2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)

Performance of Secure Boot in Embedded Systems

Christos Profentzas,† Mirac Günes,† Yiannis Nikolakopoulos,† Olaf Landsiedel,†‡ Magnus Almgren†
†Chalmers University of Technology, Sweden
‡ Kiel University, Germany
Email: {chrpro,mirac,ioaniko,magnus.almgren}@chalmers.se, ol@informatik.uni-kiel.de

***Abstract*—With the proliferation of the Internet of Things (IoT), the need to prioritize the overall system security is more imperative than ever. The IoT will profoundly change the established usage patterns of embedded systems, where devices traditionally operate in relative isolation. Internet connectivity brought by the IoT exposes such previously isolated internal device structures to cyber-attacks through the Internet, which opens new attack vectors and vulnerabilities. For example, a malicious user can modify the firmware or operating system by using a remote connection, aiming to deactivate standard defenses against malware. The criticality of applications, for example, in the Industrial IoT (IIoT) further underlines the need to ensure the integrity of the embedded software.**

One common approach to ensure system integrity is to verify the operating system and application software during the boot process. However, safety-critical IoT devices have constrained boot-up times, and home IoT devices should become available quickly after being turned on. Therefore, the boot-time can affect the usability of a device. This paper analyses performance trade-offs of secure boot for medium-scale embedded systems, such as Beaglebone and Raspberry Pi. We evaluate two secure boot techniques, one is only software-based, and the second is supported by a hardware-based cryptographic storage unit. For the software-based method, we show that secure boot merely increases the overall boot time by 4%. Moreover, the additional cryptographic hardware storage increases the boot-up time by 36%.

***Index Terms*—Embedded Systems, Internet of Things, Secure Boot, System Security**

I. Introduction

The Internet of Things (IoT) will bring connectivity to everyday objects and devices, including vehicles [1], autonomous robots [2], and smart home appliances [3] (e.g., smart vacuum cleaners, smart cookers, smart heaters). While Internet connectivity allows numerous new applications and use-cases, it exposes devices to the security threats of the Internet: if we connect a device to the Internet, it will certainly be attacked and potentially penetrated, i.e., intruders can read data or even modify executable system files. Such modifications are especially critical in the context of the IoT, as the devices often control physical objects such as the cooling system of a refrigerator or the engine of a car.

Today, we commonly find a secure boot process in regular computer systems, including personal computers [4], data centers [5] and also portable devices such as smartphones or tablets [6]. Those computers usually include extra hardware (e.g., a Trusted Platform Module) to ensure the integrity of the firmware and the operating system during the boot process. However, in the domain of embedded systems, secure boot is often overlooked. Therefore, common IoT devices rarely secure the boot process and fail to assure software free from manipulation [7]. The absence of secure boot opens the door to attacks on mission-critical IoT systems. For instance, recent work demonstrates attacks that alter the firmware of interconnected industrial robots [8]. A secure boot mechanism would have detected such modifications during the boot-up process.

Securing the boot process in embedded devices leads to two significant overheads. Firstly, a secure boot process adds additional hardware and complexity. Embedded devices need efficient and simple designs since they face several constraints when it comes to energy consumption and memory capacity. Secondly, verifying the integrity of the operating systems or firmware adds a further delay to the boot process. In some applications, longer boot time may not be affordable. For example, micro-controllers used in the automotive industry should be able to boot almost immediately, preferably in the sub-second domain, so that the vehicle can be used directly after ignition [9]. Other examples are smart home devices where users frequently turn them on and off, like vacuum cleaners and electric kettles. For those devices, longer boot-up times lead to less usability for their users.

Embedded devices include a range of different micro-controllers, which we can classify into three groups: small scale (8–16 bit), medium scale (16–32 bit), and sophisticated micro-controllers. This paper focuses on securing the boot process of medium-scale microcontrollers (e.g., Raspberry Pi, Beaglebone) equipped with an embedded operating system.

The paper makes two contributions: (1) We examine two different approaches to secure the boot process of an embedded device. (2) We show the performance and runtime overhead of the secure boot for those two approaches. Our results underline the trade-offs between security and performance: we present the tradeoffs ranging from 58ms to 245ms for a software-based secure boot, which is 5.6–16% of the boot-loader execution time, and 1.4–4% of the entire boot time (4065ms) of an off-the-shelf Linux distribution. For the hardware-based secure boot technique, we observe an overhead of 1900ms, which is 72% of the boot-loader execution time, and 36% of the entire boot time (5352ms) of an off-the-shelf Linux distribution.

The remainder of the paper is structured as follows. First, Section II introduces basic concepts and definitions of secure boot. Next, Section III presents our threat model. We present secure boot on off-the-shelf embedded hardware in Section IV and evaluate its performance in Section V. In Section VI we

2325-2944/19/$31.00 ©2019 IEEE
DOI 10.1109/DCOSS.2019.00054

IEEE computer society

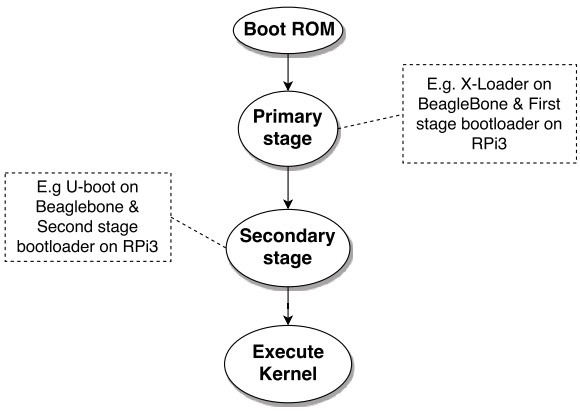


Fig. 1: The boot process of the medium-scale embedded systems consists of multiple stages. The Boot ROM and first stage boot-loader are hard-coded by the manufacturer and are hard to modify. The second stage and the kernel code are modifiable and usually stored in flash memory.

discuss the limitation of secure boot in IoT devices. Section II presents related work and Section VII concludes the paper.

II. Background and Definitions

In this section, we introduce the required background for both the boot process of a medium-scale embedded device and secure boot in particular.

Boot Process. Commonly, manufacturers of embedded devices divide the boot-up process [10] into several stages (see Fig. 1). The purpose of each stage is to prepare the CPU and transfer code fragments from external to internal memory. Eventually, the boot-loader loads the operating system and starts the execution of the kernel. The boot process begins with the Boot ROM, provided by the manufacturer, which sets and initializes the peripherals. The Boot ROM prepares the system to execute the primary boot stage. The primary stage configures the system and prepares the memory for loading the secondary stage boot-loader. The secondary stage is the actual bootloader of the operating system.

Commonly, both the Boot ROM and the primary stage are tamper-proof, as both are hard-coded in the device firmware. However, manufacturers allow the modification of the second stage to provide flexibility and support for different bootloaders and operating systems. As a result, a secure boot process has to ensure the integrity of the kernel and application code before executing the secondary stage.

Security Challenge. The code in each stage can change the overall status of the system and often the next-stage software. As a result, we cannot trust a self-verified software, as it can be modified to provide a false verification status. Therefore, we need to verify in advance, and before we run each piece of software that we give control over the system.

Secure Boot. In a secure boot process, an inherently trusted component triggers the boot process, which is a tamper-proof component referred to as the *Roots of Trust* (RoT) [11].

TABLE I: Terminology comparison

Term	Termination	RoT	Verification	Additional HW
[16] Secure boot	Auto-termination	Boot ROM	By certificate authorities (Remote attestation)	Not specified
[5] Trusted boot	Letting users decide	Boot ROM	Compare hash values	HSM
[13] Verified boot	Letting users decide	Boot ROM	Stored cryptographic hash comparison	Not specified
[21] Measured boot	No termination	BIOS	Measures hash of objects and logs them	Not specified

The Trusted Computing Group (TCG) [11] defines RoT as a set of functions designed to be trusted by the operating system. In embedded systems, RoT can be the Boot ROM (see Fig. 1), which verifies the next-stage software and executes only authentic software. Each stage verifies the integrity of the next one leading to a *Chain of Trust*. For the verification, we can use a dedicated monitoring hardware co-processor. TCG has defined an international standard called the Trusted Platform Module (TPM), which defines the properties that those modules need to fulfill.

We note that different standards and vendors use various terminology to describe a secured boot process: Common terms include, for example, Secure boot, Trusted Boot and Verified boot. Different solutions have been defined and implemented in specific environments including personal computers, data centers, routers, and mobile phones [5], [12], [13]. Especially, *Secure Boot* has been among the standard techniques to define a secured process to assure the integrity of each booting steps [14]. In table I we compare the terminology for the existing techniques.

Related Work. Recent works demonstrate the need for securing the boot process of connected devices from consumer level printers to industrial robots [8], [15]. There is a large body of research in the field of securing and verifying the boot process. Khalid et al. [16] discuss the difference between secure and trusted boot and further evaluate the performance overhead using FPGA boards. Liu et al. give a slightly different approach for system verification. [17] and Lebedev et al. [18], where they are giving examples of a remotely attested system using FPGA embedded systems. In contrast to our work, they focus on FPGA embedded systems while this paper focus on embedded IoT systems. From the practitioner's side, Google's Chromium OS uses verified boot, which builds a chain of trust [13]. For recent work regarding IoT devices, Asokan et al. [19] focus on solutions regarding the firmware update on large-scale IoT deployments. For constraint devices, Boot-IoT [20] propose an authentication scheme towards secure bootstrapping.

III. Adversary Model

In this section, we discuss attack vectors on the boot process of IoT systems and introduce our adversary model. A medium-scale embedded device commonly consists of the application itself, an embedded operating system, and the boot firmware.

The boot firmware is similar to a BIOS in commodity computers and manages the initial boot process, as discussed in Section II. From a security perspective, a secure boot process has to ensure the integrity of each of these components, i.e., that none of them has been modified maliciously [22]. In the context of connected embedded devices, i.e., IoT devices with Internet connectivity, for example, via 5G/LTE, Bluetooth or WiFi, this leads to two main directions of attack: (1) the traditional attack vector of gaining physical access to the device and (2) adversaries can manipulate the firmware via their Internet connection. Thus, connectivity opens new attack vectors, when compared to traditional embedded devices without connectivity.

To compromise a connected IoT device, an adversary may use security holes in both operating systems and its applications to trigger execution of remote, malicious code. Via this code, an adversary can potentially modify data [23], the OS [24] and also the boot process [15]. The adversary's goal is to make a permanent malicious modification, which is unobservable to security analysis. Moreover, we argue that for most IoT devices connectivity is essential for their operation, i.e., they cannot provide their services to the users without connectivity. Thus, just disabling Internet connectivity to close this attack vector is not an option for the vast majority of applications. Via physical access, the adversary can directly manipulate and modify the application, OS, and the boot process. The TPM is also exposed by an adversary with physical access, ongoing research by using Physical Unclonable Functions (PUF) [25] is a promising solution. In our threat model and further system design, we focus on the new attack vector that Internet connectivity brings.

IV. Systems Overview

In this section, we present and discuss two system designs to secure the boot process of an IoT device equipped with an operating system such as embedded Linux: one design is based solely on software mechanisms, and one additionally utilizes hardware primitives. Both designs have specific trade-offs regarding complexity, overhead, and system cost. Both approaches are established [5], [13], [16] in the field, and we do not claim their novelty. Instead, the contribution of this paper lies (1) in comparatively evaluating the overhead that both add to the boot process and (2) in contrasting this overhead to the security each design provides.

A. Software-based Secure Boot with U-Boot

For the software-based method, we rely on the U-Boot [26] bootloader to verify the integrity of the operating system. In our system design, we make the following assumptions. Firstly, the pre-boot environment of U-Boot has to be trusted, meaning that the security of the boot stages before U-Boot cannot be modified. Typically, the manufacturer embeds the first stage in the Boot ROM. Secondly, U-Boot has to be placed in read-only memory since there is no prior verification of the booting process. Lastly, this design requires read-only storage of the cryptographic hashes used to verify the integrity of the operating system.

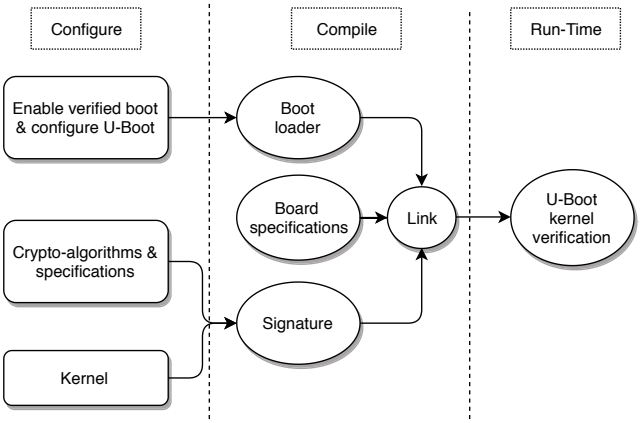


Fig. 2: From U-Boot configuration to deployment: First, we configure the U-Boot to include the verification module. Next, we link the object files to produce the secure version of U-Boot. Finally, we deploy the executable file on the platform.

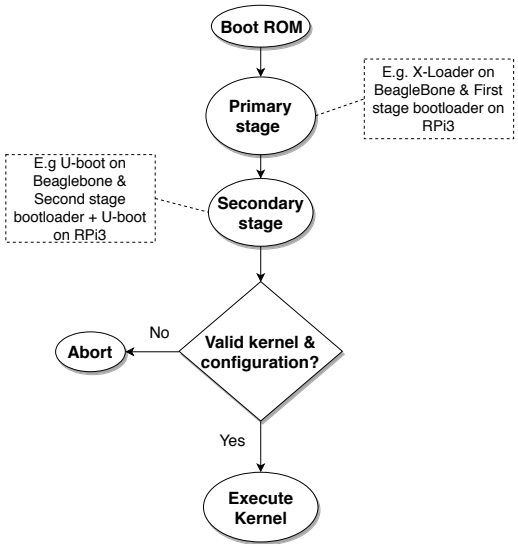


Fig. 3: Secure boot sequence with U-Boot: U-Boot runs as the second stage, which verifies the kernel and the chosen configuration. U-Boot passes the control of the system to the operating system only after successful verification.

U-Boot divides the security process into three steps: *Configure*, *Compile* and *Run-Time*, see Figure 2. The software-based secure boot process extends the boot process with an additional verification step, see Figure 3. As a result, the bootloader only boots the operating system once it has successfully verified the integrity of the operating system. In practice, U-Boot binds the kernel with the hardware information of the board. Thus, U-Boot verifies that the kernel is correct and it will run on the specific hardware configuration.

To verify the integrity of the kernel efficiently, we need

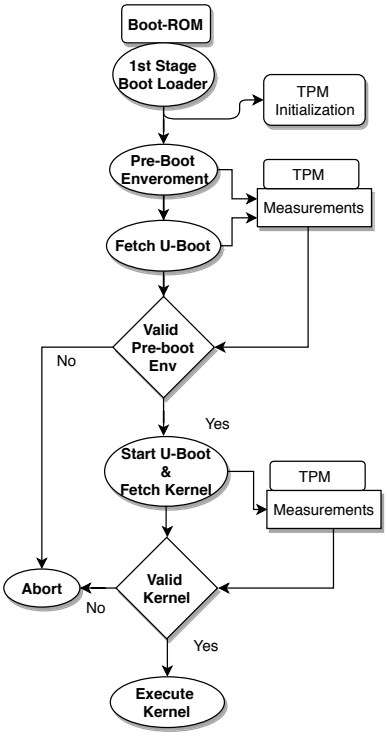


Fig. 4: Secure boot with a TPM co-processor: The manufacturer provides the boot-ROM and first stage bootloader. We split the second stage into two phases: 1) The pre-boot environment, where we check the integrity of U-Boot. 2) U-Boot execution, where we check the integrity of the operating system.

to resolve the digital block data of each image to a single value; a conventional method is to use cryptographic hash functions [27]. Cryptographic hash functions map an arbitrarily long data to a small and fixed output, but they need to fulfill specific properties to be considered cryptographically secure [27]. U-Boot supports three cryptographic hash functions, namely MD5, SHA-1 & SHA-256 [27]. The hash functions have the following digest sizes: (1) MD5: 128-bit (2) SHA-1: 160-bit (3) SHA-256: 256-bit. Finally, the hash digest is being signed by the private key, and the bootloader (U-Boot) can verify the authenticity of the hash value by applying its public key. Various public key algorithms could verify the hash digest of the image. Popular public key cryptographic algorithms are RSA [27] and Elliptic Curve Cryptography (ECC) [27]. U-Boot currently supports only RSA, and the two supported key sizes are 2048-bit and 4096-bit. The key size is the critical factor of public key algorithms; bigger key sizes are more difficult to break. On the other hand, the larger the key size, the longer the verification takes [27].

B. Hardware Security for Secure Boot

After introducing the software-only secure boot process, we next introduce secure boot with a hardware security module.

TABLE II: Hardware specifications

Model	Hardware
Raspberry Pi 3 Model B	ARM® Cortex A53 - 1.2GHz(quad-core) 1 GB LPDDR2 RAM
Beglebone Black C	ARM® Cortex A8 - 1GHz 512MB DDR3 RAM
Cryptocape (by Cryptotronix)	TPM-Module:AT97SC3204T

This design further increases the security of the boot process at the cost of adding additional hardware and boot latency.

This design is based on the TPM module as proposed by Khalid et al. [16]. The method always starts with the initialization of the TPM, which ensures that the TPM is activated. The TPM provides the following functions defined by the standard [28]: (1) **Measurement**, TPM calculates the hash of the input data using SHA-1. (2) **Extend**, TPM takes the current hash-value inside the register, appends the Measurement and produces a new hash value (3) **Control Transfer**, the TPM passes the system control to the successfully verified entity. The process continues by calculating the cryptographic hash value of the boot environment, which includes the system configuration before loading the secondary stage boot loader (see Section II). The process consists of a repetitive **Measure–Extend–Execute** procedure [16]. This method is a common way to ensure a **Chain of Trust** [29], which verifies the integrity of the different stages step by step. The TPM transfers the control of the system to each measured image only if it has successfully verified the extended hash-value. In the case of failure, the boot process will halt as shown in Figure 4.

For this technique, we make the following assumption: The first entity of the *Chain of Trust* needs to be trusted, which in this case is the boot ROM and first stage boot-loader. The manufacturer should embed this code in a way so that nobody can modify it, for example, by placing it in read-only memory.

V. Evaluation

This section evaluates both system designs in detail and focuses on the overhead of the secure boot process in each system. Practical application scenarios motivate this evaluation, for example, vehicles are expected to be immediately usable after ignition, imposing a low-latency requirement between turning on the ignition and the full boot up of all the microcontrollers of a vehicle. Also, home appliances like smart vacuum cleaners and smart heaters experience frequent turn off and on by their users, where the boot time can affect the usability of the device.

A. Experimental Setup

We implement our system design on two generic embedded platforms for IoT applications [30] that are readily available, namely a Raspberry Pi and a BeagleBone (see Table II). Finally, we extend the capabilities of the BeagleBone to support hardware cryptographic primitives.

B. Evaluation Results

Next, we present the results of our evaluation of the overhead of the secure boot process. We begin with the

TABLE III: Verification overhead of TPM

Overhead	Average time (ms)
TPM Initialization	993
Measurements	138
Extend PCR values	791
Total	1923

software-based method and continue with the hardware-based technique. The results summarize the performance of our system design.

1) Software Mechanism of U-Boot: Figures 5a, 5b, 5c & 5d present the overhead of verification using different key sizes and three different hash functions (MD5, SHA-1, SHA-256). The average execution time for U-Boot without any security mechanism is 976ms for the BeagleBone and 903ms for Raspberry-Pi. These numbers form the baseline to compare the overhead of secure boot. The entire boot time of the off-the-shelf Linux-kernel (Debian GNU 7) on BeagleBone is 4s, and for Raspberry Pi (Debian Jessie 4.4) is 7s (see Figure 6).

We begin evaluating the overhead that different hash functions in U-Boot bring. In this evaluation, we set the key-size of RSA to 2048-bit and use BeagleBone. With the MD5 hash function, the average overhead is 58ms, representing 5.7% of the U-Boot execution time (see Figure 5a), and 1.4% of the entire boot time of the Linux kernel (see Figure 6a). With the SHA-1 hash function, the average overhead increases to 117ms, which is 11% and 2.8% of the U-Boot execution (see Figure 5a) and the entire boot time of the Linux kernel (see Figure 6a), respectively. For SHA-256 hash function, the overhead increases to 164 ms, 15% and 4%, respectively (see Figure 5a & 6a).

Next, we increase the key-size of RSA to 4096-bits, using the same hash functions and BeagleBone. This key-size increases the overhead by 24ms to 35ms depending on the hash function (see Figure 5b). For the same experiment using Raspberry Pi, we observe similar results (see Figures 5c & 5d).

2) TPM Hardware on BeagleBone: Table III presents the overhead after introducing the hardware primitive (TPM). The initialization of the TPM takes 993ms. TPM uses SHA-1 as the hash function and the overhead of calculating the measurements (see Section II) is 138ms. For extending the Registers (PCR) TPM takes 791ms. Overall, the whole method takes 1923ms, which adds an overhead of 36%.

C. Discussion

Configuration choices, like the hash function, have different performance impact and trade-offs. For example, MD5 provides better performance, but it is no longer recommended [31]. FIPS 180-4 Secure Hash Standard (SHS) recommends SHA-1 and SHA-256, but Stevens et al. [32] have found the first collisions on SHA-1.

Regarding the performance of Secure boot with TPM, we have noticed a higher verification overhead (approximately eight times more) compared to the software-based technique (Verified U-Boot). The main reason for this is that there are more measurement requirements in this method: we verify the kernel, U-Boot and boot states which include the complete system configurations. Another source of overhead is the initialization time of the TPM. It is worth to notice that the extra hardware does not intend to accelerate the cryptographic functions, but rather to provide stronger security properties as we explained in previous sections.

To conclude, if we compare the different parts of the boot-time we notice that loading the kernel is the most time-consuming part. Thus, we argue that while the overhead of Secure Boot is not negligible, its overall performance overhead is limited. This performance makes Secure Boot a practical solution to secure the software-stack of medium-size devices in the Internet of Things. For application where boot-up needs to be reduced further, customized, modular OS kernels with application-specific functionality could be an option to improve the boot performance.

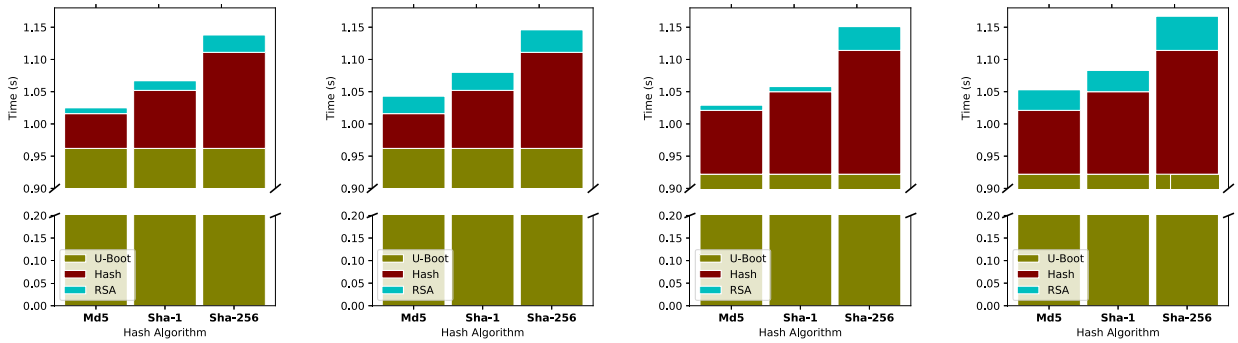
VI. Limitations and Discussion

In this paper, we evaluate the time performance of a software- and hardware-based secure boot techniques. The boot performance, i.e., the time until a device has booted, is a critical aspect, for example in industrial IoT systems like autonomous vehicles. While secure boot ensures system integrity, it cannot protect against all attacks. In the following we discuss key limitations:

Availability. A general limitation of secure boot is the lack of protection against persistent DOS-attack caused by the mechanism. An attacker can try to modify the integrity of the operating systems repeatedly and reboot the system. This attack will prevent the system from booting-up, and it is possible a DOS-attack caused by the security mechanism. We need secure boot techniques that can adapt and recognize such an attack vector. Moreover, this attack highlights the complexity of the security techniques in IoT which involves heterogeneous devices. For example, a user can still expect a compromised smart-light to work in safe mode. However, a compromised industrial robot which involves safety-critical aspects, it should immediately halt.

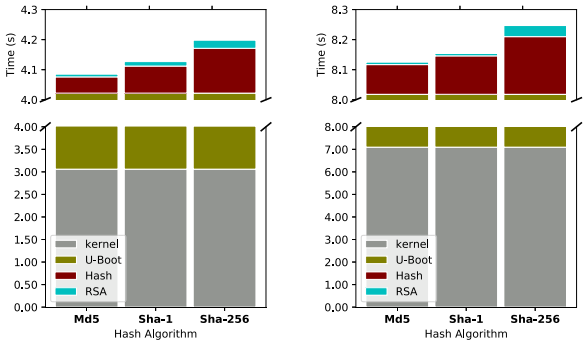
Applicability. In this paper, we focus on medium size embedded boards (e.g., ARMv7), where resource constraints do not prevent the extension of the boot process. In small constraint IoT devices (e.g., ARM Cortex M4) similar approaches may not be applicable for several reasons. First, those devices do not separate the boot process in stages. Moreover, we need to implement a bootloader efficient to meet memory and run-time constraints for those devices. Second, the firmware and the application is stored together in flash memory, without any protective barrier. Finally, the limited CPU power makes it hard to make the cryptographic calculation. We can expect a different overhead compare with that of the evaluation in section V. Hardware support like TPM is not available for small IoT device.

Scalability. One of the benefits of the Internet of Things is the ability to upgrade the firmware over-the-air (OTA), i.e., via the Internet connection of the device. This flexibility provides



(a) RSA 2048-bits key-size on BeagleBone (b) RSA 4096-bits key-size on BeagleBone (c) RSA 2048-bits key-size on Raspberry Pi (d) RSA 4096-bits key-size on Raspberry Pi

Fig. 5: For the evaluation of U-Boot using the BeagleBone & Raspberry Pi, we apply RSA with a key size of 2048 and 4096 bits. The U-Boot time refers to the performance without the verification module. All figures compare the three available hash functions: MD5, SHA-1, SHA-256. Note the graphs have discontinuing scale numbers



(a) Entire Boot Time: BeagleBone (b) Entire Boot Time: Raspberry Pi

Fig. 6: Evaluation of entire boot time using the BeagleBone & Raspberry Pi. The RSA key-size is 2048-bits, and we compare with the three available hash functions: MD5, SHA-1, SHA-256. The RSA overhead is very small and barely visible. Note: the graphs have discontinuing scale numbers

scalability for vendors to upload the firmware and application updates to already deployed IoT devices. However, after each update, the credentials matching the firmware need to be updated. This update is challenging, as many TPM modules do not allow direct updates of credentials to protect against attacks.

VII. Conclusion

The security aspects of embedded systems become more critical with the rise of the Internet of Things. Secure boot is one of the primary tools to secure IoT applications and their operating system. This paper presents and evaluates trade-offs regarding the implementation and the performance of secure boot. Our results show that the software-based method increases the overall boot-up time by 4%. The hardware-based one adds an overhead of 36%.

For future works, we plan to evaluate the impact of different system configurations and kernel configurations on the performance of secure boot. Moreover, we will focus on the design, implementation, and evaluation of secure boot on smaller and more constrained devices (e.g., ARM M4).

References

[1] M. Gerla, E. K. Lee, G. Pau, and U. Lee, "Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 241–246.

[2] Z. Bi, L. D. Xu, and C. Wang, "Internet of things for enterprise systems of modern manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1537–1546, May 2014.

[3] S. Kashyap, V. S. Rao, R. V. Prasad, and T. Staring, "Cook over ip: Adapting tcp for cordless kitchen appliances," in *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, April 2018, pp. 1–12.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, May 1997, pp. 65–71.

[5] S. Berger, K. Goldman, D. Pendarakis, D. Safford, E. Valdez, and M. Zohar, "Scalable attestation: A step toward secure and trusted clouds," *IEEE Cloud Computing*, vol. 2, no. 5, pp. 10–18, Sept 2015.

[6] The Android Team, "Verifying boot," https://source.android.com /security/verifiedboot/verified-boot, 2017.

[7] S. Eresheim, R. Luh, and S. Schrittwieser, "On the impact of kernel code vulnerabilities in iot devices," in *2017 International Conference on Software Security and Assurance (ICSSA)*, July 2017, pp. 1–5.

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

[9] M. Åsberg, T. Nolte, M. Joki, J. Hogbrink, and S. Siwani, "Fast linux bootup using non-intrusive methods for predictable industrial embedded systems," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, Sept 2013, pp. 1–8.

[10] Texas Instruments, "Boot sequence," http://processors.wiki.ti.com /index.html.

[11] H. C. A. van Tilborg and S. Jajodia, Eds., *TCG Trusted Computing Group*. Boston, MA: Springer US, 2011, pp. 1279–1279.

[12] K. Dietrich and J. Winter, "Secure boot revisited," in *2008 The 9th International Conference for Young Computer Scientists*, Nov 2008, pp. 2360–2365.

[13] The Chromium OS team, "Verified boot," http://www.chromium.org/ chromium-os/ chromiumos-design-docs/verified-boot, 2009.

[14] J. D. Tygar and B. S. Yee, "Dyad: A system for using physically secure coprocessors," *Technical Report CMU-CS-91-140R*, 1991.

[15] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation." in *NDSS*, 2013.
[16] O. Khalid, C. Rolfes, and A. Ibing, "On implementing trusted boot for embedded systems," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2013, pp. 75–80.
[17] Y. Liu, J. Briones, R. Zhou, and N. Magotra, "Study of secure boot with a fpga-based iot device," in *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2017, pp. 1053–1056.
[18] I. Lebedev, K. Hogan, and S. Devadas, "Invited paper: Secure boot and remote attestation in the sanctum processor," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 46–60.
[19] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Assured: Architecture for secure software update of realistic embedded devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, p. 2290–2300, Nov 2018.
[20] M. Hossain and R. Hasan, "Boot-iot: A privacy-aware authentication scheme for secure bootstrapping of iot nodes," in *2017 IEEE International Congress on Internet of Things (ICIOT)*, June 2017, pp. 1–8.
[21] G. Fedorkow, "What's the difference between secure boot and measured boot?" http://forums.juniper.net/t5/Security-Now/What-s-the-Difference-between-Secure-Boot-and-Measured-Boot/ba-p/281251, 2015.
[22] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady, "Security in embedded systems: Design challenges," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 3, pp. 461–491, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015047.1015049
[23] Z. S. Huang and I. G. Harris, "Return-oriented vulnerabilities in arm executables," in *2012 IEEE Conference on Technologies for Homeland Security (HST)*, Nov 2012, pp. 1–6.
[24] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 296–310.
[25] P. Choi and D. K. Kim, "Design of security enhanced tpm chip against invasive physical attacks," in *2012 IEEE International Symposium on Circuits and Systems*, May 2012, pp. 1787–1790.
[26] S. Glass, "Verified u-boot," https://lwn.net/Articles/571031/, 2013.
[27] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 3rd ed. Pearson Education, 2002.
[28] The TCG community, "Trusted platform module (TPM) summary," 2008.
[29] W. Fang, C. Zhou, Y. Zhang, and L. Zhang, "Research and application of trusted computing platform based on portable tpm," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, Aug 2009, pp. 506–509.
[30] K. J. Singh and D. S. Kapoor, "Create your own internet of things: A survey of iot platforms." *IEEE Consumer Electronics Magazine*, vol. 6, no. 2, pp. 57–68, April 2017.
[31] E. Thompson, "MD5 collisions and the impact on computer forensics," *Digital Investigation*, vol. 2, pp. 36–40, 2005.
[32] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 570–596.