SAFES: Sand-boxed Architecture for Frequent Environment Self-measurement

Toshiki Kobayashi
NEC Corporation
t-kobayashi@ib.jp.nec.com

Takayuki Sasaki
NEC Corporation
t-sasaki@fb.jp.nec.com

Astha Jada
NEC Corporation
a-jada@aj.jp.nec.com

Daniele E. Asoni
ETH Zürich
daniele.asoni@inf.ethz.ch

Adrian Perrig
ETH Zürich
adrian.perrig@inf.ethz.ch

ABSTRACT

Monitoring software of low-end devices is a key part of defense in depth for IoT systems. These devices are particularly susceptible to memory corruption vulnerabilities because the limited computational resources restrict the types of countermeasures that can be implemented. Run-time monitoring therefore is fundamental for the security of these devices. We propose a monitoring architecture for untrusted software at the I/O event granularity for TrustZone-enabled devices. The architecture enables us to measure the integrity of the code immediately before its execution is triggered by any input. To verify the integrity in a lightweight manner, we statically determine the minimal code region that needs to be measured based on the I/O operation. We develop a prototype of the architecture using TrustZone-M and demonstrate that our prototype has a low processing overhead and small ROM memory footprint.

KEYWORDS

ARM TrustZone, Trusted computing

ACM Reference Format:
Toshiki Kobayashi, Takayuki Sasaki, Astha Jada, Daniele E. Asoni, and Adrian Perrig. 2018. SAFES: Sand-boxed Architecture for Frequent Environment Self-measurement. In *3rd Workshop on System Software for Trusted Execution (SysTEX '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3268935.3268939

1 INTRODUCTION

Cyberattacks are spreading widely to various embedded systems such as industrial systems, critical infrastructure, and biomedical instrumentation. Many security products and technologies developed for traditional IT systems are not well suited to provide end-point protection for low-end embedded devices with a microcontroller unit (MCU) [19]. The most common and simple class of attacks against such devices exploits a memory bug to overwrite software code in memory [18]. The usual way to mitigate such code corruption attacks is by placing the executable code in read-only memory regions, which are set up and managed by the OS with the support of the hardware. However, some low-end embedded devices cannot support such code integrity enforcement due to the limited resources available given real-time requirements. Furthermore, even when such memory protection is supported, it can be reconfigured when the processor is running in privileged mode (also known as *kernel* or *handler* mode), meaning that a memory bug may be exploited to disable the read-only memory protection.

To ensure code integrity on low-end devices, several attestation schemes have been proposed [17]. Remote attestation enables a device owner to check the code integrity of a remote device, and it is typically based on hardware-based security technologies such as Trusted Platform Module (TPM) [8] or Intel's Software Guard Extensions (SGX) [10]. However, the biggest issue of remote attestation is the gap between the Time-of-Check and Time-of-Use (TOCTOU) [2]. Remote attestation can guarantee code integrity at a particular point in time, but it cannot ensure runtime integrity after that point, nor can it guarantee that no compromise happened in the past. Some attestation schemes have been proposed which attempt to solve this problem. ICE [15] enforces code execution after code checking by disabling interrupts. However, such solutions are not suitable for embedded devices for which interrupts are critical. ERASMUS [3] proposed periodic self measurement, an effective approach to reduce the TOCTOU gap: in this paper we build on this idea, with the aim of improving the security guarantees.

1.1 Goals and contributions

This paper proposes SAFES (Sand-boxed Architecture for Frequent Environment Self-measurement), an architecture for efficient and secure self-measurement of the code running on an embedded device. SAFES enables us to measure the integrity of the code in the sand-boxed environment at each I/O event. The measurement at input time ensures the integrity of the code and prevents tampered code from generating output from the device.

We also introduce small-area code measurements to reduce the overhead. Our scheme utilizes I/O data such as user commands to identify the smallest code region that should be measured. We statically analyze a program image to obtain a code structure at compilation time, which forms the base for the runtime attestation. We note that we limit our scope to MCU programs that have a static memory layout and do not support dynamic code loading.

The main contributions of this paper are the following.

- We propose novel scheme for runtime code verification which ensures the execution of correct code on the device

SysTEX '18, October 15, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5998-6/18/10...$15.00
https://doi.org/10.1145/3268935.3268939

and that no output is generated when the code is changed. In contrast to related work, our scheme reduces TOCTOU risk by performing self measurements at I/O events. At the same time, the scheme achieves low overhead by measuring only the code regions executed for the specific I/O event.
- We implement a prototype of SAFES on ARM Versatile Express Cortex-M Prototyping System (V2M-MPS2+).
- We evaluate our prototype, demonstrating low execution overhead and small program binary size.

2 PROBLEM STATEMENT

2.1 Code tampering

In industrial systems, the embedded devices constitute the bridge between the computational and the physical world by the use of sensors and actuators. Thus malicious attacks can cause physical damage to products and facilities, and even harm to humans. Code tampering is the most straightforward way to execute arbitrary code on a vulnerable device. The adversary overwrites a part of the code on the device's memory by leveraging vulnerabilities. For example, the adversary could make an invalid pointer go out of bounds or become dangling, and then use it to overwrite memory regions containing executable code. Even though code integrity is enforced by the OS on modern systems, many embedded devices do not support it due to limited computational resources and real-time constraints. Additionally, embedded devices should be working in real-time. Some additional scanning on the devices for ensuring code integrity introduces overhead and may break its real-time guarantees. These problems make it difficult to ensure code integrity.

To protect the devices from physical attacks using code tampering, we aim to satisfy the following requirements:

- Detect code tampering shortly after it happened;
- Prevent tampered code from generating output;
- Keep the overhead of the countermeasures low to satisfy real-time execution constraints.

2.2 Trust model (Adversarial model)

We consider an adversary performing a runtime attack against a target device to violate code integrity by exploiting memory errors. We assume the device has a trusted environment that has no vulnerability and cannot be compromised, and an untrusted environment that may be susceptible to memory errors caused by user input. Thus the adversary can overwrite a memory region of the untrusted environment via a user input port. We make the following assumptions on the adversarial capabilities:

- Only attacks via a serial or a network port for user input are possible against the target device. In particular, physical attacks and side channel attacks are ruled out.
- Trusted code (i.e., code running in the secure world) cannot be compromised and has no vulnerability.
- The adversary can change the code everywhere in the untrusted environment. However, we assume that for a single input (and therefore, for a single attack instance, e.g., a single buffer overflow) the adversary cannot both modify the code and then change it back to the original.

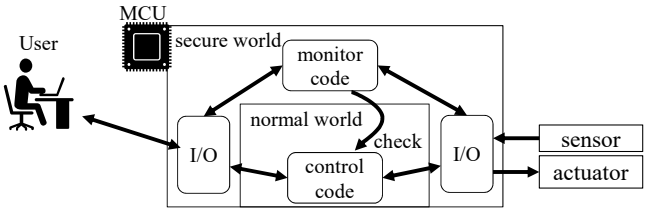

Figure 1: System model of a sandboxed architecture with code measurement. The monitor code checks the control code per I/O event.

In this work, we do not consider control-flow hijacking without any code tampering such as Return-Oriented Programming [16] and data-only attacks such as non-control data attacks [4] and Data-Oriented Programming [9].

3 ARCHITECTURE DESIGN

3.1 ARM TrustZone-M

We consider a code verification method based on TrustZone-M as a trust anchor. Code verification is achieved by measuring code integrity, for instance by checking its hash value. Since the device verifies its code locally, the verification needs to securely compute the hash value of the code, and to store on the device a whitelist of hash values which are considered to be correct. To realize these requirements, we use TrustZone-M as a trusted execution environment on ARMv8-M architecture [13].

TrustZone-M introduces a hardware-supported security extension which allows the creation of two separate worlds, the *secure world* and the *normal world*. The secure world has higher privileges and typically is used to run a small trusted kernel. The normal world instead is typically used to execute a full-fledged, untrusted OS. TrustZone allows software running in the secure world to control and access the data of the normal world, while the normal world is prevented from accessing the secure world. Furthermore, TrustZone's hardware informs the memory controllers and peripherals about whether an access originates from the normal or from the secure world, allowing the enforcement of access control by the secure world.

3.2 Per-I/O measurement

We introduce a per-I/O measurement to check the executing code at every I/O event to reduce the TOCTOU gap. Fig. 1 shows our system model. The device's MCU communicates with a user (or a software controller), a sensor and an actuator. The control code that resides in the normal world controls the actuator and returns certain values to the user based on user commands and sensor readings. The monitor code residing in the secure world measures the code integrity of the control code. Every input/output to/from the system goes through the monitor and triggers an integrity check.

Here we assume that the target device receives commands from the user or a software controller, and executes functions that correspond to the command. These functions are used for changing the internal state and to produce outputs for the user, other devices, or for elements such as displays, actuators and LEDs. This assumption

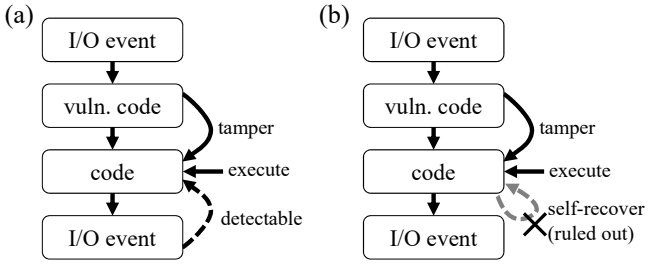

Figure 2: (a) Example of detectable tampering. (b) Example of code recovery, with the altered code changing itself back to the original to avoid detection: we exclude this scenario.

limits the scope of applications. However, it is an assumption that commonly applies to embedded devices, since most of them are configured and controlled by commands, and perform some sensor reading or physical action as a result.

If the control code has a vulnerability causing a memory error, an adversary can potentially tamper with the code everywhere in the normal world. In particular, since the timing of tampering is right after an input event, and therefore after a measurement, there is a TOCTOU gap between tampering and code measurement at the output event that follows the input event. To successfully perform an attack without detection, the adversary would need to perform three actions: tampering, executing and recovering, within the gap. An example of a failure by the adversary is shown in Fig. 2(a): there is a single memory corruption vulnerability between I/O events, so the adversary can modify the code that is executed, but the modification is detectable before the output is generated. In particular, a scenario as shown in Fig. 2(b) is not realistic, since it would require the adversary to both change the executed code, and then modify the code back to the original state, with a single memory error attack.

After the detection of tampering, the monitor code can either safely shut down the system or report the error so further recovery steps can be performed. In this way, the monitor prevents the malicious execution from controlling the output. The desired actions in that case strongly depend on the specific use case. Therefore, it should be considered for each application.

3.3 Small-area code measurement

Although a per-I/O measurement is sufficiently frequent to prevent malicious code from compromising the device's behavior, the execution time overhead will be significant, and possibly intolerable, for time-critical systems, if at every measurement the entire executable needs to be verified. To reduce the overhead of the measurement, we therefore propose a *small area* code measurement that reduces the scan area of each code integrity check based on the specific I/O operation performed. The basic concept is illustrated in Fig. 3(a). When the user sends the command B, the monitor code measures only the part of the control code related to command B. In order to obtain a code structure related to user commands, we perform (prior to deployment) an analysis of the binary of the control code, as shown in Fig. 3(b). In this figure, functions of the code are aggregated based on user commands by using a function call-graph.

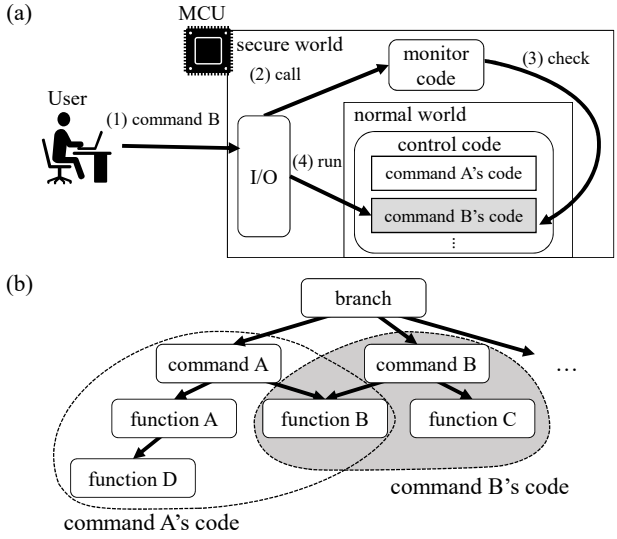

Figure 3: (a) Basic concept of small-area code measurement. (b) Example of code partitioning for small-area code measurement using a function-call graph.

Then the monitor code measures the code integrity for each user command at its input time. The sequence of steps for small-area code measurements is the following: (1) receive the user command from the user, (2) call the monitor code with the identifier of user command, (3) check the part of control code related to the user command, and (4) execute the code if it is correct. Narrowing down the focus of the measurements in this way allows us to reduce the execution time overhead of each measurement. We can further optimize and reduce the code region to be measured by analyzing the control-flow graph instead of the call-graph, trading off additional effort in the offline analysis for faster measurements at runtime.

In the above example, we have focused on a user command, but the same applies to all I/O operations. For example, when some GPIO access is used for reading sensor values, a code measurement at the GPIO access can monitor only a part of code related to the sensor. However, reducing the scan area is a trade-off with security. If the adversary modifies the code at a different location from the scan area, the code measurement will miss the attack. However, whenever the tampered code region is about to be executed because of another input, the attack will be detected.

4 IMPLEMENTATION

We implemented a proof-of-concept prototype of SAFES on the ARM Versatile Express Cortex-M Prototyping System MPS2+ configured as a Cortex-M33 processor executing at 25 MHz.

We designed the entire system including a control code that communicates with a user, with a sensor and an actuator. There are three threads on Keil RTX RTOS with the following tasks, respectively: processing user commands, reading sensor values, controlling an actuator. In order to access I/O from the control code in the normal world, we defined Non-Secure Callable (NSC) functions as interfaces of the UART and GPIO ports for communicating with the user and with the sensor/actuator, respectively. Each interface results in

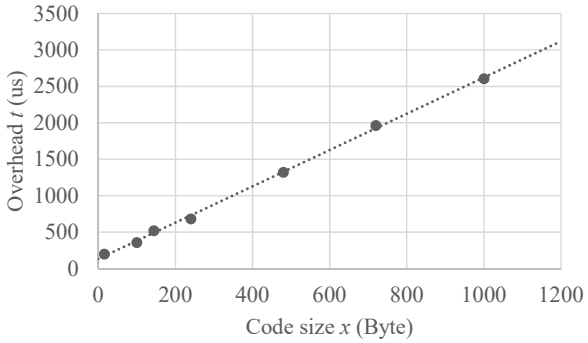

Figure 4: Overhead of the code measurement vs. code size.

a call for the monitor code with an identifier specifying which part should be measured.

For our prototype, we construct a whitelist based on the function-call graph. Fig. 3(b) shows an example of partitions of the whitelist's components. We aggregate the function lists of entry points corresponding to each user command. We then compute the hash values of these functions and register them, together with the information about target addresses, in the whitelist, associated with an identifier. The monitor code obtains the identifier from the caller and the memory addresses from the whitelist, and computes the hash value. We rely on SHA-256 for computing the hash values.

5 EVALUATION

We evaluate our prototype in terms of detection of code tampering and in terms of performance. To detect code tampering, we implement buffer overflow bugs on array indexing, allowing direct access to any point in the memory of the normal world by using negative integer indexing. Using the memory bugs, we verify that the code measurement is working as expected, detecting the attacks.

To evaluate the performance, we measure the overhead of the tamper detection process. The results are shown in Fig. 4. We measure the overhead for different sizes of monitored code regions, with the largest code region being 1 kB. The dashed line is obtained by the best fit with A x+ B, where x represents code size, A = 2.49 us/Byte and B = 132 us. In this evaluation, we disable interrupts while the hash calculation is performed. The size-dependent term of 2.49 us/Byte is mainly caused by the hash calculation. The size independent term of 132 us seems negligible. It includes the context switching between the secure world and the normal world. Owing to TrustZone-M, the context switching overhead is minimal. To check the impact of interrupts from threads in the normal world, we also measured the overhead when we enable interrupts from a thread for reading sensor values. We find that when the measurement is interrupted by the sensor thread once, the measured overhead was increased by about 0.03 ms. Regarding the security of interrupts, we note that threads in the normal world cannot affect the result of code integrity measurement because the measurement is executed in the secure world. On the other hand, an interrupting thread, if malicious, may alter the code being measured and possibly hide the fact that it had been previously altered. However, the code of the malicious thread is either being run as the result of another I/O event, in which case the code of the malicious thread will itself be measured and detected as being altered, or else caused by the same I/O event that invoked the code for the interrupted thread, in which case our assumption about the hardness of self-erasing code injections implies that the compromised code of the malicious, interrupting thread will be detected.

As for our prototype, the overhead of single small-area measurement is mostly less than a few milliseconds since most whitelist contents are smaller than 1 kB. This is sufficiently small for an IoT device since typically a single scan cycle of a programmable logic controller lasts for a few milliseconds on average, while for devices requiring Internet communication, we note that the network delay alone is much larger. If we do not use small-area code measurement, the total code size in the normal world of 15 kB takes 37 ms for every measurement. Although the effectiveness of small-area code measurement depends on the software's structure, most complex software has portions of code which are unused, and whose memory could therefore be ignored by small-area code measurement.

We also measure the code size of the monitor code by compiling our prototype with and without the code measurement. We found the (machine) code size of the monitor to be 2.5 kB. This includes the hash function and the calls to the monitor code in the normal world software. Although the calls are deployed on each I/O interface, their cost in terms of ROM data is only 8 bytes per call. Therefore, the difference in code size is mainly due to the hash function, and is sufficiently small given the MCU's memory.

As additional costs, we also consider the memory overhead of the whitelist and the implementation cost of the I/O interface in the secure world. In our prototype, each entry of the whitelist comprises a set of memory addresses, function addresses and a hash value of the function code. We implement the whitelist as a static array that can store the addresses and lengths of 20 code regions. The size of each entry is 116 bytes. The size of the whitelist depends on the complexity of target software and on the measurement granularity. As for the implementation cost of the I/O interfaces, it is minimal, since they only need to forward calls to the original I/O drivers in the secure world.

6 SECURITY CONSIDERATIONS

In this work, we focus exclusively on code integrity. This means in particular that control-flow hijacking attacks cannot be detected nor prevented with our scheme: changes in the control flow, e.g., if the functions are executed in different sequence due to an overwritten return pointer, cannot be detected by our scheme since the code is not changed by such attacks. Therefore incorporating control flow integrity would be a way to harden the security of embedded devices further, and a promising direction for future work.

We also note that we only focus on self measurement, in which trusted code in the secure world measures untrusted code locally. This may be integrated with remote attestation, with which the state of the device can be periodically checked by a remote verifier [6].

7 RELATED WORK

Prior work on runtime attestation aims to protect the integrity of the system during the execution. C-FLAT [1] is a dynamic attestation scheme that enables the verification of the control flow of software

to prevent runtime attacks that hijack the software's execution. It checks whether the control flow is benign or malicious based on the software's control-flow graph obtained by offline binary analysis. While C-FLAT requires instrumentation of software, LO-FAT [7] uses hardware components (IP blocks) to check the correctness of the control flow without requiring changes to the original software. Such control-flow attestation schemes are neither strictly weaker nor strictly stronger than our code integrity protection mechanism. For example, by using a buffer overflow, the attacker may be able to hijack an existing buffer copy to write malicious code into memory, while still preserving all the jump pointers of the original code. The control-flow attestation mechanisms cannot detect such attacks, while SAFES will detect any modification to the code. On the other hand, with SAFES it is impossible to detect attacks that are purely based on control-flow hijacking, while they are easily detectable by control-flow attestation. We therefore do not consider the two approaches as competing, but rather as complementary mechanisms to harden the security of embedded devices.

There have also been several proposals to mitigate runtime attacks [18]. Stack cookies (also known as canaries) and address space layout randomization [11] are well investigated approaches. However, these solutions are not supported on most embedded OSes due to the limited resources available given real-time requirements. Tock [12] is an OS for resource-constrained MCUs which allows software fault isolation, provides memory protection and manages dynamic memory allocation efficiently. These features harden the security of embedded devices in general. CaRE [14], which is also complementary to our approach, can enforce return integrity for TrustZone-enabled MCUs. CaRE utilizes a shadow call stack [5] which holds a copy of the return addresses in secure world. Although its control-flow protection supports only return addresses, it covers interrupt-driven devices.

7.1 Hardware-supported protection

The security of SAFES is based TrustZone-M's hardware-based features. Initially, ARM TrustZone was designed for high-end Cortex-A processors, but with TrustZone-M the same strong isolation capabilities are made available to MCUs. Currently, only a small subset of Cortex-M processors support TrustZone, but we believe that as ARM and the IoT industry move towards adopting security as a fundamental design aspect for IoT devices [20], a larger fraction of processors will support TrustZone or similar technologies.

ARM also offers simpler security features besides TrustZone, such as the memory protection unit (MPU), which can be configured to, e.g., make the code region read-only during normal execution. However, when the MCU is running in privileged (*handler*) mode, the MPU may be reconfigured by writing to the appropriate memory region used for system control [13]. To achieve security in depth, it may be reasonable to use the MPU along with SAFES, since their scope is somewhat orthogonal: for instance, the MPU can protect normal world memory regions used for system control and peripheral control during unprivileged execution, an attack on which may not be detectable with SAFES.

8 CONCLUSIONS

The security of safety- and time-critical embedded devices faces numerous challenges. Lack of code integrity enforcement is the most critical point for protecting these devices. In this paper we introduce SAFES, a system for monitoring the code integrity of low-end devices at the frequency of I/O events. This system enables us to ensure the integrity of code regions before their execution, and before they are allowed to generate output of their execution results. To minimize the overhead of this scheme, we perform integrity checks on minimal code areas, based on the pre-computed execution paths corresponding to each user command. Our prototype implementation based on TrustZone-M can detect code tampering, and demonstrates good performance. We discuss how SAFES can be integrated with other techniques such as control-flow integrity and remote attestation, showing how our scheme fits into the general security ecosystem of embedded devices.

REFERENCES

[1] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM CCS*.

[2] Sergey Bratus, Nihal D'Cunha, Evan Sparks, and Sean W Smith. 2008. TOCTOU, traps, and trusted computing. In *International Conference on Trusted Computing and Trust in Information Technologies*.

[3] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. 2017. ERASMUS: Efficient Remote Attestation via Self-Measurement for Unattended Settings. *IEEE/ACM Design, Automation, and Test in Europe (DATE)*.

[4] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*.

[5] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *ASIA CCS*.

[6] Karim El Defrawy, Aurelién Francillon, Daniele Perito, and Gene Tsudik. 2012. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS*.

[7] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, and Ahmad-Reza Sadeghi. 2017. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *DAC*.

[8] Trusted Computing Group. 2011. TPM Main Specification Level 2 Version 1.2, Revision 116. https://trustedcomputinggroup.org/resource/tpm-main-specification, Last accessed: 21 August 2018.

[9] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*.

[10] Intel. 2014. Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Last accessed: 21 August 2018.

[11] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy*.

[12] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *ACM SOSP*.

[13] ARM Ltd. 2016. ARMv8-M Architecture Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0553a.b, Last accessed: 21 August 2018.

[14] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *RAID*.

[15] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2006. SCUBA: Secure code update by attestation in sensor networks. In *ACM workshop on Wireless security*.

[16] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*.

[17] Rodrigo Vieira Steiner and Emil Lupu. 2016. Attestation in wireless sensor networks: A survey. *ACM Computing Surveys (CSUR)* 49, 3 (2016), 51.

[18] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*.

[19] John Viega and Hugh Thompson. 2012. The state of embedded-device security (spoiler alert: It's bad). *IEEE Security & Privacy* 10, 5 (2012), 68–70.

[20] Paul Williamson. 2017. It's Here: A Common Industry Framework for Protecting a Trillion Connected Devices. https://www.arm.com/company/news/2017/10/a-common-industry-framework.