2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)

Towards Hardware-Assisted Security for IoT Systems

(Invited)

Yier Jin
Department of Electrical and Computer Engineering
University of Florida
yier.jin@ece.ufl.edu

***Abstract*—As computing devices become more commonplace in every day life, we have seen an increase of possible attacks on commercial devices and critical infrastructure. As a result, both academia and industry have proposed solutions to mitigate or outright eliminate the ever expanding set of viable targets. Initially, this resulted in an influx of software-based defenses against these emerging threats. Unfortunately, it was found that software solutions could be bypassed with more advanced attacks and often resulted in high performance overhead. As such, hardware-assisted security defenses have been developed to provide improved security while keeping performance overhead to manageable levels, especially for IoT devices. In this paper, we will provide a survey of prominent hardware-assisted security defenses. We will enumerate the attacks these defenses aim to protect, as well as their effectiveness. We will also discuss the implications in both performance and system design. A comparison between approaches that target the same set of issues, and possible directions for future research will be presented.**

I. Introduction

With the advent of inexpensive and low-power processors, we have seen an explosive growth in interconnected embedded devices. These devices are categorized under the umbrella term *Internet of Things* (IoT). The rate of penetration of these devices in the market has exhibited accelerated growth, with Gartner predicting over 20 billion devices by 2020 [1]. This popular trend has made IoT devices a viable target for attackers. For example, the Mirai botnet exploits weak credentials in IoT devices to install malicious payloads [2]. However, we have found that security issues in IoT go beyond weak credentials and software vulnerabilities.

As the complexity of emerging devices increases, new attack surfaces are exposed. A wide array of vulnerabilities, such as improper data validation [3], [4], [5], memory corruption [6], [7], [8], and improper permission checks [9] are surfacing as a result. Responding to this, both industry and academia have developed new and novel defenses to protect newly developed devices against these emerging attack vectors.

To help researchers better understand the challenges and the state-of-the-art of IoT security, in this paper, we will summarize popular IoT attack categories, as well as present some of the defenses that have been developed as a result. We also discuss a few security primitives provided by the industry to ameliorate the security burden placed in IoT device manufacturers and their developers. We will describe the functionality of the defense as well as a discussion on the applicability of these defenses to modern devices and their deployment requirements.

The remainder of this paper is structured as follows: Section II introduces popular attack categories which have been used in the wild to compromise devices. Section III presents a series of hardware primitives which offer the means of building secure systems. We follow this with a discussion of defenses in Section IV, then present a discussion of their applicability and deployment issues in Section V. Lastly, we present concluding remarks in Section VI.

II. Attacks

In this section, we will introduce the emerging attacks on IoT devices. Some of these attacks are derived from general purpose computing platforms due to the increasing complexities of IoT systems.

A. Code-Reuse Attacks

Code-reuse attacks are attacks that utilize code already existing in the device to deploy a malicious payload [10]. To deploy this kind of attack, possibly remote attacker corrupts and injects code pointers in a device to alter control-flow in a running application. For example, in most architectures when a function calls another function, the return address for the caller is stored in the stack. An attacker with access to a memory vulnerability that allows for corrupting stack values can change the stored return address. When the function returns, it no longer goes back to the caller, but to an attacker controlled program location.

Snippets of code targeted this way are called *gadgets*. Gadgets end in indirect call or jump instructions. An attacker needs a way to control the value used by the indirect control-flow instructions, otherwise the attack is much harder to pull off. Some popular IoT architectures, such as ARM and MIPS, store the return address to the caller in a register rather than the stack. However, the contents of the register must be spilled to the stack before the called function calls another function. In these architectures even though the return address is stored in a register, non-leaf functions spill this value into the stack, where it can be subject for corruption.

Another common technique for attackers to utilize are use after free allocation errors. In this case, the attacker has the program allocate an object over a previously allocated object. Due to an error in the program, the previously deallocated

978-1-7281-3391-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ISVLSI.2019.00118

object is reused. The program utilizes the contents of the newly allocated object as part of the old one. An attacker can utilize this method to, for example inject virtual pointer tables to perform code reuse attacks.

B. Firmware Modifications

Firmware modification attacks are more invasive than code-reuse attacks. This style of attack involves physically changing the software being run in the device. Firmware modifications can utilize vulnerable update facilities, exposed debug interfaces, fault injections, etc. Through firmware modification an attacker can run arbitrary code on the device without the limitations present in a code reuse attack. No longer does the attacker need to find a gadget catalog to utilize, nor an exploitable memory vulnerability to launch the attack itself.

Cui et al demonstrate the ability and effects of remotely modifying the firmware of an HP printer by exploiting the firmware update procedure [11]. The update process utilizes a specially crafted print job containing the update image. Unfortunately, insufficient checks were done on the update image, allowing anything to be written as device firmware. This allows an attacker capable of sending a print job to the device to replace the firmware with a malicious one, for example allowing to transmit the documents printed to a remote location.

Ronen et al discovered a flaw on the update procedure used in the Phillips Hue smart bulb [12]. The flaw was caused by a vulnerability in the ZigBee stack provided by Atmel Corporation, one of the main vendors behind the technologies used in the bulb. The authors demonstrate how a single device can be compromised and the malware spread to its neighbors by only requiring being physically near the first victim.

C. Brute Forcing

Brute forcing is a relatively uncommon type of attack where the attacker will try random combinations to bypass a security mechanism. This attack is often impractical, in the sense that its theoretical computational runtime is $O(n)$. However, popular IoT devices often ship with a default username and password for remote management purposes. These default credentials for different devices are enumerated and stored in a database. To launch the attack against a device, the attacker continuously tries the stored credentials until a valid login is achieved. At this point, the attacker has local access to the device, often with administrator privileges.

This style of attack is centerpiece in the popular Mirai botnet [2]. The Mirai botnet targets exclusively IoT devices. Infected devices scan the IPv4 address space for possible attack targets. Once a device is found, it is scanned for known administrator interfaces by looking at common ports. If known ports are found, then the target device is bombarded with credentials until valid credentials are found. The attacking device then commands the victim device to download and execute a payload, resulting in a firmware modification attack. The payload has two major functions. First, it integrates the device into the Mirai botnet which makes it capable of receiving remote commands from the botnet operator. Second, it makes the device scan for new targets as means to further expand the botnet.

III. Secure Hardware Primitives

In this section we describe a few commercialized security hardware primitive designs that system-on-chip (SoC) vendors have been including in their devices. Device manufacturers are given the option of utilizing them to build defenses.

A. ARM TrustZone

The ARM Security Extensions, popularly known as TrustZone, is a mechanism that provides hardware-backed isolation of program data and code [13]. This is accomplished by defining a new operation mode, or world, in the CPU: *secure mode*, or *secure world*. The fabric of the AMBA/AXI bus is also extended to allow for bus peripherals to properly respond to requests depending on the mode the CPU is in. SoC vendors licensing the Security Extension have the option of gating certain peripherals to be accessible only when the CPU is executing in secure mode. Accesses to these pre-defined regions of memory result in an access violation and trigger an interrupt.

When a TrustZone-enabled SoC starts up, the CPU begins executing code in secure mode. The CPU can then set up attributes for a few regions of memory, gating them from non-secure mode, or normal world, access. The size and number of regions that can be defined are dependent on the memory controller present in the platform. The software must also create an interrupt vector table for the secure world. Then, the software drops privileges to normal mode and continues to execute. Requests done by the normal world to the secure world are done through the secure monitor call instruction, `smc`.

In effect, platforms that make use of TrustZone can be seen as having two operating systems running in parallel: the normal world and the secure world operating systems. The secure world operating system is often smaller, and provides a limited number of security-related services. This is done to reduce the potential attack surface. In contrast, the normal world operating system is much larger and feature-rich. A popular example that makes use of this technology is the Android operating system. The secure world software manages operations such as storing fingerprints, performing fingerprint verification, and digital rights management (DRM) related tasks.

B. ARM TrustZone-M

The often called ARM TrustZone-M, is the version of the security extension present in the ARMv8-M profile [14]. Although the bus matrix in the SoC is subject to changes similar to those in regular TrustZone, the overall mode of operation is different. The ARMv8-M security extension makes use of a *Secure Attribute Unit (SAU)* and an optional, implementation defined attribute unit (IDAU). The SAU and IDAU define a set of regions in the address space an ARMv8-M based microcontroller, flagging them as *secure*, *non-secure*, and

secure and non-secure callable. As their name imply, the secure region can only be accessed by secure software, the non-secure region can only be executed from by non-secure software, whereas the secure and non-secure callable region can be executed with the CPU in either mode.

When software starts up, it begins executing with the CPU in secure mode. Software is then responsible for setting up the SAU, and if present, the IDAU. The software then sets up the stack pointer for the non-secure program, and uses a *veneer* trampoline to jump to it. This region is flagged as *secure and non-secure callable*. Non-secure software can request services from the secure software by trampolining to it using the veneer region. These trampolines contain a secure gate instruction, `sg`, which changes the CPU mode and allows transitions between states. The `sg` instruction also serves as a basic protection against unintended use of secure software, as it also acts as the entry point to all secure software.

C. Trusted Platform Modules

The Trusted Platform Module (TPM) is a series of specifications which establish the working mechanisms of a standalone integrated circuit which provides cryptographic and secure storage services to applications [15]. TPM-provided services include facilities to generate random numbers, secure generation of cryptogrpahic keys, remote attestation, and the binding and sealing of keys. Software must be made aware of the TPM in order to utilize it.

TPMs expose a series of registers that can be modified by software through calls to TPM routines. This is usually done through a concatenate-hash-store primitive. At first, the software resets the contents of the TPM registers to zero. Then, the software computes a hash which is then sent to the TPM. The TPM concatenates this hash to one of its internal registers, then performs a hash of the concatenation, and stores the results. As more code is loaded and executed, hashes are sent to the TPM, which proceeds to process them in the same fashion. When software requests a key from the TPM, the contents of the registers must be the same as they were when the key was generated. If not, the key can not be unlocked and sent out by the TPM. The idea behind this process is not to release secrets from the TPM if the software is in an unknown execution state.

IV. Defenses

In this section we describe a few defense approaches that have been proposed to defend against attacks in IoT devices. These defense approaches target different style of attacks, and should be viewed orthogonal to each other.

A. Control-Flow Integrity

Control-flow integrity (CFI) is a powerful defense mechanism that aims to prevent code-reuse attacks by enforcing an application's control-flow graph (CFG). Under a CFI policy, deviations from the intended CFG results in an error that can be handled by either the hardware or some supervisor software. The major CFI implementations can be categorized in *heuristic-based* and *instrumentation-based*. Heuristic-based CFI policies examine the behavior of the software to determine whether a CRA is under progress. Instrumentation-based CFI policies add checkpoints or instructions to program code to signal the CPU or some form of supervisor of control-flow transfers that need to be tracked. Both types of policies may require specialized hardware support to gather the necessary runtime information. We now describe a few CFI policies.

1) HAFIX and derivatives: HAFIX [16] and its successors [17], [18] modify an embedded LEON3 SPARC CPU, and Intel Siskiyou Peak core in the case of HAFIX. These approaches add new instructions and a dedicated subsystem to dynamically track a properly instrumented program's control-flow while keeping state metadata in a secure memory location. We enumerate the added instructions for the approach in [18] in Table I. Hereafter we refer to this approach as HAFIX++.

The HAFIX++ model also requires the introduction of a label shadow stack and a label state register. These are directly accessible to the operating system, but not to a running process. That is, non-privileged code can only modify the contents of the label shadow stack and label state register through the CFI instructions, whereas the operating system can use move and load/store instructions to change state in these two components.

TABLE I
BASIC HAFIX++ INSTRUCTION SET. INSTRUCTION ENCODINGS ARE HARDWARE DEPENDENT AND TO BE DECIDED BY THE IMPLEMENTATION. IMMEDIATE FIELDS MUST BE AT LEAST 16 bit WIDE AND MUST MATCH THE SIZE OF THE ENTRIES IN THE LABEL SHADOW STACK.

Mnemonic	Action
`cfibr`	Pushes an immediate value into the label shadow stack. Must be issued by a compiler two instructions before an indrect call. Hardware must enforce this instruction order.
`cfiret`	Pops from the label shadow stack and compares obtained value with an immediate value. Raises interrupt if values are not equal. Must be issued by compiler after a call instructions. Hardware must enforce this instruction is executed after a return.
`cfijmp`	Set the label state register to an immediate value, discarding any previous value. Instruction must be issued prior to an indirect jump instruction. The hardware must enforce this execution order.
`cficall`	Set the label state register to an immediate value, discarding any previous value. Instruction must be issued prior to an indirect call and after a `cfibr`. The hardware must enforce this execution order.
`cfichk`	Compare an immediate value to the value stored in the label state register. Raises an interrupt if values are not equal. Must be issued by compiler at the prologue of every function. Instruction must be executed after an indirect call. The hardware must enforce this execution order.

HAFIX++ instrumentation requires compiler support, in that compiled binaries must be generated with instructions in proper places. Figure 1 shows how instrumentation for HAFIX++ proceeds. After linking binaries, a tool is run to instrument labels. Function returns are trivially instrumented, but indirect calls and jumps require extra work.

In the example in Figure 1, the program starts executing on the `main()` function. The function is instrumented. The values for labels are arbitrarily assigned, with the exception

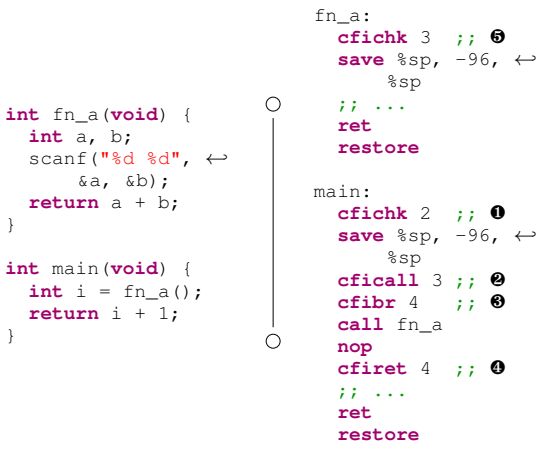

Fig. 1. HAFIX++ instrumentation. The compiler inserts new instructions before and after call/jump instructions, as well as in function entries and targets for indirect jumps.

being for the entry point in `main()`. This is represented by ❶. The runtime expects the entry point to be instrumented with the value of 2. This instruction causes the immediate label to be checked against the contents of the label state register. As the softare continues to execute we prepare to make a function call. First, the `cficall` instruction is executed, ❷, storing the value 3 in the label state register. Then, the instruction `cfibr` is executed pushing the value 4 into the label shadow stack, ❸. The call is then executed. Upon entering `fn_a()`, the `cfichk` instruction is executed, ❺. Its immediate label, 3, is checked against the contents of the label state register. After the function returns, we execute the `cfiret` in ❹. This instruction pops the last value in the label shadow stack and compares it to the immediate value of 4. A successful comparison indicates that we are at the proper return site.

Comparison failures in the HAFIX++ model indicates that a code-reuse attack is underway. For this reason, it is imperative that software instrumentation is properly done, with unique labels for call-return pairs, and for function entries jump targets. To deal with the situation where multiple callers target the same function, HAFIX++ suggests the usage of trampolines, where trampolines can be uniquely instrumented for each function.

2) CFI CaRE: Nyman et al. propose CFI CaRE [19] as means of providing an interrupt-aware CFI policy for ARM microcontrollers without the need for any hardware modifications. CFI CaRE requires code to be instrumented, replacing all control-flow instructions with calls to a Branch Monitor. Furthermore, it leverages the security extensions introduced in the ARMv8-M architecture as means of storing control-flow metadata.

B. Firmware Attestation

Firmware attestation is a technique that is utilized to make a claim about the properties of a device's software. In an attestation scheme, two mutually exclusive parties are involved. The *verifier* in an attestation scheme is a trusted party which can determine whether a device is operational or not given information received about a device. The information is collected by a *prover*, which usually resides on the device itself. The main challenge in attestation approaches is the design and implementation of the prover. The prover must collect enough information about a device, as well as provide it to a verifier in a way so that the provided information can not be forged by an attacker. In this section, we summarize previous attestation works.

1) SMART: Eldefrawy et al. propose a small root of trust for embedded devices in SMART [20], providing a static remote attestation solution. It incorporates the prover into a small on-chip ROM. This code cryptographically hashes and computes an HMAC [21] over a range of code based on the attestation request. The computed HMAC is sent to a remote verifier to ensure correctness. The HMAC is computed using a pre-shared attestation key that is securely stored in the device, as well as a *nonce* that is sent by the verifier as part of the attestation request with the objective of avoiding replay attacks. Leakage of secrets is avoided by ensuring memory erasure whenever the ROM code finishes executing. Further precautions are taken to avoid indirect leakage by controlling execution within the ROM, allowing only a single point of entry and a single exit point. Since ROM code is formally verified to be memory safe, no code reuse attacks are possible to leak the secret key.

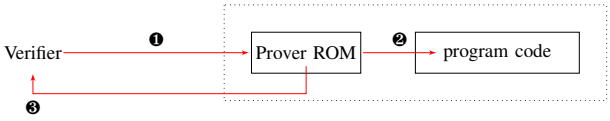

Fig. 2. Flow of operations in SMART. A remote verifier sends a request to a device, to which the prover responds by securely hashing the requested portion of program code using a pre-shared key.

We show the basic flow of SMART in Figure 2. When the prover receives an attestation request from the remote verifier ❶, the prover suspends the currently running task and hashes the requested portion of code memory, ❷. The result of the hash operation is sent to the verifier, ❸. Using the returned HMAC, the verifier can determine whether program code on the device has been mutated.

2) C-FLAT: Abera et al demonstrated in [22] that it is insufficient to use only the device code as means for attestation as code-reuse attacks are able to bypass attestation mechanisms that use this type of system. They proposed Control-Flow Attestation (C-FLAT) [22]. C-FLAT is a remote attestation mechanism that statically aggregates the execution path of a running program, including branches and function returns. The prover collects control-flow information as code executes and hashes it to compute a measure of the device. On an attestation request, the verifier receives the hashed control-flow information and compares to the expected behavior on its end. The device passes attestation if the expected hash matches the obtained one. Loops and conditional branches in C-FLAT are treated and checked as subprograms. Doing otherwise causes the possible number of valid measures that the verifier has to

compute to check device operation increase at an exponential rate.

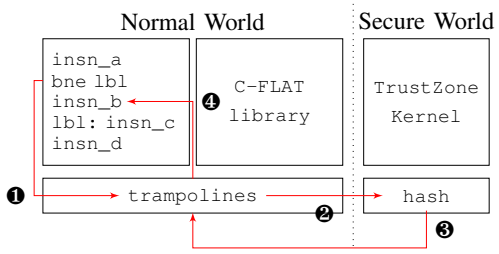

Fig. 3. C-FLAT test implementation model. Trampolines are used to enter the Secure World and perform control-flow attestation.

C-FLAT was implemented and tested in a Raspberry Pi 3 single board computer. We show a simplified view of the implementation in Figure 3. Code is instrumented so that control-flow instructions target a trampoline section which belongs to a *runtime tracer*. The branch instruction in question target the trampoline area ❶. The trampolines allow software to transition to a BLAKE2 [23] Measurement Engine which resides in a TrustZone environment, ❷. The Measurement Engine is part of the device's prover and is responsible for hashing control-flow. When the hash engine finishes executing it returns control to the trampolines, ❸, which ultimately returns control flow to the program ❹. When the remote verifier sends an attestation request to the device, the prover replies with the collected information. The authors report that as the number of control-flow events increase, overhead increases linearly.

3) ATRIUM: Zeitouni et al demonstrated how Time of Check Time of Use (TOCTOU) attacks can bypass attestation schemes, compromising the attestation result. As a result, the authors propse ATRIUM [24]. ATRIUM borrows concepts from C-FLAT and SMART in that it utilizes control-flow and code being executed as part of the information collected by the prover to generate a measure of the device. However, unlike C-FLAT and SMART, ATRIUM dynamically collects this information by tapping into the processor's pipeline to extract both control-flow and instruction information. This allows live analysis of the code being executed by the CPU. The obtained information is hashed using a hardware implementation of the BLAKE2b cryptographic algorithm.

A RISC-V PULPino core [25] was extended to include ATRIUM. The modified core was synthesized and tested targeting a Virtex-7 XC7Z020 FPGA with minimal hardware overhead. Under the configured conditions, the authors report a total resource utilization of 15% of the total slice registers, 20% of slice LUTs, and 18 kbit of BRAM. Performance overhead ranged from 1.7% to 22.69%, depending on the amount and frequency of control-flow instructions in the tested algorithms.

C. More Defenses

ARM mbed uVisor is a self-contained software hypervisor gives programmers the ability to create secure compartments with the characteristic of being independent of each other [26]. ARM uVisor targets the popular Cortex-M3 and Cortex-M4 microcontrollers. Containers execute in non-privileged mode, and with the use of the ARM Memory Protection Unit (MPU) access to critical system resources is restricted. Resources are exposed to containers through the service call interface.

In a similar vein, Minion [27] provides guarantees close to those of uVisor. Minion ensures that the real-time constraints of an embedded operating system are not violated. Much like in uVisor, Minion dynamically reconfigures the ARM MPU whenever a task switch occurs. This way, Minion creates a separation between running processes. MPU information is encoded in a bitmap-like data satructure, which is associated to every task at compile time. As further precaution, only a small codebase is executed with privileges.

Raj et al. propose fTPM as a software implementaiton of a Trusted Platform Module utilizing ARM TrustZone in [28]. Authors note that portions of the TPM specification cannot be fully implemented without contributing factors from the SoC vendor. For example, authors note that a secure clock cannot be properly made, nor the case with secure storage. This is because in order to provide a secure clock or storage, the SoC vendor must have designed their product with this in mind: a real-time clock module and a non-volatile memory must be gated inside the TrustZone environment. However, authors were capable of implementing a large portion of the TPM specification within the TrustZone environment, demonstrating that even without full SoC support, advanced security functions can still be provided with the ARM Security Extension.

V. Discussion

In this section we discuss the matter of deployability of these defenses in terms of the requirements from device manufacturers, SoC vendors, and IP core licensors.

Most of these defenses require some form of change to the hardware platform they run on. CFI CaRE, as well as Minion and uVisor are exceptions to this. Hardware changes are not well within the scope of most vendors. For example, devices built around ARM-based cores rely on SoC vendors for the main component of their device. In turn, SoC vendors license the CPU core in their integrated circuit from ARM. Device manufacturers who wish to deploy defenses that require hardware modifications need for ARM to implement the required hardware primitives or additions to the instruction set into the IP that is eventually licensed from SoC vendors.

We note that in the case of SMART [20], Texas Instruments (TI) controls the processor core IP and also designs and sells microcontrollers using this core. It is reasonable to believe that, if there was interest on their side, a defense mechanism such as SMART can be implemented[1]. We should mention that similar primitives already exist in the MSP430 architecture. Some microcontrollers offered by TI include the IP Encapsulation option which is capable of isolating sensitive code with different permissions [29]. This feature can be extended by TI to provide the necessary functionality to implement SMART.

[1]Please be aware that ARM does not allow for any customized microprocessor architecture modifications so far.

In contrast, mechanisms such as CFI CaRE, Minion, and uVisor do not require currently unavailable hardware primitives to be implemented. CFI CaRE makes use of the ARMv8-M Security Extension, while Minion and uVisor make use of the ARM Memory Protection Unit, which can be found in popular embedded platforms such as the STM32F407.

Control-Flow Integrity approaches also require full knowledge of the control-flow graph (CFG) of the program at hand. Unfortunately, it is impossible to provide an algorithm to compute the CFG in the general case, as this would involve solving the halting problem. However, for individual cases, with manual intervention, a CFG can be obtained for a program. This, however, requires time and money in part of the device's manufacturer. Further compounding this issue is the asynchronous nature of embedded devices. Interrupts generate unexpected control-flow transfers which must be handled by the CFI policy at hand. CFI CaRE addresses this to a degree, but with a limitation: CFI CaRE's policy does not allow for an interrupt vector to return to a different position in code, as it would be the case with schedulers on a Real-Time Operating System (RTOS).

Moreover, the overhead of some of this approaches, such as C-FLAT, prove to be prohibitive for performance critical applications. C-FLAT reports an overhead of 72% to 80% when testing performance with Open Syringe Pump. ATRIUM halts the CPU whenever the prover needs time to finish the hash process. This may be detrimental for systems that require a 100% uptime.

VI. Conclusion

In this paper, we briefly introduced the emerging threats to IoT systems. While these attacks are not new to general computing systems, the ever increasing complexities of IoT systems makes these attacks applicable to IoT devices. A series of hardware-assisted protection mechanisms were also summarized which, compared to software-based solutions, are more effective countering attacks and cause less performance overhead. We expect that more hardware-oriented solutions will be proposed. In parallel, we hope that microprocessor and SoC providers will quickly adopt these hardware based methods for IoT security.

Acknowledgement

This work is partially supported by the Department of Energy through the Early Career Award and the National Science Foundation (CNS-1801599). Any opinions, findings, conclusions, and recommendations expressed in this material are those of the author and do not necessarily reflect the views of the U.S. Department of Energy or the National Science Foundation.

References

[1] A. Nordrum, "Popular internet of things forecast of 50 billion devices by 2020 is outdated," 2016.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[3] D. Rosenberg, "Reflections on trusting trustzone," *BlackHat USA*, 2014.

[4] Mitre Corporation, "CVE-2015-4421: Huawei tzdriver Module Vulnerable checks," 2015.

[5] ——, "CVE-2015-4422: Huawei TEEOS Vulnerable Checks," 2015.

[6] ——, "CVE-2015-6639: QSEE - PRDiag* Commands Privilege Escalation," 2015.

[7] ——, "CVE-2018-16522: AWS secure connectivity modules – uninitialized pointer free," 2018.

[8] ——, "CVE-2018-16526: usGenerateProtocolChecksum memory corruption," 2018.

[9] ——, "Cve-2017-13209: Insecure permissions check that allows hal service change," 2017.

[10] P. Larsen and A.-R. Sadeghi, Eds., *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2018.

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

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

[13] ARM, "Building a secure system using trustzone technology," *ARM Limited*, 2009.

[14] ARM Limited, *ARMv8-M Architecture Reference Manual*, 2019.

[15] ISO/IEC, *ISO/IEC 11889:2015 Trusted Platform Module*, 2015.

[16] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 74.

[17] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 38–49.

[18] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, "Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 2016, pp. 1–6.

[19] 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.

[20] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "Smart: Secure and minimal architecture for (establishing dynamic) root of trust." in *NDSS*, vol. 12, 2012, pp. 1–15.

[21] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions: The hmac construction," *RSA Laboratories' CryptoBytes*, vol. 2, no. 1, pp. 12–15, 1996.

[22] 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.

[23] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn, and C. Winnerlein, "Blake2: simpler, smaller, fast as md5," in *International Conference on Applied Cryptography and Network Security*. Springer, 2013, pp. 119–135.

[24] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *International Conference On Computer Aided Design (ICCAD)*, 2017.

[25] E. Zurich and U. of Bologna, "PULP Platform," http://www.pulp-platform.org/.

[26] M. Meriac, "Practical real-time operating system seecurity for the masses," 2016, https://www.mbed.com/en/technologies/.

[27] 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.

[28] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A software-only implementation of a TPM chip," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 841–856.

[29] Texas Instruments, *MSP Code Protection Features*, 2015, slaa685.