BenchIoT: A Security Benchmark for the Internet of Things

Naif Saleh Almakhdhub*§, Abraham A. Clements¶, Mathias Payer‖, Saurabh Bagchi*
*Purdue University, {nalmakhd, sbagchi}@purdue.edu
¶Purdue University and Sandia National Laboratories, aacleme@sandia.gov
‖EPFL, mathias.payer@nebelwelt.net
§King Saud University

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

Evaluating security mechanisms for IoT-μCs 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-μCs is limited and ad-hoc. A sound benchmarking suite is essential to enable robust and comparable evaluations of security techniques on IoT-μCs.

This paper introduces BenchIoT, a benchmark suite and evaluation framework to address pressing challenges and limitations for evaluating IoT-μCs 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-μCs 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].

I. 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 [1]. 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 (μCs), 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 paper as IoT-μCs.

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-μCs. For example, SPEC CPU2006 [2] 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 [3]. Several benchmarks [4]–[7] are designed specifically for comparing performance on μCs. 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-μCs.

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

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

TABLE I
A SUMMARY OF DEFENSES FOR IOT-μCS WITH THE EVALUATION TYPE.

Defenses	Evaluation Type	
	Benchmark	Case Study
TyTan [8]		✓
TrustLite [9]		✓
C-FLAT [10]		✓
nesCheck [11]		✓
SCFP [12]	Dhrystone [7]	✓
LiteHAX [13]	CoreMark [6]	✓
CFI CaRE [14]	Dhrystone [7]	✓
ACES [15]		✓
Minion [16]		✓
EPOXY [17]	BEEBS [4]	✓

and renders comparison between different defenses infeasible as each defense is evaluated according to different benchmarks and metrics. Table I compares the evaluations of several recent security mechanisms for IoT-μCs, 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-μCs 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 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-μCs 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.

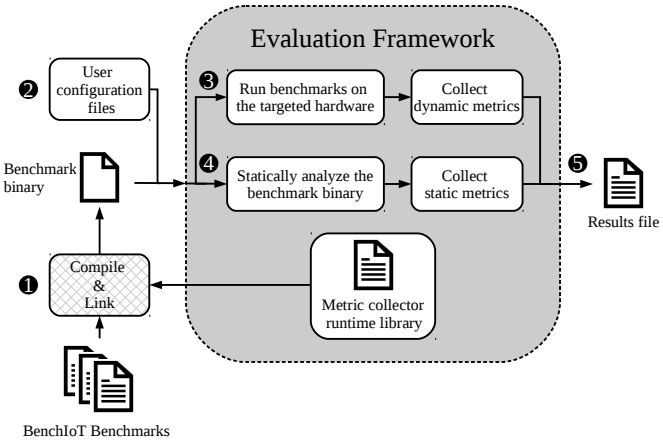

Fig. 1. An overview of the evaluation workflow in BenchIoT.

Our Contribution: BenchIoT

This paper introduces the BenchIoT benchmark suite and evaluation framework that fulfills all the above criteria for evaluating IoT-μCs. 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 benchmarks enable deterministic execution of external events and utilize network send and receive. BenchIoT targets 32-bit IoT-μCs implemented using the popular ARMv7-M architecture. Each BenchIoT benchmark is developed in `C/C++` and compiles both for bare-metal IoT-μCs, 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-μCs 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 [18], [19].

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 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

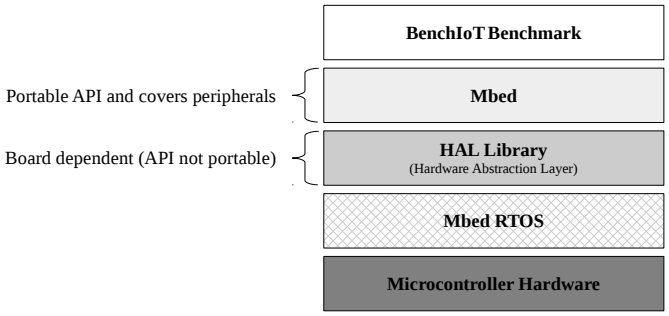

Fig. 2. Illustration of software layers used in developing BenchIoT benchmarks. BenchIoT provides portable benchmarks by relying on the Mbed platform.

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 paper makes the following contributions: (1) This is the first realistic benchmark suite for security and performance evaluation of IoT-μCs. It enables the evaluation of IoT-μCs 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-μCs depending on their effect on sleep cycles. The benchmark suite along with the evaluation scripts is open sourced and available to the research community [20].

II. Scoping and Background

A. Scoping and Target Systems

The goal of this work is to enable security evaluation and comparison for different security defenses on IoT-μCs 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-μCs.

We define an IoT-μC 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. μCs 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 light-weight 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 μCs typically lack security hardware commonly available on server-class systems (e.g., MMUs). However, they commonly have a Memory Protection Unit (MPU) [21]. 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-μCs (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-μCs applications, BenchIoT targets 32-bit IoT-μCs using the ARM Cortex-M(3,4,7) μCs, which are based on the ARMv7-M architecture [22]. ARM Cortex-M is the most popular μC for 32-bit μCs with over 70% market share [23], [24]. This enables the benchmarks to be directly applicable to many IoT devices being built today. As shown in Figure 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 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.

B. Background

Cortex Microcontroller Software Interface Standard: The Cortex Microcontroller Software Interface Standard [25] (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

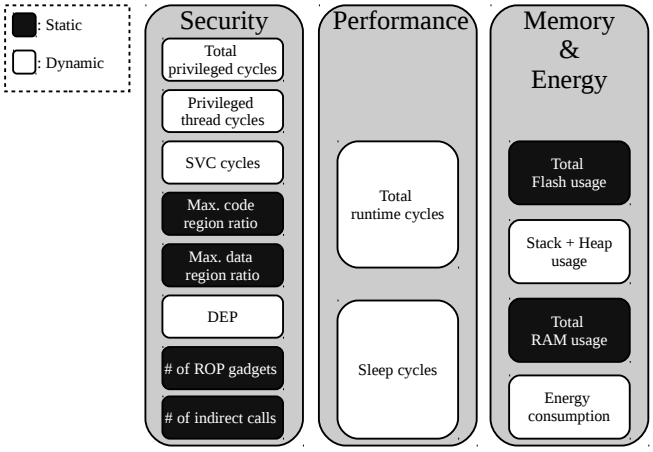

Fig. 3. A summary of the BenchIoT metrics.

(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 [22] 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.

III. Benchmark Metrics

The goal of the BenchIoT metrics is to enable quantifiable evaluation of the security and practicality of proposed defenses for IoT-μCs. 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-μCs (e.g., principle of least privilege); (2) enable evaluating performance effects of defenses on IoT-μCs.

BenchIoT provides 14 metrics spanning four categories, namely: (1) Security; (2) Performance; (3) Memory; (4) Energy. Figure 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.

A. Security Metrics

Total privileged cycles: An important challenge in securing IoT-μCs 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-μCs 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.

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 [26], [27]. 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-μCs [16], [17]. As a result, DEP has been added to the BenchIoT evaluation framework to raise awareness of such an important defense for IoT-μCs 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 $\texttt{W} \oplus \texttt{X}$).

Number of available ROP gadgets: Return Oriented Programming (ROP) [28] 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 [29] 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) [30] is a desired security property. We measure the number of indirect calls by parsing the benchmark binary.

B. 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-μCs 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 [1], [31] use a sleep manager that automates entering sleep mode to minimize energy consumption of IoT-μCs. 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 [32], 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.

C. 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.

IV. Benchmark Design

To develop a realistic benchmarking suite for IoT-μCs, 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.

A. Deterministic Execution Of External Events

Including external events (e.g., user pushing a button) is necessary to represent realistic IoT-μCs 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.

B. Application Characteristics

To increase the coverage of IoT-μCs 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 [33]. 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 [34].

C. Peripherals

The BenchIoT benchmarks are designed to include various peripherals to represent realistic interactions of IoT-μCs. In addition, peripherals can increase the attack surface for an application [35], [36], and thus their security evaluations

differ. For example, attacks against autonomous vehicles target vulnerabilities in the Controller Area Network (CAN) bus [37]. 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.

D. Portability

BenchIoT aims to enable security evaluation for IoT-μCs across a wide set of hardware platforms. Since IoT-μCs 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 [1], [38]. Unlike other RTOSs [39], 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 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.

E. 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.

V. Benchmark Applications

In this section we describe the BenchIoT benchmarks and highlight the notable features of IoT applications that each demonstrates. Table II 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 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 [40].

TABLE II
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

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 [41], [42]). 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 [43].

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 [44].

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.

VI. Evaluation Framework

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

Automating the evaluation of IoT-μCs is important since evaluating IoT-μCs 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.

A. Static Metrics Measurements

We collect the static metrics explained in Figure 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 [29]. 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.

B. 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, 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 rewriter [45]. 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 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.

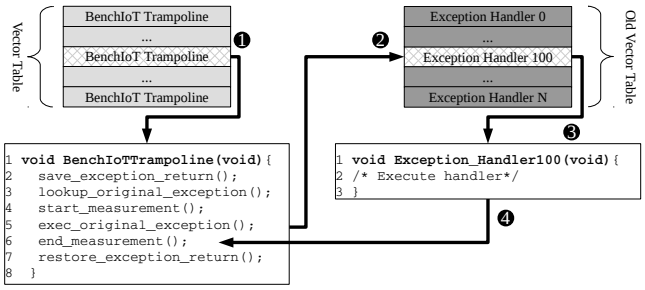

Fig. 4. Exception handlers tracking with BenchIoT.

VII. Evaluation

To demonstrate how the metrics described in Section III 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.

A. Defense Mechanisms

The first defense is ARM's Mbed-μVisor [46]. 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 [10], [47]–[49], 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 [16], [46], [50], [51] 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-μCs 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 [52] 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.

B. Performance and Resource Usage Evaluation

Figure 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 [53] 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 Section IV-C.

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 5 and all other dynamic results, the evaluation framework used the metric collector runtime library. As shown in Figure 5(c), the metric collector library has a low average overhead of 1.2%.

Figure 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

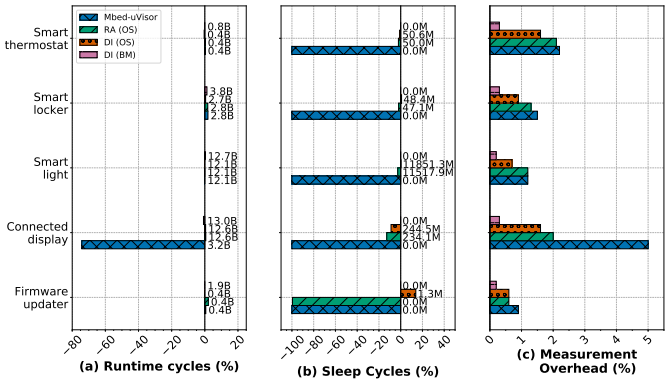

Fig. 5. Summary of BenchIoT performance metrics for μVisor, Remote Attestation, (RA) and Data Integrity (DI) defense mechanisms overhead as a % of the baseline insecure applications. BM: Bare-Metal.

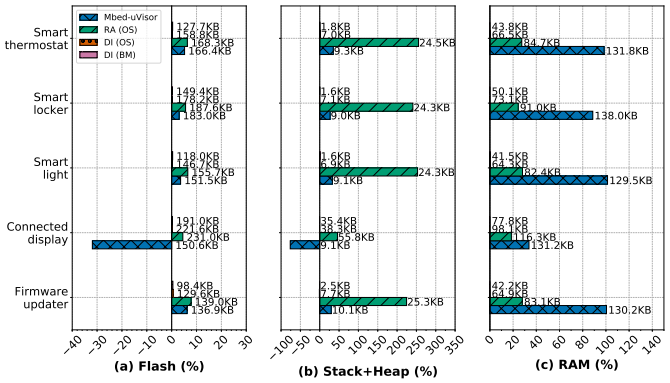

Fig. 6. Summary of BenchIoT memory utilization metrics for μVisor, Remote Attestation (RA), and Data Integrity (DI) defense mechanisms overhead as a % over the baseline applications. The size in KB is shown above each bar.

TABLE III
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%)	✗

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 V, the bare-metal benchmarks differ in their implementation although they provide the same functionality. These differences are also manifested in the Flash metrics.

C. Security Evaluation

Minimizing privileged execution: Minimizing privileged execution is a desired security property (Section III-A). However, as shown in Figure 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 7(a)). The μVisor, however, shows the highest 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 III. 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 III, 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

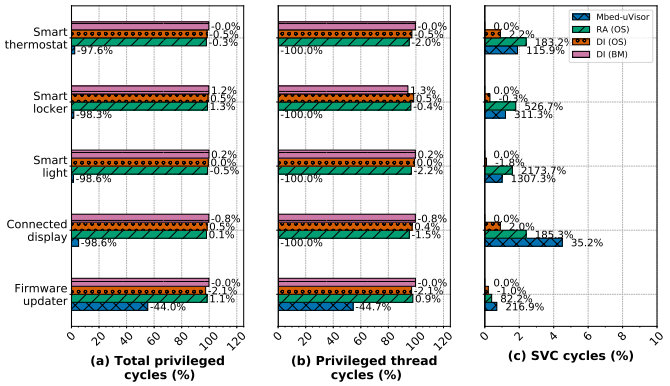

Fig. 7. Summary of BenchIoT comparison of minimizing privileged execution cycles for Mbed-μVisor, Remote Attestation (RA) and Data Integrity (DI) defense mechanisms as a % w.r.t the total runtime execution cycles. The overhead as a % of the baseline insecure applications is shown above each bar. BM: Bare-Metal

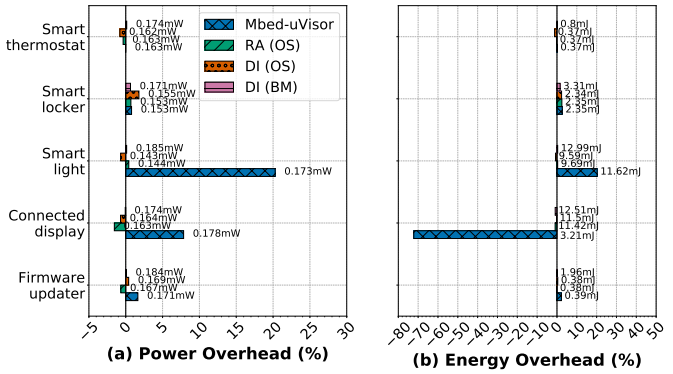

Fig. 8. Summary of power and energy consumption with the BenchIoT benchmarks for the defense mechanisms as a % overhead of the baseline insecure applications. Power and energy values are shown above each bar in mW and mJ, respectively. BM: Bare-metal

established mechanisms such as CFI. Moreover, an important 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*. [15]. 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.

D. Energy Evaluation

Now we look at the energy implication of the benchmarks (Figure 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 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 5 for the bare-metal benchmarks. Thus, difference in energy consumption is caused by the increase in total runtime due to the defense mechanism.

E. Code Complexity Comparison to BEEBS

To measure the complexity of BenchIoT benchmarks, we measure the cyclomatic complexity [54] and compare to the BEEBS [4] benchmarks. BEEBS has been used for security evaluation in embedded systems by EPOXY [17] and for energy evaluation by many prior works [55]–[57]. We exclude HAL libraries from the measurements for both benchmark suites as they differ based on the vendor and the hardware as discussed in Section IV-D. This provides a consistent comparison without the influence of the underlying hardware.

TABLE IV
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

Table IV 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 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.

VIII. RELATED WORK

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

Desktop benchmarks: Soteria [60] is a static analysis system targeting IoT platforms (e.g., Samsung's SmartThings

TABLE V
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 Connectivity	Peripherals
	Sense	Process	Actuate		
BEEBS [4]		✓			
Dhrystone [7]		✓			
CoreMark [6]		✓			
IoTMark [58]	✓	✓		partially (bluetooth only)	only I^2C
SecureMark [59]		✓			
BenchIoT	✓	✓	✓	✓	✓

market), which are assumed to be connected to the cloud. IoTAbench [61] and RIoTBench [62] are benchmarks for large-scale *data analysis* of IoT applications. BenchIoT however targets IoT-μCs.

High-end embedded systems benchmarks: Mibench [5] 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 [63] is an extension of Mibench targeting multi-core embedded processors. Other benchmarks target specific applications of embedded systems. MediaBench [64] targets multimedia applications. DSP-stone [65] evaluates compilers for Digital Signal Processing (DSP) applications for embedded systems. BenchIoT benchmarks differ from the above since we focus on IoT benchmarks, enabling security evaluations of IoT-μCs, and incorporating networking.

μCs Benchmarks: The Worst-Case Execution Time (WCET) [66] evaluates the worst execution time for real-time systems. Dhrystone [7] is a synthetic benchmark to evaluate integer performance. BEEBS [4] is a collection of benchmarks from Mibench, WCET, DSP-stone, and the Livermore Fortran kernels [67] to evaluate energy consumption for bare-metal systems. CoreMark [6] 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-μCs benchmarks: IoTMark [58] evaluates the energy overhead of wireless protocols such as Bluetooth. SecureMark [59] 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.

IX. 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 [20].

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.

X. 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-μCs, 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-μCs 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 [20].

Acknowledgements

We thank our shepherd Mohamed Kaaniche and the anonymous reviewers for their insightful comments. This work was supported by NSF CNS-1801601, NSF CNS-1718637, Northrop Grumman Cybersecurity Research Consortium, and in part by Sandia National Laboratories. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. SAND2019-4009 C.

REFERENCES

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

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

[3] ——, "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

[4] 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

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting bare-metal embedded systems with privilege overlays," in *Security and Privacy Symp.* IEEE, 2017.

[18] 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.

[19] 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)*.

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

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

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

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

[39] 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.

[40] 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.

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

[42] "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.

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

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

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.
[51] FreeRTOS-MPU, "FreeRTOS-MPU," https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.
[52] "STM32479I-EVAL," http://www.st.com/resource/en/user_manual/dm00219329.pdf.
[53] Mbed-uVisor, "mbed OS can't sleep when uVisor is enabled," 2017, https://github.com/ARMmbed/uvisor/issues/420, 201.
[54] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.
[55] 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.
[56] 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.
[57] J. Constantin, L. Wang, G. Karakonstantis, A. Chattopadhyay, and A. Burg, "Exploiting dynamic timing margins in microprocessors for frequency-over-scaling with instruction-based clock adjustment," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*. IEEE, 2015, pp. 381–386.
[58] EEMBC, "Iotmark - an eembc benchmark," https://www.eembc.org/iot-connect/about.php.
[59] ——, "Securemark - an eembc benchmark," https://www.eembc.org/securemark/.
[60] 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.
[61] 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.
[62] 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.
[63] 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.
[64] 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.
[65] V. Zivojnovic, "Dsp-stone: A dsp-oriented benchmarking methodology," *Proc. of ICSPAT'94*, 1994.
[66] 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.
[67] F. H. McMahon, "The livermore fortran kernels: A computer test of the numerical performance range," Lawrence Livermore National Lab., CA (USA), Tech. Rep., 1986.