few 100 s of kBs and the flash which contains the program is typically up to a few MBs; (iii) slow buses—such as SPI or I^2C, which are relatively slow compared to the processor. Overall, our target class, the bare-metal embedded system, is a highly cost-conscious segment of the market. There are typically one or more peripherals attached to an embedded board. These provide functionality such as USB or sensing and they are typically accessed through a Hardware Abstraction Layer (HAL), which eases the programming by abstracting away low-level control signals and other details needed to access the peripherals. For specificity, let us consider one device that fits within our target domain. Figure 2 shows ARM's memory model for the ARMv7-M

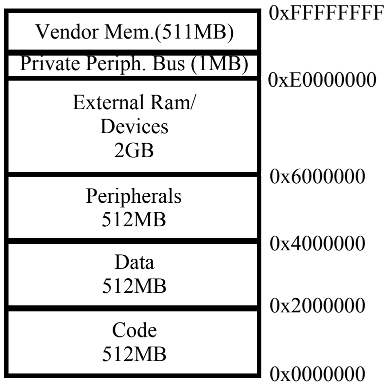

Fig. 2 ARM's memory model for ARMv7-M devices

architecture. It breaks a 32-bit (4 GB) memory space into several different regions. It is a memory-mapped architecture, meaning that all I/O is directly mapped into its memory space (peripherals, and external devices). While the architecture reserves large amounts of space for each area, actual devices only use a small portion of it. For example, the Cortex-M4 (STM32F479I) device we use in our evaluation has 2 MB of flash in the code area, 384 kB of RAM, and uses only a small portion of the peripheral space.

Embedded software development has traditionally been done mostly in C, with some limited use of assembly code. Low-level coding often requires close interaction with the hardware platform, even though the HAL does abstract away the lowest levels of hardware detail. Because of the hardware resource constraints, embedded software often is very compact, at the cost of readability and generalizability (to different hardware platforms). Software is usually written assuming any memory or peripheral can be accessed any time and from any region of the code. Further, the preferred mode of programming is event-driven programming, whereby the software reacts to external stimuli such as a sensor providing a value sensed from the physical world after running it through its analog-to-digital converter. There are expectations from embedded software that it will run unattended for long periods of time (say, months) and the end user will have limited ability (if at all) of programming the system "in the field." The vendor of the embedded board usually provides the compiler and linker tool chain to convert the C program to executable code. There has been robust development of LLVM-based toolchains for various target embedded platforms. We will leverage this trend by building our toolchain on top of LLVM as additional passes or modifications to existing passes.

Certain software design patterns frequently occur in embedded software. First, the software statically allocates all the memory that it will require, rather than relying on dynamically allocating memory. The second is the careful and parsimonious use of memory, such as fitting multiple, possibly unconnected, variables into a single register. Third, debugging tools in embedded development are more limited. At the powerful end of the spectrum is a JTAG-based debugger, often called an in-circuit emulator (ICE). In fact, in a 2015 survey of embedded developers, debugging was found to be the single greatest challenge [30].

A commonly found piece of hardware in our target class of devices is the *Memory Protection Unit (MPU)*. It enables setting privileges on regions of memory, which control read, write, and execute permissions for both privileged and unprivileged execution modes. On the ARMv7-M architecture for example, the MPU can define up to eight regions, numbered 0–7. Each region is defined by setting a starting address, size, and permissions. We assume, at the high end, machines with an MPU but without a *Memory Management Unit (MMU)*, such as Cortex ARM M0 to M4. These machines generally run in a 32-bit address space. Machines with an MMU are out of scope. The cost for adding an MMU to embedded systems is seen as prohibitive and does not fit current software design patterns where code runs bare-metal or with only a thin operating system layer. At the low end, we assume 8051-style or Atmel AVR-style Harvard-like machines without MPU or MMU. These machines generally run with a 16-bit address space, sometimes with multiple 16-bit address spaces, e.g., for ROM and RAM.

2.2 Threat Model

IoT devices are heavily connected and susceptible to different forms of attacks. For the research proposed here, we assume that the software running on the devices contains software flaws (bugs) that are reachable through potentially adversary-controlled outside input. Input, including malicious input, to the device can be local or remote. Local input is any input that requires close physical proximity to the IoT device such as a connection through local I/O, a serial port, or near field communication such as Bluetooth or ZigBee. An example of local input is data access through a diagnostic port. Remote input is something that can be sent over a network interface, such as WiFi. All IoT devices communicate with the cloud in some form. Remote input requires an internet connection. If the IoT device runs any services, any internet device can connect to those services. An example would be a listening telnet server.

Any data attack where an adversary connects to the device, intercepts a connection from the device, or uses an I/O port to communicate with the device is in scope. We assume that the software running on the IoT device has flaws that are reachable through adversary-controlled input. Our threat model includes both data confidentiality and data integrity attacks during the runtime of the device. Verifying the integrity of the device at boot time is out of scope. Cryptographic attacks that break encrypted communication with the cloud are out of scope but implementation bugs in cryptographic protocols remain in scope. Physical attacks—for example, flashing a new firmware onto the device—are out of scope.

2.3 Lack of Defenses on Embedded Systems

Due to hardware resource and development constraints, current IoT systems lack any mitigations against memory safety violations that people have become accustomed to for desktop and server systems. Full scale operating systems leverage virtual memory to isolate processes from each other. Best programming practices ensure that each process runs with least privileges and only communicates with other processes through a well defined API. Operating systems restrict access based on fine-grained permissions and capabilities along access control lists. Inside the process, the MMU ensures the separation between code and data (DEP) and allows segments to be placed at random locations whenever a process is started (ASLR). Additional mitigations such as stack canaries or control-flow integrity (CFI) [7, 29] may be added on a per-process basis as part of the compiler toolchain.

Compared to full hardware support and regular operating systems, embedded systems are much more constrained. First, the hardware is highly constrained and storage or memory overhead are hard to justify due to the additional cost. The embedded devices we target also do not have an MMU and, at best, use an MPU to overlay privileges on a flat physical address space. The lack of an MMU makes defenses such as ASLR impossible as they require a virtual address space. Second, embedded operating systems generally do not enforce isolation between the operating system kernel and the individual processes or even among processes—the memory structure is comparable to a set of threads that run in the same address space together with privileged software. The lack of separation between privileged code (kernel code) and unprivileged code (the applications) prohibits defenses such as DEP as all privileged memory and peripherals are directly reachable from unprivileged code. Third, the rigid compiler toolchain and development environment with lack of good debugging facilities hinders compiler innovation and prohibits the use of modern compiler-based defenses such as stack canaries or CFI. While these mitigations would have to be adapted for embedded systems to fit their unique constraints, there is no fundamental reason why they should not be used.

3 Guided IoT Exploration

IoT software is fundamentally different from desktop or server software. Both desktops and servers run multiple applications at different privileges (e.g., different users, separation between kernel and user-space, or virtualization). Low-end IoT devices are highly resource constrained. IoT devices have limited CPU, memory, power, and communication abilities and software is generally highly adapted to these devices. Due to this customization, existing software analysis techniques do not apply to IoT systems. In this task we develop static and dynamic analysis techniques that infer information about IoT applications. This information is then leveraged in task II to enforce strong security policies such as per-task compartmentalization, targeted memory safety to protect against control-flow hijacking and data-only attacks, and event-aware state protection. Note that all policies are geared towards the special circumstances IoT systems run in.

Advantages of IoT software are that application source code is generally available, the amount of code is manageable, and frameworks such as the ARM Mbed IoT platform [2] generalize common tasks such as access to peripherals. Unfortunately, these advantages are offset by several challenges to security in low-end embedded systems: (i) many embedded systems engineers have little or no security experience resulting in code that does not follow security best practices, (ii) programmers face the complexity of cross cutting concerns across all layers of the stack: from low-end I/O and pin management to high-end application concerns such as communicating with a backend server in the cloud, (iii) applications are developed in an ad hoc manner on stale tool chains (i.e., the compiler is rarely updated due to the complexity of setting up a cross-compilation tool chain), and (iv) lack of defense mechanisms, mitigations, and analysis methods (e.g., static analysis or fuzzing) that are used ubiquitously on desktop and server software.

Preliminary work. We have broad experience in protecting different forms of embedded systems and in developing sanitizers and mitigations to find a wide variety of vulnerabilities. Our most recent work to protect bare-metal embedded devices is EPOXY [12]. EPOXY is an LLVM-based mitigation that enforces a light privilege overlay, dropping privileges for all instructions and selectively raising privileges for a few privileged operations such as writing to I/O registers. Based on this privilege overlay, we enforce data execution prevention to prevent code injection, a safe stack [18] to protect against return oriented programming, and diversification to protect against data-only attacks.

Earlier, we have developed a more holistic defensive approach that enforces full memory safety for tiny embedded systems through nesCheck [20]. This work leverages a CCured-like [21] pointer analysis that classifies pointers as safe, sequence, or dynamic, allowing different instrumentation depending on the type of the pointer. The overheads for nesCheck are higher than for an EPOXY-based approach. Both EPOXY and nesCheck protect different classes of embedded systems against wide types of attack vectors at low overhead, adhering to performance and power constraints of embedded devices.

Orthogonally, we have developed a wide set of sanitizers that enforce security policies for regular software systems such as Desktops or servers. We have worked on Control-Flow Integrity [7, 9, 14, 22, 29], a mitigation that protects against control-flow hijacking by checking that the target of control-flows observed at runtime belongs to the set of valid targets. As an extension to CFI, we have explored a mechanism that keeps state for variadic function calls [6], allowing us to make the relationship between caller and callee explicit and to check argument types whenever they are used. The arguments of variadic functions (e.g., printf) depend on an implicit contract between caller and callee and cannot be checked statically by the compiler. Our mechanism enforces a dynamic runtime integrity check to ensure that the arguments pushed by the caller are correctly used by the callee.

Type confusion [15, 16] is another attack vector that enables memory corruption as a secondary effect. Type confusion abuses differences between object sizes to compromise systems. Our mechanisms track type information of all live objects to ensure type integrity for all type conversions and type checks.

Approach. To address the lack of defenses, we require detailed information about individual IoT applications to automatically employ defenses given the sparse resources available on IoT platforms. In a first step, we therefore propose novel static and dynamic analysis methods to recover necessary information about the IoT applications. We address the diversity of the IoT environment by developing a hardware abstraction language that encapsulates the differences between individual instruction set architectures, resource configurations, and availability of sensors and actuators in a portable manner. Second, we develop an event-aware static analysis that decodes event loops of embedded devices. As a proof of concept defense, we develop an event-aware version of CFI for IoT devices. Third, we develop an emulator to simulate different IoT configurations and software. IoT applications are often event-based, so we need precise knowledge of different program paths and interactions with the underlying hardware (such as sensors and actuators), based on a hardware configuration. Indirect control-flow transfers remain challenging for any static analysis due to the aliasing problem. We use emulator-based tracing to handle the limitations of static analysis in handling indirection. We leverage the fact that IoT applications often have constrained control paths that they execute. This will allow the community to test their IoT software for security weaknesses, allowing precise bug discovery and targeted patches for vulnerable software.

4 Runtime Enforcement Techniques

In Task I, we create accurate control and data flow graphs using both static analysis and fuzzed data inputs. In this task, we take this information and automatically, in a policy-driven manner, create containers of code, data and peripherals, which serves as fault containment domains. Here we develop graph theoretic algorithms on the above-mentioned graphs to enforce the principle of least privilege (Task II.1) and then we enforce isolation through compartments, which are realized through available hardware resources, however scarce they may be (Task II.2). We then monitor the execution of the application with the initial degree of compartmentalization and incrementally change it if the performance impact is unacceptable (Task II.3).

Preliminary work. We have used the MPU, commonly available in embedded devices, in pre-liminary work [12] to create a proof-of-concept called EPOXY with simply two privilege levels of software. This provides the foundation on which code integrity, adapted control-flow hijacking defenses, and protections for sensitive I/O can be applied, by building on the "two privilege level" idea. We have evaluated the performance of our combined defense mechanisms for a suite of 75 benchmarks and 3 real-world IoT applications. Our results for the application case studies show that EPOXY has, on average, a 1.8% increase in execution time and a 0.5% increase in energy usage; however, the worst-case execution overheads will make the technique unusable for many applications. There are some specific technical constraints imposed by each generation of MPU, such as, for the MPU on the ARMv7-M architecture, each region must be a power of two in size, greater than 32 and start at a multiple of its size (e.g., if the size is 1 kB then valid starting addresses are multiples of 1 kB). Regions can overlap with the high numbered region's permissions taking effect. The MPU's hardware restrictions significantly constrain the design of compartments. For example, of all the MPU registers available (only 8 to start off with), several are used for enforcing basic protections such as making the code region not writable. The number of compartments available at any point in the execution is restricted to those that are remaining.

The use of MPUs to create isolation boundaries has been proposed and developed by ARM in its mBedOS platform, through a software module called μVisor [3]. Using this, the ARM development environment *allows* a developer, but does not provide any automation support, to create multiple "boxes." Each box gets its own memory region, including stack, and interactions among boxes are monitored and allowed/disallowed by trusted code called “gateways.” The usability challenge with the current concept is daunting. In our work, we fundamentally reduce this usability barrier by providing novel techniques to automatically infer data and control flow, and from that and the policies for security enforcement, automatically create the isolated containers.

4.1 Task II.1: Automatic Least Privilege Separation

In this task, we take the control flow and data flow graph created in Task I and create compartments out of them to achieve the desired goal of least privilege execution. The graph nodes are partitioned into disjoint compartments with the invariant that at any point of time in the execution, only a single code region belonging to the currently active compartment is executing. Further, that code region only has access to the data regions and peripherals that are within that compartment. The graph algorithms will have the goal of creating the appropriate-sized compartments, balancing the needs of the performance overhead and hardware resources used versus the level of privilege separation achieved. To expand on this, if there are fewer compartments, then there is less performance overhead and hardware resources (such as, MPU regions) used, but there is a higher degree of privilege to more code regions, thus reducing security.

We develop the graph algorithms using both static and dynamic information (from Task I). The static information contains the graph structure—the nodes and the edges, while the dynamic information annotates the edges with the frequency and nature of interactions. The latter can include for example the amount of data being passed among the compartments. We design and develop three variants of graph algorithms of progressive complexity. In all of these, we use the insight that the graphs for embedded software are likely to be much smaller than for general-purpose software and are relatively sparse in terms of indegree and outdegree.

1. No code or data motion: This will operate without a feedback loop and create compartments in one shot. Thus, this will not take into account the possibility of moving code or data to create more compact compartments.
2. Automatic code or data motion: This will run in multiple passes (we anticipate 2–3) where each pass will indicate the quality of compartmentalization and this will trigger some movement of code regions or data regions to create more compact compartments. The necessity of multiple passes arises because there is a coupling of the two steps—the creation of the compartments and the layout of code and data in memory.
3. Programmer annotation: This will be driven by an objective function where the amount of exposed code at any point in the execution needs to be minimized subject to the hardware and the performance constraints. The exact performance impact may not be known at the outset and will be fed back as input from Task II.3. If this objective function does not reach a certain specified value, which practically speaking is likely to be specified as an improvement over the

baseline, then the programmer will be requested for annotation about criticality of code regions. Alternately, the criticality can be inferred by doing some form of scalable taint tracking [32] to determine which code regions are more susceptible to unvalidated user input.

4.2 Task II.2: Enforcing Isolation Among Compartments

The goal of this task is to put in place the embedded software to enforce isolation among compartments for control and data. A compartment may access data only within its own compartment or some data that is explicitly marked as shareable. Control flow can go from one compartment to another compartment with the mediation of some privileged code, which will validate that the transition is allowed, as determined by one or more of static analysis, paths learned through fuzzing, or developer annotation. Such privileged code will form part of the trusted computing base for our system RESIN and will thus have to be minimized.

The way we envisage this task working is that the embedded program will be instrumented to trigger the privileged code whenever the code region within compartment A invokes the code region in compartment B. For example, if the code granularity is simply a function, then the call and the return instructions can be instrumented. The privileged code enforces the appropriate check, namely, that a control flow transfer is allowed here. This can be inferred from the static analysis, augmented with the trace-based emulation. If there are further violations detected at runtime, then this will be stored in a trace, for further offline, post-mortem debugging. We expect that such a trace will be highly compressible, drawing from our prior insights from deterministic record and replay in such embedded platforms [28]. The insight here is from the regular pattern of embedded application executions, as introduced in subsection 2.1.

We use MPU permissions to enforce the compartment-specific constraints. An MPU register can designate a contiguous region of memory to be read/write/execute, for privileged or unprivileged code. However, the number of registers is limited (8 in current ARMv7-M architecture, 16 in the next generation). Therefore some compartments have to be merged. This can be achieved by a mix of code and data motion and increasing the range of addresses accessible to some code regions. In general, the more interconnected the CFG and DFG are, the more challenging it will be to move all the relevant code and data regions into the same compartment. Our initial examination of baremetal applications (as in [12, 20]) has shown that the graphs have a bi-modal characteristic—some parts are sparse (where the code accesses a few libraries and no other code region is dependent on it), while some parts are dense (code regions in the hardware abstraction layer which are accessed by multiple higher-level code regions).

A broader design space that we need to consider is isolation versus resource requirements, e.g., separate stack for each compartment versus shared stack. If it is a shared stack, then portions of the stack will have to be protected, such as, only some parts of the caller's stack should be accessible from the callee. This results in a greater requirement for MPU registers. But if the caller and the callee stacks are kept separate, then this has higher overhead in terms of the memory usage and the runtime overhead of switching between the stacks of the different compartments. Note that mBedOS, the open source embedded operating system from ARM that runs on the Cortex-M microcontroller, requires separate stacks for each compartment ("box" in their terminology).

5 Evaluating Security

Building defenses for embedded systems is only worthwhile if the defenses stop attacks without compromising correctness, performance, or energy usage. In essence, we need an objective method to measure the characteristics of interest before and after applying the defense.

5.1 IoT Metrics

Meaningful metrics are an essential component of any evaluation methodology. We would like to be able to say that Approach A provides more security than Approach B with respect to Attacker C. Unfortunately, good qualitative and quantitative metrics for security have thus-far proved elusive. The difficulty of constructing useful metrics is, in some respects, intrinsic to security. As an illustration of this difficulty, consider hardware-enforced, per-page memory protections with a write-xor-execute (W X) policy where no page of memory can be both writable and executable. Computer systems where this policy is strictly enforced (e.g., in Apple's iOS) appear to be more secure than computer systems without such a policy-enforcement because the policy prevents attackers from injecting and executing new code as well as modifying existing code. More advanced exploitation techniques, such as return-oriented programming, may still allow attackers to exploit vulnerabilities in the system. It is thus difficult to say that W X enforcement leads to increased security compared to no enforcement. In essence, it is difficult to quantify security gains from a defense mechanism, even if that mechanism rules out entire classes of attack techniques.

Despite the difficulty, several approaches to measuring the security of control-flow hijacking defenses have been previously proposed. The first is a tool-based approach wherein a tool like ROPgadget [24] is run over a binary to determine the number of return-oriented programming gadgets existing before and after a defense is applied. The second approach tries to quantify how much an indirect control transfer instruction's target set size has been reduced [31]. Both approaches were the best metrics at the time of their introduction; however, they are flawed and do not provide a sufficient notion of security.

Instead, we will develop qualitative and quantitative metrics for IoT security building on our preliminary work [7]. Qualitatively, we consider the defensive mechanisms' strengths in terms of the classes of attacks mitigated by the mechanism. For example, a defense mechanism can be evaluated in one dimension by considering whether it allows attacker-controlled memory writes to memory-mapped I/O registers or not. Whereas [7] uses the sets of instructions that can be targeted by control-flow instructions, we will use our analyses from Task I to abstract the notion of target sets to sets of input constraints for transferring control to those instructions. Similarly, we will abstract sets of memory locations that can be written by memory-storing instructions to sets of constraints on writing to those locations. Based on these constraints and the privileged and unprivileged compartments as described in Tasks I and II, we will construct quantitative security metrics. The input constraints are a refinement of target sets. As a result, our metrics will be more precise and better capture the security properties of the system under test.

Since introducing security mechanisms invariably involve trade-offs, we will also measure performance (both raw performance as well as any performance variation due to the mechanisms); resource utilization such as memory, flash-storage (e.g., for code size increases), or power; and reliance on hardware capabilities (such as the number of MPU registers required).

Undoubtedly, the impact on security, performance, and resource utilization of some defense mechanisms will be "tunable." For example, using more MPU registers to increase the number of compartments will likely lead to greater security at the expense of runtime and resource use. An important question to answer is how does the mechanism scale? For example, is there a break-down point where small increases in security come with large performance penalties? Similarly, how portable is the mechanism? Does it rely on specialized hardware not present on other embedded systems? Answers to these questions are essential for evaluating defense mechanisms. We can study this tradeoff by varying the resource description in the emulator, described in Task I.2.

5.2 IoT Benchmarks

Workloads for IoT or other embedded devices look very different from workloads for desktop, server, and mobile applications. As a result, existing benchmarks for these domains do not adequately measure IoT systems. For example, the well-known SPEC CPU suite of benchmarks are focused on "measuring and comparing compute intensive performances" [26]. In particular, SPEC CPU is concerned with the performance of a single task consisting of integer or floating point computations. An IoT device, by contrast, may spend most of its time in a low-power mode waiting for an event such as a timer firing or receiving input from a sensor or network. Once the event occurs, the device switches into a higher-power mode and executes a short task, often involving interaction with the physical world by means of attached peripherals, and then returning to the low-power mode.

Requirements. Benchmarks for IoT devices must meet several criteria. First, the applications must be realistic and mimic the application characteristics discussed above. While an 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 code complexity, 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. Similarly, the network characteristics tend to be quite variable and can affect the timing measurements. The IoT devices 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 (local resources like memory and stable storage, and energy resources), 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.

5.3 BenchIoT: Our Contribution

We have developed BenchIoT, a benchmark suite and evaluation framework that fulfills all the above criteria for evaluating IoT devices [1]. Our benchmark suite comprises 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 benchmarks enable deterministic execution of external events and utilize network send and receive. BenchIoT targets 32-bit IoT devices implemented using the popular ARMv7-M architecture. Each BenchIoT benchmark is developed in C/C + + and compiles both for bare-metal IoT devices (i.e. without an OS), and for the ARM Mbed Operating System (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 devices such an embedded 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 us to precisely control when and how the event is delivered to the rest of the software. This approach has been used in the past in embedded systems for achieving repeatability as a means to automated debugging [27, 28]. We can also control for the exact content of the events, which again enables the goal of repeatability of external events such as sensors and actuators without relying on the physical environment.

BenchIoT's evaluation framework enables automatic collection of 14 metrics covering four categories: (1) Security; (2) Performance; (3) Resource usage, and (4) Energy consumption Fig. 5. 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 Fig. 3. 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, timing of the interrupts, sensor values to use) 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.

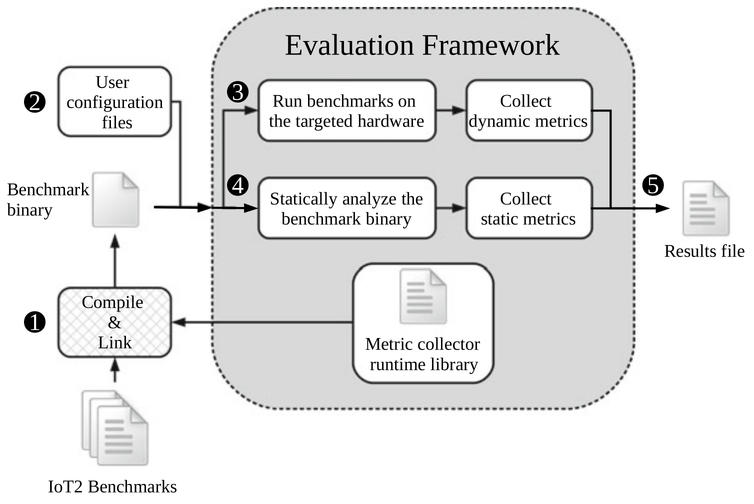

Fig. 3 An overview of the evaluation workflow in BenchIoT. BenchIoT provides five realistic IoT benchmarks spanning one or more of the key functionalities of sense, process, and actuate. BenchIoT measures four types of metrics: security, performance, resource usage, and energy consumption

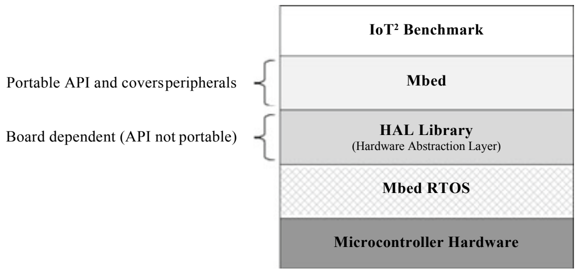
IoT² Benchmark
Mbed
HAL Library
(Hardware Abstraction Layer)
Mbed RTOS
Microcontroller Hardware
Portable API and coversperipherals
Board dependent (API not portable)

Fig. 4 Illustration of software layers and APIs used in developing BenchIoT benchmarks. BenchIoT provides portable benchmarks by relying on the Mbed platform

To implement the benchmarks and demonstrate rich and complex IoT devices applications, BenchIoT targets 32-bit IoT devices using the ARM Cortex-M (3, 4, 7) μCs, which are based on the ARMv7-M architecture. ARM Cortex-M is the most popular μC for 32-bit μCs with over 70% market share. This enables the benchmarks to be directly applicable to many IoT devices being built today. As shown in Fig. 4, 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 μCs, 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.

Benchmark Applications

Table 1 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/TP stack operates in polling mode and uses a different code base. As a result the runtime of bare-metal and OS benchmarks are different.

6 Conclusion

It is upon us to significantly and promptly improve the security for bare-metal embedded and IoT systems. This has become imperative as they form the fabric, sometime hidden, of many critical systems, ranging from industrial control systems, public use equipment (like elevators and escalators), autonomous transportation facilities, personal IoT devices (smart devices and home assistants), to the innards of high-end computing equipment (like disk drives) or mobile equipment (like baseband processors on mobile phones). A high-level direction that we and other members of the community are pursuing is to restrict the privileges and capabilities of different

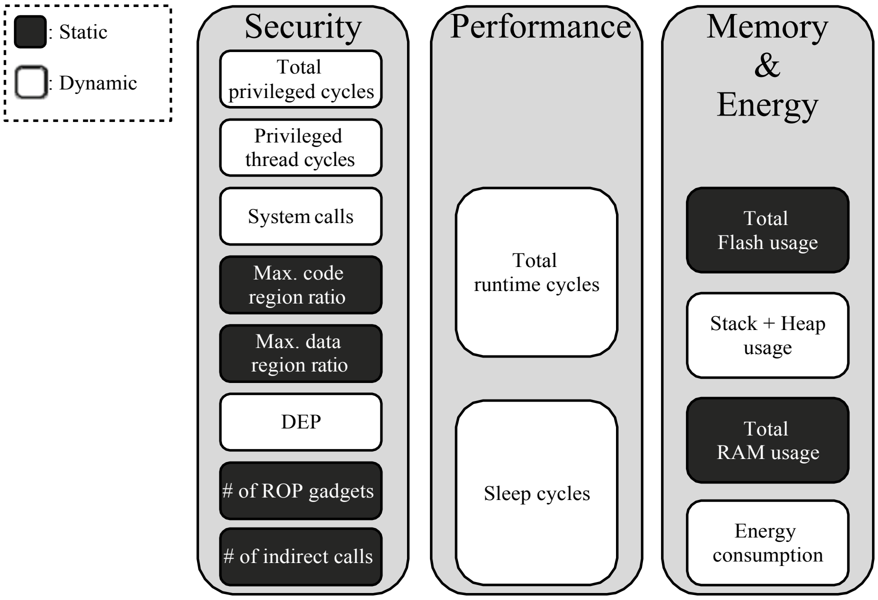

Fig. 5 A summary of the BenchIoT metrics. A white box indicates a dynamic metric, and a black box indicate a static metric

Table 1 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
Lock		✓	✓	Serial (UART/USART), display, μSD card, real-time clock
Firmware updater		✓	✓	Flash in-application programming
Connected display		✓	✓	Display, μSD card

regions of the application to the lowest necessary to perform intended operations. This is ideally done without needing application modification and with limited user annotations, to indicate what denotes security-critical operations, thus easing the application of the solution to legacy embedded applications. In this article, we have identified three interactive thrusts to achieve this solution:

(i) New static and dynamic analyses to identify security and functionality characteristics of each part of the application; **(ii) New runtime techniques** that enforce the desired security properties while minimizing the performance impact; and **(iii) New security metrics and benchmarks** that accurately measure the security and performance impacts of defense mechanisms for embedded systems.

References

1. Almakhdhub NS, Clements AA, Payer M, Saurabh Bagchi B (2019) A security benchmark for the internet of things. In: 2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, 234–246
2. ARM (2017) mbed IoT platform. https://www.mbed.com/en/platform/
3. ARM Inc. Mbed uVisor, 2017
4. Banerjee SS, Jha S, Cyriac J, Kalbarczyk ZT, Iyer RK (2018) Hands off the wheel in autonomous vehicles?: A systems perspective on over a million miles of field data. In: 2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, pp 586–597
5. Barr Michael, Massa A (2006) Programming embedded systems: with C and GNU development tools. O'Reilly Media, Inc.
6. Biswas P, Di Federico A, Carr SA, Rajasekaran P, Volckaert S, Na Y, Franz M, Payer M (2017) Venerable Variadic Vulnerabilities Vanquished. In: SEC: USENIX security symposium
7. Burow N, Carr SA, Nash J, Larsen P, Franz M, Brunthaler S, Payer M (2017) Control-flow integrity: precision, security, and performance. ACM Comput Surv 50(1)
8. Cao PM, Wu Y, Banerjee SS, Azoff J, Withers A, Kalbarczyk ZT, Iyer RK (2019) CAUDIT : continuous auditing of SSH servers to mitigate brute-force attacks. In: 16th USENIX symposium on networked systems design and implementation (NSDI 19), pp 667–682
9. Carlini N, Barresi A, Payer M, Wagner D, Gross TR (2015) Control-flow bending: on the effectiveness of control-flow integrity. In: SEC: USENIX security symposium
10. Chung K, Kalbarczyk ZT, Iyer RK (2019) Availability attacks on computing systems through alteration of environmental control: smart malware approach. In: Proceedings of the 10th ACM/IEEE international conference on cyber-physical systems, pp 1–12
11. Chung K, Li X, Tang P, Zhu Z, Kalbarczyk ZT, Iyer RK, Kesavadas T (2019) Smart malware that uses leaked control data of robotic applications: the case of raven-ii surgical robots. In: 22nd International symposium on research in attacks, intrusions and defenses (RAID 2019), pp 337–351
12. Clements AA, Almakhdub NS, Saab K, Srivastava P, Koo J, Bagchi S, Payer M (2017) Protecting bare-metal embedded systems with privilege overlays. In: IEEE symposium on security and privacy (Oakland), pp 289–303
13. Esiner E, Mashima D, Chen B, Kalbarczyk Z, Nicol D (2019) F-pro: a fast and flexible provenance-aware message authentication scheme for smart grid. In: 2019 IEEE international conference on communications, control, and computing technologies for smart grids (SmartGridComm). IEEE, pp 1–7
14. Ge X, Talele N, Payer M, Jaeger T (2016) Fine-grained control-flow integrity for kernel software. In: EuroSP: IEEE European symposium on security and privacy
15. Haller I, Jeon Y, Peng H, Payer M, Bos H, Giuffrida C, van der Kouwe E (2016) Type sanitizer: practical type confusion detection. In: CCS: ACM conference on computer and communication security
16. Jeon Y, Biswas P, Carr SA, Lee B, Payer M (2017) HexType: efficient detection of type confusion errors for C++. In: CCS: ACM conference on computer and communication security
17. Jha S, Cui S, Banerjee S, Cyriac J, Tsai T, Kalbarczyk Z, Iyer RK (2020) Ml-driven malware that targets av safety. In: 2020 50th annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE, pp 113–124

18. Kuzentsov V, Payer M, Szekeres L, Candea G, Song D, Sekar R, Code pointer integrity. In: OSDI: symp. on operating systems design and implementation, vol 214
19. Lou X, Tran C, Tan R, Yau DKY, Kalbarczyk ZT (2019) Assessing and mitigating impact of time delay attack: a case study for power grid frequency control. In: Proceedings of the 10th ACM/IEEE international conference on cyber-physical systems, pp 207–216
20. Midi D, Payer M, Bertino E (2017) Memory safety for embedded devices with nes check. In: AsiaCCS: ACM symposium on information, computer and communications security
21. Necula GC, Condit J, Harren M, McPeak S, Weimer W (2005) CCured: type-safe retrofitting of legacy code. Trans. Prog. Lang. Syst. 27(3):477–526
22. Payer M, Barresi A, Gross TR (2015) Fine-grained control-flow integrity through binary hardening. In: DIMVA: conference on detection of intrusions and malware and vulnerability assessment
23. Rudd R, Skowyra R, Bigelow D, Dedhia V, Hobson T, Crane S, Liebchen C, Larsen P, Davi L, Franz M, Sadeghi A-R, Okhravi H (2017) Address-oblivious code reuse: on the effectiveness of leakage-resilient diversity. In: Proc Netw Distribut Syst Secur Symp (NDSS 17), pp 1–15, 2017
24. Salwan J (2011) ROPgadget—Gadgets finder and auto-roper
25. Schuster F, Tendyck T, Liebchen C, Davi L, Sadeghi A-R, Holz T (2015) Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in C++ applications. In: Security and privacy (SP), 2015 IEEE symposium on. IEEE, pp 745–762
26. Standard Performance Evaluation Corporation (2017) SPEC CPU®2017. Online: https://www.spec.org/cpu2017/
27. Tancreti M, Hossain MS, Bagchi S, Raghunathan V (2011) Aveksha: a hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In: Proceedings of the 9th ACM conference on embedded networked sensor systems, pp 288–301
28. Tancreti M, Sundaram V, Bagchi S, Eugster P (2015) 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 (IPSN). ACM, pp 286–297
29. Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson Ú, Lozano L, Pike G (2014) Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Fu K (ed) Proceedings of USENIX Security 2014. USENIX
30. UBM Canon (2015) 2015 embedded markets study
31. Zhang M, Sekar R (2013) Control flow integrity for COTS binaries. In: Proceedings of USENIX security 2013. USENIX
32. Zhu DY, Jung J, Song D, Kohno T, Wetherall D (2011) Tainteraser: protecting sensitive data leaks using application-level taint tracking. ACM SIGOPS Oper Syst Rev 45(1):142–154