RealSWATT: Remote Software-based Attestation for Embedded Devices under Realtime Constraints

Sebastian Surminski
University of Duisburg-Essen
Essen, Germany
sebastian.surminski@uni-due.de

Christian Niesler
University of Duisburg-Essen
Essen, Germany
christian.niesler@uni-due.de

Ferdinand Brasser
Technical University Darmstadt
Darmstadt, Germany
ferdinand.brasser@trust.tu-darmstadt.de

Lucas Davi
University of Duisburg-Essen
Essen, Germany
lucas.davi@uni-due.de

Ahmad-Reza Sadeghi
Technical University Darmstadt
Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Smart factories, critical infrastructures, and medical devices largely rely on embedded systems that need to satisfy realtime constraints to complete crucial tasks. Recent studies and reports have revealed that many of these devices suffer from crucial vulnerabilities that can be exploited with fatal consequences. Despite the security and safety-critical role of these devices, they often do not feature state-of-the-art security mechanisms. Moreover, since realtime systems have strict timing requirements, integrating new security mechanisms is not a viable option as they often influence the device's runtime behavior. One solution is to offload security enhancements to a remote instance, the so-called remote attestation.

We present RealSWATT, the first software-based remote attestation system for realtime embedded devices. Remote attestation is a powerful security service that allows a party to verify the correct functionality of an untrusted remote device. In contrast to previous remote attestation approaches for realtime systems, RealSWATT does neither require custom hardware extensions nor trusted computing components. It is designed to work within real-world IoT networks, connected through Wi-Fi. RealSWATT leverages a dedicated processor core for remote attestation and provides the required timing guarantees without hardware extensions.

We implement RealSWATT on the popular ESP32 microcontroller, and we evaluate it on a real-world medical device with realtime constraints. To demonstrate its applicability, we furthermore integrate RealSWATT into a framework for off-the-shelf IoT devices and apply it to a smart plug, a smoke detector, and a smart light bulb.

CCS CONCEPTS

• **Networks** → *Cyber-physical networks*; • **Security and privacy** → **Embedded systems security**; • **Computer systems organization** → **Real-time system architecture**.

KEYWORDS

attestation; firmware security; IoT; embedded systems; realtime; critical infrastructure

ACM Reference Format:
Sebastian Surminski, Christian Niesler, Ferdinand Brasser, Lucas Davi, and Ahmad-Reza Sadeghi. 2021. RealSWATT: Remote Software-based Attestation for Embedded Devices under Realtime Constraints. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. *https://doi.org/10.1145/3460120.3484788*

1 INTRODUCTION

Commodity realtime embedded systems often suffer from security vulnerabilities already known from classical computing. However, due to the resource constraints embedded devices often lack basic security mechanisms that are common in most other types of systems [4]. At the same time, realtime applications, which are essential in many safety-critical domains, place highly conservative requirements to guarantee the strict realtime operation.

The need to secure embedded devices is further amplified by the trend of Internet of Things (IoT) to connect previously unconnected and isolated devices to the Internet to enhance features and services. This leads to large vulnerable ecosystems consisting of millions of devices.[1] In particular, any malfunction in realtime devices can have fatal consequences since they perform highly critical real-world tasks in many safety-critical domains as cyber-physical systems, medicine and transportation to name some: A modern automobile can have up to 150 control units [58] and offers a large attack surface [35]; or controllers of industrial robots suffer from a high number of software vulnerabilities [44], as a well-known incident at a steel mill has shown where the blast furnace could not be shut down properly resulting from an attack on its control systems, leading to severe physical damage [65]. These systems are a large subset of IoT devices with sparse resources and strict requirements. Since realtime systems serve highly critical tasks in the field of industry, medicine, and transportation (e.g., as a vehicle control unit), the introduction of security mechanisms is vital but challenging.

[1]The Mirai botnet [5], that compromised and controlled millions of IoT devices, was used for multiple large-scale distributed denial of service attacks, with up to 1 Tbit s^{-1} bandwidth, consisting of more than 300,000 devices [32].

Legacy realtime embedded devices lack hardware features, e.g., secure boot or trusted execution environments (TEEs). Moreover, they are commonly integrated into machines and run customized software, and hence, cannot be simply replaced. On the other hand, incorporating protection mechanisms in software such as control-flow integrity [1] always impacts execution times of tasks [17, 55]. This is highly critical in the realtime realm, since they must adhere to strict timing behavior and hence go through an extensive development and profiling phase. Any changes in the execution, even through instrumentation, e.g., to integrate control-flow integrity (CFI) [17, 55], or abnormality monitoring [48], affect the runtime behavior of the device.

Hence, adequate security solutions for realtime applications, must have strictly limited impact on the realtime operations [52]. Currently there exist no practical solutions that can tackle these challenges. Furthermore, the handling of detected suspicious or malicious behavior is an important question for critical realtime systems. Solutions like control-flow integrity may terminate if an illegal path is executed.

At first glance, a promising solution to tackle these challenges seems to be remote attestation (RA), as it offloads the verification of the monitored device to an external trusted party. RA allows a trusted party, called verifier, to gain assurance about the correctness of the state of a remote device, called prover. It has been used for embedded devices [2, 10, 19, 40] and sensor networks [53]. However, the main challenge for attesting realtime devices is, however, to utilize the attestation independently from the execution of the monitored application.

Moreover, another vital aspect of remote attestation is to get a genuine attestation report from an untrusted device. An attacker could forge the attestation report, for example by using a different device or an emulation of the attested system. There have been a variety of proposals for attestation schemes to address this issue: (1) hardware-based using trusted computing [2], (2) hybrid using custom hardware extensions [10, 19, 20, 34, 40], and (3) software-based [46, 53].

However, none of these solutions is an option for legacy realtime embedded devices: While hardware-based and hybrid approaches to attestation require changes to or customization of the underlying hardware, software-based attestation poses strict timing assumptions on the response of the prover and the verifier induces many requirements upon the implementation of attestation logic and communication [7, 46]. We will elaborate on these approaches in detail in the related work in Section 9.

Our goal and contributions. In this work, we present RealSWATT, the first remote attestation framework that is applicable to realtime systems without requiring any hardware changes. We leverage off-the-shelf hardware and do not require any trust anchor on the attested device. A key aspect of our design is based on the observation that many modern low-cost embedded systems, such as the ESP32, are built on a multicore architecture where the cores are often not fully utilized. Especially in realtime context, multithreading is hard because it is of utmost importance to meet all deadlines under all circumstances. So in practice critical tasks are often not scheduled in parallel. In specific areas, e.g., avionics, there are even regulations to limit the usage of additional cores [13]. As a result, one or more processing cores are idle. We leverage this circumstance and utilize an idle processor core to develop a new attestation framework. This allows the attestation and the realtime tasks to be properly scheduled by the underlying realtime operating system and makes RealSWATT suitable for legacy embedded devices in industry, medicine, and cyber-physical systems.

However, the usage of multicore processors involves tackling several new challenges: while the benign execution only uses one processor core, an attacker can now use all processor cores in order to forge an attestation report. We address this issue by selecting adequate cryptographic functions that cannot be accelerated by parallelization. Furthermore, the communication, especially in wireless networks commonly deployed for IoT devices and cyber-physical system setups, is prune to variation in transmission times. This also conflicts with the strict assumptions for software-based attestation [7].

Because of these required strict assumptions and shortcomings like the vulnerability against compressing attacks [12] software-based attestation has been assumed insecure and infeasible in practice, receiving only little to no attention. It requires specific assumptions about the execution speed of the prover logic on the attested device and precise timing measurements, making the implementation challenging [46]. Software-based attestation however is a good fit for legacy devices, where other attestation schemes simply are not available due to the lack of specialized hardware on the given device. Thus, we have re-evaluated software-based attestation and solved several challenges, which allow us to deploy a software-based attestation scheme in a real-world scenario.

We developed continuous attestation where the verifier sends the next attestation request before it receives the previous response of the prover so that the prover can start the next attestation-run directly after the previous one. This procedure omits the transmission and verification time, so that variations in transmission time do not influence the attestation. Our continuous attestation approach induces the strict requirement that the attacker cannot run two attestation protocols in parallel in order to get a time-span in which no attestation is performed, thus effectively solving the Time-of-Check Time-of-Use (TOCTOU) problem. The combination of these techniques (multicore and continuous attestation) ensures that RealSWATT can reliably attest realtime embedded systems in real-world wireless networks without impairing the realtime operation.

In summary, we provide the following contributions:

- We propose RealSWATT, the first software-based remote attestation framework for realtime-critical devices that works on commodity off-the-shelf low-cost embedded devices.
- We present the first attestation framework that exploits a separate processor core for attestation to ensure the correct scheduling and timing of realtime operations.
- We propose a new scheme called continuous attestation and a network architecture for the software-based attestation of embedded devices which allows us to tackle the strict timing constraints and hardware requirements of existing software-based attestation schemes [7, 46].
- Our framework allows to remotely verify code- and data sections to detect malware infection and malicious changes of configuration parameters.

- For our proof-of-concept implementation, we used one of the most popular IoT platforms ESP32 microcontrollers and conducted a detailed evaluation on a medical device syringe pump. We performed a full end-to-end example with an attack that compromises the syringe pump's configuration which is detected by RealSWATT.
- We implemented RealSWATT into ESPEasy, a framework to use on real-world off-the-shelf IoT devices and used it on different devices such as a smart plug, a smoke detector, and a smart light bulb.

2 BACKGROUND

In this section we explain the concepts and basic properties of realtime applications and remote attestation as foundation of our attestation framework.

2.1 Realtime Systems

In contrast to the best-effort compute system, realtime systems require strict response times of the tasks they are running. There are different classes of realtime systems distinguished by their strictness, i.e., the consequences of missing deadlines; hard, firm, or soft realtime requirements [49]. In the hard class, no deadline may be missed, otherwise there will be severe consequences, akin to a device failure. Typical examples for such systems with realtime constraints are control units for vehicles, e.g., braking or engine control units where missing deadlines have direct influence on the physical world. Also industrial machines, cyber-physical systems, and medical devices often have hard realtime constraints [49].

As realtime devices are often safety-critical there are specific procedures and regulations to be considered during development. Changes induce a new validation process as well as a re-certification in case of safety-critical devices, e.g., in avionics [45].

2.2 Remote Attestation

Remote attestation is a security service that allows a trusted entity, called verifier to verify the integrity of the state of a remote, unstrusted system, called prover [40]. Remote attestation schemes have been used for different devices including embedded devices [2, 10, 19, 40] and sensor networks [53], as they offload the verification of the device (prover) to an external party (verifier).

In a typical attestation protocol, the verifier sends a challenge to the prover who responds with a message that indicates a measurement of the state of the prover device, usually software state. The main challenge in attestation schemes is to obtain trustworthy measurements about the prover's state. There are different approaches to achieve this: (1) hardware-based solutions use trusted computing modules [2], (2) hybrid attestation uses custom hardware extensions [10, 19, 40], and (3) software-based attestation (SWATT) [46] does not need any hardware support. In the following we focus on software-based attestation, in the related work in Section 9, we also elaborate on other attestation approaches.

Software-based attestation (SWATT) can be used on commodity hardware and legacy devices. However, SWATT poses strict timing assumptions on the response of the prover and the verifier. SWATT builds upon the core assumption that an attacker cannot replace or alter the attested device. Hence, the verifier relies on the integrity of the prover's computing capabilities, i.e., the memory and processing speed cannot be manipulated. Based on these assumptions the verifier precisely measures the response times for the attestation requests, this includes the time it takes for the prover to compute the attestation report as well as the communication overhead between prover and verifier. If the response times differ from the expected values, the prover device is assumed to be compromised. Furthermore, it is important that all memory, both used and unused memory, is covered in the attestation. We discuss these requirements in the Security Considerations in Section 8.

These assumptions are however, very strong and pose many hard requirements on the implementation of attestation logic and communication channels [7, 46]. In the real world an attacker can circumvent the attestation in various ways [12].

3 PROBLEM STATEMENT

Detecting software attacks on devices in a connected system is a highly challenging task, in particular if the adversary has gained full control over a subset of devices. Even more so, if the connected devices have high requirements with respect to their timing behavior due to the execution of realtime tasks.

Existing solutions to detect and report attacks in connected systems are either heuristic [23] or pose assumptions that are unrealistic for many critical realtime tasks [46]. Monitoring solutions at the network level are heuristic in nature and suffer from high false-positive rate [63]. Hardware-assisted security mechanisms [34] rely on extensions and components, such as a TEE architecture or cryptographic co-processor (TPM), that are not available is the vast majority of deployed legacy embedded devices. Software-based attestation approaches target legacy devices that cannot provide a trust anchor; however, their approach to ensure integrity of the measurement function inherently conflicts with the execution timing demands of realtime application: Software-based attestation asserts integrity of the measurement procedure by demanding all system resources, to prevent the adversary from using free resources, while precisely measuring its execution time. Beside the inherent conflicts when applied to realtime systems, software-based attestation is the only option (to enable heuristic detection of complete software compromise) for legacy systems.

A secure software-based attestation for realtime embedded systems poses a number of challenges on the underlying design and implementation:

Challenge 1: A secure attestation scheme — without any trust anchor — running tasks with realtime execution requirements needs to overcome the inherent conflict between realtime execution guarantees and integrity guarantees for the measurement procedure to capture the prover device's state.

Challenge 2: Allowing the measurement procedure to respect the realtime demands of the system's tasks could be easily misused by an adversary, e.g., to restore a benign state while the measurement is performed [11]. Therefore, the measurement procedure must be able to capture the system state independent of the execution of realtime tasks.

Challenge 3: Permitting the execution of potential malicious tasks in parallel to the measurement procedure provides the adversary with the option to dynamically adapt and move to itself between memory areas, always restoring the currently measured section of memory.

Challenge 4: In remote scenarios software-based attestation faces the challenge to account for the jitter in network transmission, which prevents the verifier from precisely measuring of the execution time: To avoid false positives the verifier has to tolerate considerable time gaps, which might be exploited by an adversary to manipulate the measurement and hide its existence on the prover device.

Challenge 5: Remote attestation, and in particular software-based attestation, face the Time-of-Check Time-of-Use problem (TOCTOU) [64], i.e., an attestation report only presents a snapshot of the prover's state. The verifier learns no information whether the prover has been compromised (and restored) before the attestation or will be immediately after.

RealSWATT overcomes these challenges by executing the measurement procedure on a dedicated CPU core, while allowing the continuous execution of realtime tasks on another CPU core. Our approach deploys novel techniques to tackle all challenges (C1–C5) in order to design and implement a secure attestation solution.

4 SYSTEM AND THREAT MODEL

RealSWATT attestation is designed to work with legacy IoT devices in real-world network environments. The network architecture is sketched in Figure 1. It targets scenarios with untrusted embedded systems running critical realtime tasks that should be attested by the remote verifier.

4.1 System Model

We consider a system of connected low-end embedded devices executing realtime tasks. We assume an untrusted system running tasks with realtime deadlines, the so-called prover, which is being attested by the remote verifier. Furthermore, we assume that the prover is running a realtime operating system (RTOS), which ensures correct scheduling and proper realtime operation. The RTOS is no mandatory requirement, however, it simplifies the integration of the RealSWATT framework into legacy devices. Without an operating system, the attestation logic has to be manually integrated and ensured, that the added methods do not influence the other tasks running on the device.

The system features a multicore processor, of which one core is not utilized and not required for correct realtime operation. The attested device is connected to a remote verifier via a wired or wireless network. There are no strict timing requirements towards the connection. Bandwidth and timing requirements are elaborated in Section 7.

All devices of the network are known to the verifier device. They can communicate to each other and the verifier directly, however, all communication with other entities is routed through a gateway.

We assume an IoT network structure consisting of multiple IoT devices that are being attested, a trusted verifier, and an IoT gateway for external communication, as sketched in Figure 1. The IoT gateway monitors external communication to detect abnormalities to prevent offloading of attestation tasks. Offloading remote attestation tasks to an external party requires frequent communication

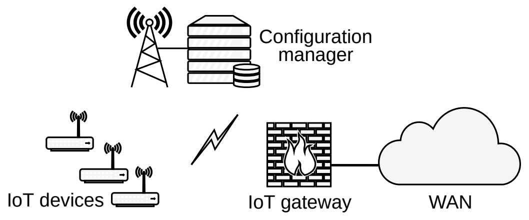

Figure 1: Architecture of the Realtime-Attestation Approach.

to a dedicated remote instance which clearly differs from normal behavior of IoT devices. Monitoring traffic by its origin, goal, packet size, frequency, or content is a common method to secure internal or dedicated IoT networks. For example, the National Institute of Standards and Technology (NIST) suggests to use such gateways to secure communication of embedded devices [9, 54]. In recommendation ITU-T Y.2060 [33], the International Telecommunication Union (ITU) also considers such gateways for IoT networks.

The configuration manager, as shown in Figure 1, is a common component in modern IoT architectures. It keeps track of the devices and their applied configuration. Configuration management software for IoT devices is commercially available [50]. Recent research also considers the use of such a configuration manager to set and organize security features on IoT devices on demand [14]. With the help of such a configuration manager, attestation can be enabled on large scale without the need to setup attestation on each device individually. A further benefit is that the configuration can be easily provided to a verifier and included into the attestation reports.

4.2 Threat Model

The adversary can compromise all embedded devices in the network via software attacks. The adversary knows the benign state and configuration of every device. It can observe all network communication.

The adversary is able to modify program as well as configuration data. Furthermore, the adversary can compromise an embedded device at each point in time as well as restore a devices benign state at any point in time.

The verifier and the gateway are assumed to be immune, i.e., the adversary cannot compromise them. All communication, except between devices and/or the verifier, are routed through the gateway. The adversary cannot introduce additional devices into the network. The devices' hardware cannot be modified or manipulated by the adversary, e.g., by a more powerful device with more memory or a faster processor.

5 CONCEPT OF REALSWATT ATTESTATION

Our design introduces two new concepts: (1) Using a separate processor core for attestation to separate the normal operation and the attestation tasks from each other, and (2) Continuous attestation, i.e., attesting the system continuously during runtime. Multicore processors have many advantages in processing speed and energy consumption [31] and are becoming increasingly widespread [18], even if the development of realtime applications for multicore processors is challenging [45, 57]. We leverage a multicore processor

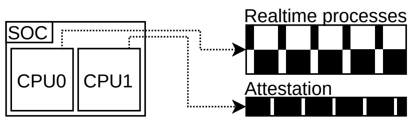

Figure 2: A system with multiple processor cores can use one core for realtime tasks and one core to run the attestation in parallel.

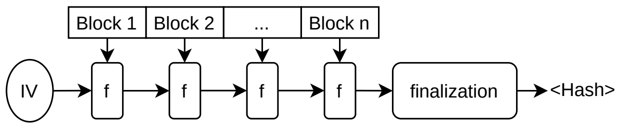

Figure 3: Merkle–Damgård construction of hash values. The process cannot be parallelized for speedup.

design, which is nowadays commonly available on popular IoT platforms, but often not fully utilized.

We observed that in many IoT devices and especially realtime systems with multicore architecture not all cores are fully utilized. In some specific application areas, e.g., avionics systems, there are even regulations that limit the use of additional cores for realtime operation [13]. In realtime computing, there exist several security frameworks that use a separate processor core of a multicore system to implement new security features [47, 61]; for instance, Yoon et al. [61] utilizes it for intrusion detection and other follow-up works leverage it to cover memory usage [62] and analyze system call traces [60].

Software-based attestation (SWATT) relies on precise prediction of the response times of the attested device. The verifier sends a challenge and measures the response time of the prover which includes the execution time of the attestation and the transmission time. Hence, SWATT also relies on direct and undisturbed communication between the prover and the verifier. That is, if the response to an attestation request is delayed, the prover cannot distinguish between a false alarm caused by a transmission delay and an attack. For the latter the delay is caused by the attacker covering her traces.

However, the assumption of undisturbed communication is unrealistic in practice. Nowadays, IoT devices communicate via wired or wireless networks that are shared with many other devices. These devices influence each others transmissions; especially in wireless network like Wi-Fi. Wireless networks inherently use a shared medium that is not only shared between the devices within the network, but also with all other devices using the same frequency band. Hence, traditional software-based attestation [46] cannot be applied to these communication networks. We tackle this limitation by developing continuous attestation that eliminates the transmission time from the timing measurements of the attestation. We leverage this by continuously running the attestation, such that the verifier can safely assume that at all time the prover is running an attestation. To do so, we use a dedicated processor core for the attestation.

Separate processor core for attestation. Figure 2 shows the distribution of the realtime tasks and the attestation on different processor cores. The RealSWATT framework requires at least two processor cores but supports more cores without any changes. A single processor core is selected to execute the attestation. Both the attestation runs and realtime jobs are time-critical. Missing timing deadlines for the normal operation results in device malfunction as realtime properties are not met, timing problems for the attestation make the verification fail, as the verifier assumes the device to be compromised as it does not respond in time.

Continuous attestation. As a dedicated processor core can solely be used to perform the attestation, this allows to introduce continuous attestation, where attestation runs constantly in background. In traditional software-based attestation (SWATT) [46], integrity of the prover code is based on the response between sending the attestation request and receiving the attestation report of the prover, where the attestation request contains a nonce to ensure freshness and preventing replay attacks. As a consequence, the transmission time between verifier and prover needs to be included in the timing assumptions, which makes SWATT impractical for all communications with varying transmission times, e.g., wireless networks or the Internet.

Our continuous attestation relaxes these timing assumptions as the attestation is constantly running so that even though the response is delayed, the verifier can safely assume that the prover has been running the attestation task. In RealSWATT, the verifier sends a new nonce while the prover is not yet finished calculating the attestation report for the verifier (Section 5.2). So, the prover can continue with the next attestation request directly after the last one was finished. We call this attestation method continuous attestation as it removes the gap between attestation runs. In contrast to the communication delay, the time required for the attestation can be determined precisely. In Section 7, we measure the runtime of attestation processes.

In the following, we describe the challenges that emerge when implementing a continuous attestation scheme.

5.1 Design Considerations

While the usage of a separate processor core for remote attestation seems like a straightforward solution, it requires careful design decisions to ensure coverage of a variety of security aspects. Software-based attestation has many strict requirements that have to be fulfilled to reliably verify the prover besides the accurate timing of the responses. It is of utmost importance that an attacker cannot accelerate the attestation run itself. There are multiple ways how an attacker could speedup the attestation. Each of them has to be addressed accordingly.

Parallelization. An attacker can potentially use all processor cores for attestation and ignore realtime-critical jobs, while the genuine attestation can only use a single processor core. If the attacker is able to speed up the execution of the attestation function, the attacker can circumvent the timing checks that are based on hardware limitations and potentially evade the remote attestation.

Therefore, the attestation scheme must be designed such that an attacker cannot benefit from multiple cores. Further, attestation relies on a hashing function. The hashing method must be designed such that it cannot be accelerated by parallelization, i.e., using multiple processor cores. We tackle this challenge by using a Merkle–Damgård construction [36] as this popular hashing method fulfills this requirement. The functionality is shown in Figure 3: The process starts with an initialization vector. The hash is calculated by adding block by block, where in each step the next block is added. In order to add the next block, the previous result is taken as input. This is a strictly sequential process. Hence, the process cannot benefit from parallelization or multiple processor cores [3]. Popular hashing methods using the Merkle–Damgård construction are, for instance SHA-1 and SHA-2 [37].

Optimality of Hash Function Implementation. As security of software-based attestation relies on the computational capability and timing threshold, i.e., the execution speed of the attestation function, it must be ensured that the implementation of the attestation function is optimal and cannot be significantly accelerated. Otherwise, if the attacker is able to generate a valid hash, the saved time can be exploited for malicious activity. RealSWATT addresses this challenge by leveraging built-in hardware modules if available (as is the case for our target architectures) or well-studied hash algorithms. RealSWATT is not limited to a certain hash function: any secure hashing method that fulfills the Merkle-Damgård scheme is suitable for our attestation approach. For example, the popular SHA-2 hash function fulfills this requirement, which we also use in our implementation in Section 6 and case study in Section 7. For platforms without hardware support, we study their security regarding attacks against the hash function in detail in Section 8.3.

Empty memory. As memory that is not covered by the attestation process could be used by an attacker, all executable memory has to be covered by the attestation process. Furthermore, an attacker could compress the data stored on the device in order to free up memory which then can be used to store malicious code. RealSWATT prevents this as its continuous attestation constantly monitors all executable memory. As shown in Section 7, continuous attestation induces strict timing requirements. Deviations, e.g., due to the need for decompression, make the attestation fail.

Offloading. An attacker could also offload attestation work to another device. In our attacker model in Section 4, we describe a remote attacker and excluded the scenario in which a local attacker is able to introduce more computing power into the attested device. However, the attacker could offload the attestation task to another powerful device thereby breaking the attestation scheme. Due to the longer and varying transmission time this would not be detected by the verifier. To tackle this issue, we introduce an IoT gateway that monitors all traffic from and to the network with the attested devices. Such security gateways are a common measure in commercial and industrial networks. But such filters can also be added to routers for small business and home networks. We elaborate on this network architecture in Section 5.3.

In the following, we will explain the attestation scheme and the network architecture in detail.

5.2 Attestation Scheme

As mentioned in Section 5.1, we needed to consider and evaluate several aspects in our design to create a practical software-based attestation approach for realtime embedded devices. Common and more advanced attestation methods like control-flow attestation [2] are not applicable as they either interfere with the runtime (instrumentation), which conflicts with realtime constraints, or require

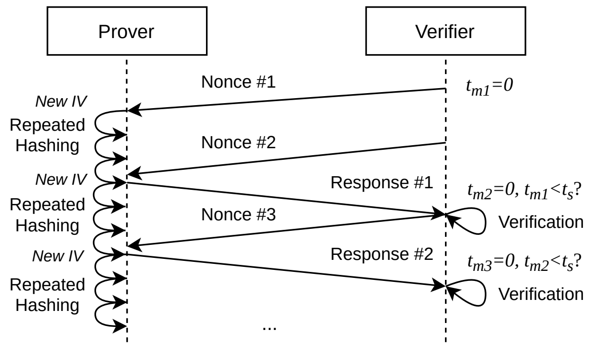

Figure 4: Protocol of the attestation process.

additional hardware like TrustZone. Thus, with RealSWATT, we attest code and data regions of those legacy devices. A device can have multiple partitions containing the executable code and data, including device configuration. We hash those dedicated memory areas based on the protocol shown in Figure 4: The verifier sends a nonce to prevent replay-attacks to the prover. The prover uses this nonce as an initialization vector for the hashing algorithm. Hence, the attacker cannot start the attestation before the nonce is known. Next, the prover calculates the hash of the memory region that has to be attested, e.g., code or data sections. This concept is a common and reliable method for remote attestation [10, 20, 40, 46]. We read all data from the attested partitions and feed it either to the available hardware hashing module or into to the optimized hashing algorithm. Usually all code and data sections are combined and hashed, thus one single hash value represents the code and data integrity of the device. It is also possible to limit the hashing only to certain memory sections. This option, however, should be used with care as it limits the appropriate state representation of the embedded device.

The hash is then repeatedly computed and returned to the verifier. The hash calculation is chained and previous hash results are feed into the next repetition. The verifier measures the time t_m between sending the nonce and receiving the response. If the measured time is below the expected threshold time t_s, i.e, $t_m < t_s$, the device is assumed genuine, otherwise it has potentially been compromised. This process is continuously repeated to ensure that any compromise or malicious modification of the device is being detected. Therefore, it is important that the hashing function has a predictable runtime. If the runtime varies, this allows the attacker to shift tasks to get computation time. This remains undetected, as the verifier has to assume the worst-case runtime. The determination of an appropriate threshold is a key feature for software-based attestation. In Section 7, we measure the execution time for the attestation of a real-world device and describe how the threshold time t_s can be determined.

In a simple attestation protocol there exists a gap between sending the resulting hash and receiving of the next nonce, which consists of the network transmission time and the time of the verifier to send the next nonce after verification. In order to close this time gap between two successive attestation requests, in our solution the verifier sends a second nonce while the prover is processing the previous attestation request (i.e., computing the hash). The second nonce is received and temporarily stored in a queue. This allows the prover to continuously process attestation requests and removes the impact of network delays.

To do so, the verifier has to send the nonces such, that even under worst-case network latency, the next attestation request arrives before the previous attestation run is finished. Sending two nonces without delay allows an adversary to simultaneously compute the hash on other cores. Therefore, it is important to the send the hash just-in-time. Given a time t_{att} to complete the attestation and a network latency t_{rtt}, the verifier has to send the next nonce $t_{att} - \max(t_{rtt})$ to ensure that the next nonce arrives on time. Note, that it is required $t_{att} \ggg t_{rtt}$ to guarantee correct attestation.

The nonce sent by the verifier serves both as a new initialization vector and as a synchronization point. In a scenario with a long-running attestation task with an one-time initialization only, side effects like clock skews between devices would come into play. By continuously sending nonces as new initialization vectors we reliably synchronize verifier and prover.

The verifier checks the interval in which the results are returned from the prover. If results are delayed or missing, a compromise can be assumed and the verifier can react accordingly, e.g., by raising an alarm or rebooting the prover to return to a trustworthy state.

This process, where the prover saves the next nonce in advance, makes our attestation scheme independent from the transmission time between prover and verifier. Even variances in the transmission time do not pose any problems, as long as the transmission time is significantly shorter than the time required for the attestation $t_{att} \ggg t_{rtt}$. It is possible to configure the runtime of the attestation by repeatedly executing the hashing function: the result of the hashing is used as the initialization for the next hashing. So, a long non-parallelizable row of executions is generated. This makes it possible to adapt the duration of the attestation to the actual requirements in transmission time and consider the processing speed of the device. Because a potential attacker cannot offload the computation to an external device and we carefully choose the time intervals for attestation requests (sending the nonces) intercepting the next nonce does not provide any benefit to an attacker. In Section 7, we elaborate on how the attestation time can be configured using a real-world example. If the worst-case transmission time is significantly shorter than the execution time of the attestation, the next nonce can safely arrive at the prover before the previous attestation process finishes. Thus, there are no time gaps between successive attestation requests. We call this approach continuous attestation, which is also a key aspect in enabling practical software-based attestation.

Furthermore, our attestation method has several benefits with regard to existing legacy embedded devices. Our attestation protocol is lightweight with a nonce of 4 B and an attestation report of 32 B. Thus, it causes only a slightly increased network load and is suitable for low-speed IoT networks as discussed in Section 7.3.

Another aspect of RealSWATT is its realtime capability and ease of integration, which we evaluated on real-world devices in Section 7.5. As already shown in Figure 2 we exploit the availability of a second core to handle the attestation process in parallel to realtime operation. While the use of a second core allows to maintain realtime capability it comes with its own set of challenges such as the parallelization of the hash computation by two cores, which we discussed to in Section 5.1 and in Section 8.

5.3 IoT Network Architecture

The usage of a dedicated network architecture allows reliable and secure software-based attestation with varying transmission time as described in Section 4. It consists of several parts: the attested IoT devices, the central configuration and attestation server, and the IoT gateway. This common network architecture allows to prevent offloading attacks, i.e., moving the attestation task to external devices.

Like many IoT devices nowadays, the attested devices communicate over wireless communication (Wi-Fi, IEEE 802.15.4/ZigBee[2], Z-Wave[3]) with the configuration and device management server. There are no strict requirements towards the connection speed, transmission times, or jitter. In the RealSWATT attestation scheme, the verifier is implemented in a central device and configuration management server. This configuration and device management server keeps the current configuration of the IoT devices and performs the verification of the IoT devices. Hence, it is possible to include each device configuration into the attestation and also detect modifications in these configurations, even though the program memory itself is not affected. The IoT gateway monitors external communications and prevents a corrupted device to communicate with external entities to offload the attestation routine and hence break the attestation. Both the central device and configuration management as well as IoT gateways are commonly deployed in real-world networks as previously discussed in Section 4.1.

6 IMPLEMENTATION OF REALSWATT

We implemented RealSWATT on commercial off-the-shelf hardware to show its general applicability. The prover was integrated into FreeRTOS, which is a popular realtime operating system [30]. The verifier was implemented on a Raspberry PI running Linux. The verifier can also be implemented on other devices such as X86, the only requirement is a connection to the IoT network and enough computing resources to handle and measure the attestation requests. We are using raw UDP packets for communication in order to reduce side effects of the network and minimize communication overhead.

We implemented different use-cases for the prover (IoT device) to show its broad applicability: a syringe pump, smart plug, a smoke detector, and a smart light bulb. For the plug, smoke detector and light bulb we have used a framework called ESPeasy[4], which allows to generate alternative firmware images for off-the-shelf IoT devices powered by the ESP32 and includes code for peripherals such as the smoke sensor. Event without this framework, integration into existing off-the-shelf devices is generic and straightforward as described in Section 6.3.

To evaluate the functionality, we integrated all components into a real-world testbed consisting of the typical components of an IoT network. We tested the RealSWATT attestation using a full end-to-end example, consisting of a device being monitored by the

[2]https://zigbeealliance.org/
[3]https://z-wavealliance.org/
[4]https://espeasy.readthedocs.io/en/latest/

verifier. The attested device is then being compromised, which the verifier instantaneously detects.

In the following, we describe the main components of the implementation: (1) Prover and Verifier, (2) Testbed, and (3) Real-world Implementations. Please note that this section only gives a general overview. Implementation details like timing thresholds need to be fine-tuned for typical embedded devices and their networks. We evaluate and provide this details for our testbed in Section 7.

6.1 Prover and Verifier

We use the Espressif ESP32 system-on-chip (SoC) which is a popular component of typical IoT devices, e.g., smart light bulbs and power plugs [24] as it also integrates Bluetooth and Wi-Fi modules.

We have implemented the RealSWATT remote attestation method using the popular FreeRTOS realtime operating system. FreeRTOS can manage multiple processor cores and allows to attach processes to a dedicated core. The scheduler then does not move the attached processes across cores. We dedicate one core to the attestation process. We have implemented the RealSWATT remote attestation method using the popular FreeRTOS realtime operating system on the ESP32. The ESP32 has two Tensilica Xtensa processor cores [25]. Since the attestation is scheduled on a dedicated core, the attestation does not interfere with the realtime operation. Realtime operation as well as attestation are handled by different cores and in parallel.

RealSWATT performs static attestation covering code and data memory of the prover. This is achieved by including the program and configuration data partition into the attestation requests. The hashing is performed using the mbedtls[5] library, which also supports hardware-supported hashing on the ESP32. In order to prevent replay attacks or the usage of pre-computed results, each attestation run is initialized using the nonce provided by the verifier. Continuous attestation is realized using a queue. When the prover receives a UDP packet containing a nonce from the verifier, the nonce is written to a queue of limited size until it is handled by the prover. This way, attestation runs are executed seamlessly after each other. The result of the attestation is returned as a UDP-Packet to the verifier.

The verifier implements the RealSWATT attestation protocol as described in Section 5.2, sending nonces to the prover and handling incoming attestation reports. We provide further implementation details on the prover and verifier in Appendix A.1.

6.2 Testbed

To evaluate RealSWATT, we built a testbed of an IoT network as sketched in Figure 1 consisting of IoT devices, a Wi-Fi access point and a verifier. The IoT devices were implemented on NodeMCU ESP32 developer boards[6], a TP-Link TL-WDR4300 Wi-Fi router[7] running OpenWRT 19.07.7 was used as an IoT gateway and a Raspberry Pi 3+[8] with Linux running the *C++* implemented the verifier. The setup was located in an office environment during workdays with frequent Wi-Fi usage. The Wi-Fi access point provided a separate IoT network in a 2.4 GHz range as the ESP32 is only able

[5]https://tls.mbed.org/
[6]https://joy-it.net/en/products/SBC-NodeMCU-ESP32
[7]https://www.tp-link.com/ch/home-networking/wifi-router/tl-wdr4300/
[8]https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/

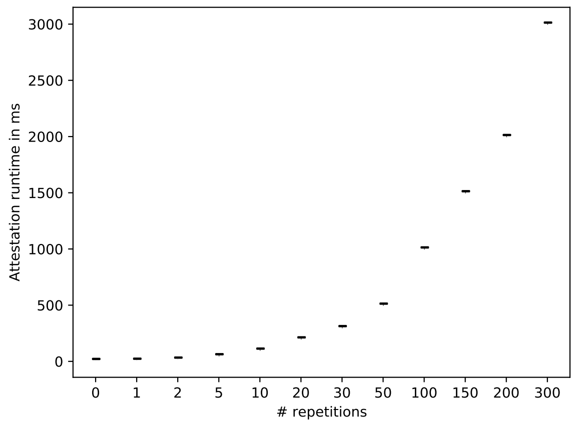

Figure 5: Runtime of the attestation process on the prover with different number of repetitions.

to work within this frequency range. The testbed reflects typical usage in practice, e.g., in hospitals or factories with different interfering Wi-Fi traffic and other wireless devices that influence the communication between the prover and the verifier. The influence of other wireless devices and Wi-Fi traffic and its implications for the attestation is further analyzed in Section 7.

6.3 Integrating RealSWATT

The integration of RealSWATT is straightforward. IoT devices often rely on a realtime operating system (RTOS) [8], which allow to manage and appropriately schedule multiple concurrent tasks on multiple cores. The usage of a RTOS gives standard interfaces and methods to add the attestation service. In order to integrate RealSWATT we have created additional tasks. We added a UDP service for the communication and an attestation task on the dedicated core. It is also possible to integrate RealSWATT into devices with no operating system, so-called bare metal systems. However, the integration will need to be performed much more carefully as one cannot rely on the abstraction and features provided by a realtime operating system.

In the next section we evaluate RealSWATT and show its general applicability. To do so, we perform a case study and integrate RealSWATT into a medical device and an IoT framework.

7 EVALUATION

In this section, we show that RealSWATT attestation is well-suited for real-world IoT setups and can be applied in practice. As described in Section 6, the RealSWATT attestation was deployed on different embedded devices. To show the general applicability of the RealSWATT attestation concept, we investigate its runtime and timing constraints. As elaborated in Section 5, timing is a crucial security factor in software-based attestation. We measure the response times in our exemplary syringe pump example and explain how timing thresholds for the verification of the attestation can be

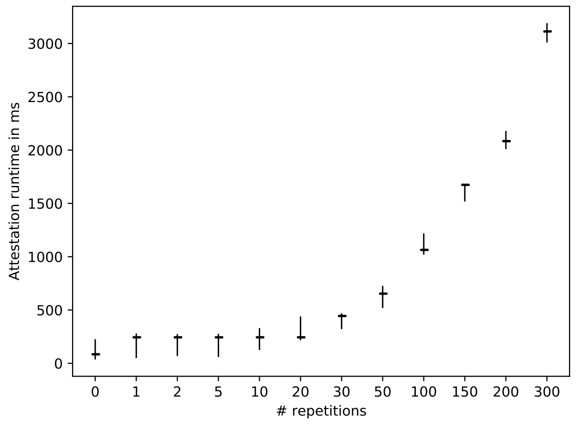

Figure 6: Response time of the prover with different number of repetitions, this includes both the attestation runtime and the overhead due to the Wi-Fi communication.

Table 1: Measurement of the runtime of the attestation with and without the delay due to the Wi-Fi communication. All measurements are taken in ms.

Rounds	Type	Min/Max	Mean	Std. Dev.
0	Direct	7.979/8.14	7.988	0.032
	Network	36.066/226.36	72.207	11.43
1	Direct	9.752/10.265	10.004	0.038
	Network	49.502/279.684	225.76	20.84
2	Direct	19.748/20.276	20.006	0.057
	Network	69.367/274.108	225.96	18.81
5	Direct	50.005/50.005	50.005	0.0
	Network	59.662/276.617	224.61	23.26
10	Direct	100.004/100.004	100.004	0.0
	Network	125.149/329.905	228.32	9.952
20	Direct	200.004/200.004	200.004	0.0
	Network	217.307/439.912	241.46	46.96
30	Direct	300.004/300.004	300.004	0.0
	Network	320.665/469.364	431.11	16.91
50	Direct	500.004/500.004	500.004	0.0
	Network	518.593/726.847	637.27	13.01
100	Direct	1000.004/1000.004	1000.004	0.0
	Network	1019.64/1218.943	1048.91	13.15
150	Direct	1500.005/1500.005	1500.005	0.0
	Network	1517.224/1679.623	1661.35	11.12
200	Direct	2000.004/2000.004	2000.004	0.0
	Network	2009.749/2179.673	2072.29	9.24
300	Direct	3000.004/3000.004	3000.004	0.0
	Network	3009.71/3190.993	3095.92	8.67

determined. We further analyze the overhead induced by the attestation. This covers both additional power consumption caused by the usage of the second processor core as well as the communication overhead for attestation requests and responses.

In a full end-to-end example, we show the functionality of the RealSWATT attestation by performing an attack on a vulnerable device which is detected by the attestation.

7.1 Timing

Timing is a crucial component of the security in software-based attestation. While in traditional software-based attestation the response time to the verifier is the relevant part of the security, in continuous attestation the runtime of the attestation itself is important while the transmission time can be neglected.

We performed a measurement study in our IoT testbed to determine the response times of the prover to the challenge depending on the number of repetitions of the hashing function and the variance of the transmission in the Wi-Fi network. In order to perform a reliable attestation, the number of repetitions of the hashing function has to be chosen such that its runtime dominates the variance of the transmission. The continuous attestation fails if the verifier does not receive the response of the prover before the prover finishes the next attestation request. A too long attestation run increases the time span between two attestation reports of the prover. Hence, the delay between compromise and its detection becomes larger.

Table 1 shows the measured runtime of the attestation function including and without the overhead due to the wireless network using different numbers of repetitions. In Figure 5, the runtime of the attestation process with respect to the number of hashing repetitions is plotted. Since the hashing needs to dominate the variance of the transmission we also plotted the response time with their respective variances in Figure 6 for direct comparison.

We conducted all measurements on the described testbed using the syringe pump implementation and repeated them to cover for any variations. We repeated the Wi-Fi measurements 600 times, the measurements without Wi-Fi were repeated 100 times due to their lower variability. The time including the Wi-Fi transmission was measured on the Raspberry Pi. The runtime without the Wi-Fi overhead was measured on the ESP32 with its internal clock.

As expected, the variance of runtime of the attestation without any communication is minimal. The highest standard deviation in the experiments was 56.83 µs in case of only two repetitions. In all cases with more repetitions, i.e., more than 5, we measured no deviation. This makes the implementation well-suited for software-based attestation, as strict timing limits can be selected.

In comparison, the measurements which include transmission via Wi-Fi have much larger deviations, as Figure 6 shows. For example, in case of ten repetitions, the time until the verifier gets a response from the prover varies between 110 ms and 303 ms. These results clearly show, that such a Wi-Fi setup is inadequate to be directly used for software-based attestation.

The values in Table 1 can be used to find optimal parameters for the attestation. To select suitable parameters, the minimum and maximum values of the attestation can be compared to find the optimal compromise between the delay until a compromise is detected, the required amount of communication, and the required difference between delay due to network communication and the time a single attestation run takes. These parameters are also used to configure the timeout-thresholds for the verifier to detect delays in the attestation.

All attestation runs have about the same execution time and the variance between the executions is negligible. In contrast, the time until the verifier actually receives the attestation response varies widely. A software-based attestation without the RealSWATT continuous attestation approach is not feasible under these circumstances. Based on these results, it is possible to determine the adequate number of repetitions for the given use-case. In case of the syringe pump, we opted for 100 repetitions, yielding in attestation reports in about 1 s intervals. These measurements are also required to configure the verifier to detect malicious behavior resulting in timeouts as the response time varies. The measurements show that RealSWATT is capable to work on IoT devices with wireless communication in practice.

7.2 Power Consumption

The second important aspect for a real-world deployment of RealSWATT is power consumption. Continuous attestation causes constant additional computational tasks for the attestation core, which results in increased power consumption. Often, IoT devices are battery powered. A good example is the smoke detection sensor placed in the corner of the ceiling. Therefore, the power consumption is also a concern of such IoT devices.

Consequently, we have conducted a case study and measured the power consumption of the syringe pump with and without the attestation running. Without attestation, we have measured an average consumption of 46.2 mA and with attestation a slightly increased power consumption of 46.8 mA. So, attestation accounts for an increase in power consumption of 0.6 mA in this case, about 1.3%. A more detailed analysis of the power consumption of our ESP32 evaluation board can be found in Appendix B.2.

7.3 Communication Overhead

Another important aspect is the amount of communication required for the attestation. First, IoT devices often use wireless communication, which is a shared resource with a limited frequency spectrum. With multiple devices communicating via the same channel the network latency as well as package drops increase. Wireless communication is similar to traditional bus communication in that regard. Packets sent simultaneously collide and need to be retransmitted. Each wireless transmission takes a portion of the available bandwidth. For the RealSWATT remote attestation protocol, we only require the transmission of the hash value with 32 B for the attestation reports and 4 B nonce as attestation request. The frequency of transmission can be configured as discussed above, between several attestation requests per second to one every few seconds. These low demands make RealSWATT attestation also suitable to work with low-bandwidth transmission protocols. For example, the popular IoT wireless protocols Zigbee and Z-Wave have transfer speeds ranging from $20\,\text{kbit}\,\text{s}^{-1}$ up to $250\,\text{kbit}\,\text{s}^{-1}$ [51]. Even the lowest transfer speed is sufficient to successfully run RealSWATT attestation with reasonable attestation frequencies.

7.4 Race Conditions

Attestation and realtime operation run in parallel on two dedicated cores. However, the attestation process requires access to the application memory to check for malicious activity. Thus, even when both operations are executed on a separate core, resources still need to be shared, which could lead to a potential race condition. In practice, race conditions between realtime operation and attestation will occur very rarely. Most embedded applications focus on GPIO (General Purpose Input/Output) and thus have little memory interaction. Access to memory can be prioritized depending on the attestation goals and type of realtime application. Most embedded devices in the domain of soft realtime systems will tolerate infrequent deadline misses. Thus, the attestation can potentially be prioritized in these cases. Hard realtime systems are usually strongly tight to the outside world and very GPIO intensive. Memory accesses on hard realtime systems are very short. The acceptable delay for the attestation can be set to a value that will detect malicious activity, but allow short delays caused by the realtime application. This value $t_{rtdelay}$ is distinct for each embedded device and application context and should satisfy: $t_{attack} \geq t_{hash} + t_{rtdelay}$. The delay $t_{rtdelay}$ is determined by the longest operation the realtime application would perform on the flash memory: These are often quick reads of configuration data (e.g., the amount of medicine for injection on the syringe pump).

We have implemented RealSWATT into systems with hard realtime requirements like medical devices (syringe pump) and common IoT devices like smart plugs, see Section 7.5. We evaluated the impact of race conditions in these settings and found that they are highly unlikely and do neither influence the realtime requirements nor the attestation.

7.5 Implementation on Real-World Devices

To show the applicability of our approach to real-world applications and deployments, we applied RealSWATT to a medical device with strict realtime requirements and integrated RealSWATT attestation into an open-source firmware for IoT devices.

The first use-case is a syringe pump [56], a medical device that injects medication into a patient at a defined time interval. Hence, a syringe pump provides critical functionality and has strict realtime requirements. This open-source implementation of a syringe pump has been already used in previous works [2, 39] to show feasibility of control-flow attestation and hotpatching of realtime devices.

In addition, we implemented RealSWATT on top of ESPEasy. ESPEasy is an alternative popular open-source firmware that allows to replace the firmware of existing IoT devices like smart plugs or temperature sensors. ESPEasy supports a wide range of different devices and even extends their functionality. By implementing RealSWATT on ESPEasy we proved that a wide range of legacy IoT devices can be easily secured through our attestation method. We explain real-world details of the attestation based on the syringe pump use-case implemented on the ESP32 in Appendix B.1.

7.6 End-to-End Case Study

To show the full capabilities of RealSWATT, we developed a full end-to-end example: a vulnerable realtime device that is being monitored. The vulnerability is used to compromise the device, this is then detected by the verifier.

To do so, we have integrated a common vulnerability into the syringe pump: an insecure configuration interface. The most common attack vectors of IoT devices are weak, guessable, or hard-coded passwords, and insecure network interfaces and services [43]. In case of ESPEasy, the web interface is only reachable via plain HTTP, hence a passive man-in-the-middle (MitM) attack can easily be used to obtain passwords or authentication tokens [15]. Especially the usage of wireless interfaces further eases MitM attacks. Furthermore, per default the login process of the web interface does not have any rate limiting, allowing efficient brute-force or dictionary attacks, e.g., using hydra[9].

For our proof-of-concept (PoC) we hijack the command interface of our medical device and send a malicious configuration to the unit. This could trigger a buffer overflow and launch a more sophisticated attack, or just manipulate the configuration. In case of the syringe pump, even a configuration change could lead to lethal consequences for the patient as through our attack it is possible to arbitrarily modify the amount of injected medicine. As soon as these changes are applied, the configuration on the *nvs* partition is updated. At latest in the next attestation run the hash value of the *nvs* partition changes, which is sent in the attestation report to the verifier. The verifier determines that the configuration differs from the intended configuration and raises an alarm. For more details on the partitions and the implementation of the attestation see Appendix B.1.

7.7 Conclusion of Evaluation

In our practical evaluation of RealSWATT, we have shown that the attestation runtime can be adjusted by hash repetitions to dominate the variances in network response times of heavily used wireless networks. As a consequence, our proposed attestation method for legacy devices is feasible for wireless IoT networks. In addition, we have measured only a slightly increased power consumption from 46.2 mA to 46.8 mA per hour for attestation. The increase is so small due to the use of the commonly available hardware hashing unit, which reduces the workload of the second core. Furthermore, as most IoT devices already use the wireless communication module at regular basis, the additional power consumption for wireless communication is also minimal. Concluding, RealSWATT is suitable for the application in real-world IoT devices.

8 SECURITY CONSIDERATIONS

The RealSWATT attestation framework uses several new techniques to perform reliable software-based attestation of realtime-critical embedded devices. The security of software-based attestation is based on multiple premises which all have to be fulfilled in order to guarantee the integrity of the attested device. In the following, we discuss the formal criteria for the attacker to stay undetected as well as possible attack scenarios, including mitigations.

Attack Model. To prevail malicious activity, the attacker must perform her attack and hide all traces before the attestation can detect those changes by means of hashing the memory. We define the following variables to analyze diverse attack scenarios: (1) $t_s \widehat{=}$ threshold time, (2) $t_m \widehat{=}$ message response time, (3) $t_{rtt} \widehat{=}$ round trip time, (4) $t_{att} \widehat{=}$ attestation run time (all runs), (5) $t_{hash} \widehat{=}$ hashing run time (1 run), (6) $t_{attack} \widehat{=}$ minimal run-time for attack, and (7) $t_{write} \widehat{=}$ time of the flash to write to the sector.

A successful attack must satisfy $t_{attack} < t_{hash}$, which translates to a scenario, where an attacker completes the attack before a check can be performed (single hash iteration with time t_{hash}).

[9]https://github.com/vanhauser-thc/thc-hydra

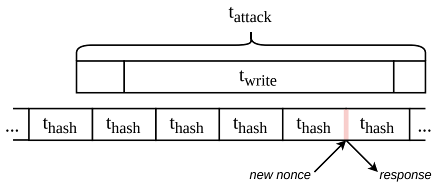

Figure 7: The device is attested in background. The minimum attack duration is longer than the attestation runtime.

Further, an attack is only successful if the attacker is able to either manipulate the program code or configuration data stored in the flash memory (given the threat model defined in Section 4.2). As a result, the attacker needs to rewrite at least one flash page, which takes t_{write} time. The attack scenario is depicted in Figure 7. t_{hash} denotes the runtime of the hashing of the complete memory. Every n repetitions, the result is sent to the verifier in the attestation report. One hash iteration covers all n flash memory blocks b, i.e., $b_0 - b_n$, starting at a random location r. To prevent replay attacks, we integrate a nonce *nonce* provided by the verifier: $hash = \{nonce, b_r...b_n, b_0...b_{r-1}\}$. Multiple hash iterations are concatenated, so that instead of the nonce, the hash result of the previous iteration is used as the starting point: $hash_i = \{hash_{i-1}...\}$.

Attack Requirement. Within the *Attack Model*, we have recapitulated the attestation procedure from the security viewpoint and introduced variables to formally describe the attacker's success conditions. The attacker's goal is to stay undetected. Obviously, the only suitable strategy for the attacker is to perform the attack before it can be detected by the attestation. As described above, this requires $t_{attack} < t_{hash}$. The time of an attack (t_{attack}) depends on the concrete attack scenario and cannot be exactly determined. However, a lower bound of t_{attack} can be given due to hardware limitations such as the flash write time (t_{write}). We will elaborate on this in the following.

8.1 Hardware Restrictions

Embedded devices have hardware restrictions such as fixed times to write and read memory that cannot be changed by the adversary (see Section 4.2). Since the attacker is not able to replace physical hardware, the attacker is bound by the hardware; especially, slow write operations to flash memory. This allows to derive a lower bound for attacks (t_{attack}).

In case the device features a hardware hashing module, the attacker has to deal with fast hashing, i.e., short intervals in which the memory is attested. In addition, embedded devices provide flash memory as their main memory and embedded devices such as the ESP32 typically rely on external flash. Flash memory has to be written sector-wise and flash typically uses a sector size of 4 kB. To erase one sector of a commonly used and quick flash chip takes about 50 ms. A similar fast flash chip is also deployed on our ESP32 boards [22]. Other flash chips require even 100 ms to erase a sector [21]. Since a successful attack requires at least one write, the minimal duration of any attack is t_{write}, so $t_{attack} \geq t_{write}$. The availability of hashing modules and flash speeds differ. Hence, hardware restrictions, i.e. t_{write}, need to be considered for every device. In the specific case of our ESP32 board a sector write to the flash requires 50 ms [22] but a single hash iteration only takes 10 ms (c.f., Table 1). As such, memory manipulation attacks are detected by RealSWATT for all modern embedded devices shipped with a hashing module.

8.2 Common Attack Scenarios

Compressing data on the device. A compression attack is a typical technique to undermine software-based attestation [46]. The idea is to use compression mechanisms on the attested device to free up memory that is not covered by the attestation and can be used to store software that remains undetected from the attestation. In the worst-case, the attacker decompresses malicious data on-the-fly at attestation time. Since RealSWATT targets multicore processor architectures, the attacker has the capability to perform the attestation and decompression in parallel. However, a compression attack requires rewriting parts of the memory. However, these write accesses take more time than the interval t_{hash}, i.e., $t_{write} > t_{hash}$ still holds. Hence, we conclude that it is not possible to rewrite parts of the memory by means of a compressed version without detection.

In addition, RealSWATT deploys further techniques to prevent compression attacks. Since it is challenging to compress random data [46, 59], we fill free memory with random data provided and controlled by the verifier. The attestation reports include the random data thereby allowing the verifier to easily detect modifications. Furthermore, due to continuous attestation, the device is attested frequently (every 10ms as shown in Table 1). Thus, the attacker can only guess whether a data segment is currently being attested.

TOCTOU. It is well-known that existing (remote) attestation schemes are susceptible to Time-of-Check Time-of-Use (TOCTOU) attack [53]. An adversary capable of restoring the memory in a given time frame, before the next attestation, will remain undetected. However, in contrast to known attestation schemes RealSWATT significantly reduces the adversary's success by leveraging the following:

(1) The attestation continuously runs in the background, checking the complete memory in regular intervals while the device is operating normally. (2) The attestation starts from an arbitrary memory position (derived from the nonce). The attacker can neither predict nor influence this starting point. Halting the attestation core or process will be detected by timing thresholds.

Existing software-based attestation techniques like SWATT [46] interrupt normal operation for attestation. This is not an option for realtime systems, as these interruptions conflict with realtime requirements. As RealSWATT attestation runs in the background, the TOCTOU problem is limited to the interval in which each memory area is attested. This time span t_{hash} is very short due to the optimized implementation (c.f., Section 8.1), so that manipulations of the flash memory can be detected. This also implies that no pseudorandom memory traversal is required, as used in SWATT [46].

Memory Manipulation. Another strategy to avoid detection is to use, create, or find unmonitored memory, e.g., the RAM memory. The attacker could also try to change the memory layout by manipulating the partition table. In general, moving malicious code to RAM is not feasible. Code usually resides in flash memory and RAM is therefore typically marked as non-executable [27]. Furthermore, manipulating the partition table is also a highly challenging task [29]. In the case of the ESP32 several preconditions must be met: first, SPI Dangerous Write must be enabled. Second, an entire flash page would need to be rewritten. Third, partition changes and rewrites are typically implemented by OTA (Over-the-Air Update) [28], which inevitably causes a reboot. Thus, manipulating the partition table is unfeasible for an attacker. Note that runtime reconfiguration of memory permissions is also not possible because the MMU (memory management unit) is privileged and set during boot.

8.3 Attacks on Attestation Protocol

There are two attack strategies to undermine the underlying security assumptions of the protocol used by RealSWATT: (1) using multiple cores, and (2) optimizing the implementation of the attestation loop. We discuss how RealSWATT mitigates both of them.

Using multiple cores to break attestation. The attacker can use the full computing power of the device, while RealSWATT has to obey strict limitations due to realtime-critical jobs. The attacker could ignore realtime jobs and try to use more than a single core for hash calculations. In case hash calculation can be accelerated through parallelization, the attacker gains an attack window. However, the attestation process still cannot be accelerated using multiple processor cores since we use a Merkle–Damgård construction for hashing (e.g., SHA-1, SHA-2). This method sequentially hashes each block and requires the previous block as input. Thus, an attacker cannot use multiple processor cores to parallelize this process. The Merkle–Damgård hash construction is strictly sequential.

Optimizing implementation of attestation loop. Software-based attestation (SWATT) [46] is based on the assumption that the implementation of the attestation algorithm cannot be accelerated by an attacker. Thus, an optimal implementation of the attestation function and its main component, the hash function, is required. Algorithms are complex and there are alternative ways to implement the same functionality. Castelluccia et al. demonstrated that SWATT [46] can be undermined using a faster implementation of the attestation function [12]. As described in Section 3, this gives an attacker a time slot, where the attestation does not run although the verifier assumes the prover is currently running the attestation function. RealSWATT addresses this issue using standard hash functions like SHA-256, for which optimized implementations or even hardware acceleration exist, as elaborated in Section 5.1. As these hash functions are widely and have been well-studied in the past [38], it is unlikely that there exist implementations that significantly improve the execution speed; especially compared to hardware-assisted hash functions as we use in RealSWATT. Since we use global and standardized hashing algorithms (SHA-2) [38], highly optimized software implementations are available in case the target platform features no hardware hashing module.

8.4 Network-based Attacks

Connected devices are inherently prune to network-based attacks. Common practical issues in remote attestation, such as offloading the attestation process, have been considered in the design as discussed in Section 5. In the following, we explain network-based attacks remote attestation faces and how these issues have been addressed.

Shifting the attestation to another device. The IoT gateway prevents external communication. Hence, it is neither possible to leak attestation data (e.g., the nonce) to a remote party nor to receive attestation reports from outside the network. All IoT devices in the network are covered by the attestation, so there are no free resources to perform an attestation on behalf of another device.

Delaying communication. The attacker can delay the communication between the prover and the verifier. As some variation in the transmission time is normal, this would remain undetected to the verifier. While this would be a critical problem in traditional software-based attestation, RealSWATT attestation is designed to work with delays as imposed by communication for example using standard Wi-Fi networks. The concept of continuous attestation ensures that even if communication is delayed, the attestation process remains unaffected. Continuous attestation sends a second nonce before the previous attestation process has finished. The verifier times the second nonce such that it should arrive even under the worst expected network latency (see Section 5.1). All attestation runs on the prover are continuously running and have constant attestation time t_{att}. If communication is intentionally delayed by the attacker, the verifier will detect the shift in communication delay and the attestation time will exceed the threshold t_s.

9 RELATED WORK

There exist different approaches to implement remote attestation as we review in the following:

Hardware-based. Trusted hardware that can provide trusted execution environments (TEE) allows to securely execute computations and store information on an unstrusted or potentially compromised system. Examples for platforms with TEEs are Intel SGX [16] and Arm TrustZone ARM Limited. Trusted computing technology is integrated into many sophisticated off-the-shelf processors as they are used in servers, personal computers, and smartphones. In contrast, less powerful processors as they are used in embedded and IoT devices do not feature such TEEs due to cost reasons. TEEs can be used to reliably monitor the data and execution in the normal world from within the secure world. Hence, the monitoring cannot be influenced. For example Abera et al. proposed C-FLAT, a framework to remotely verify the control flow of applications running in the normal world using the TrustZone of an Arm processor [2]. However, these trusted computing modules are often not available on legacy embedded devices due to cost reasons.

Software-based. Software-based attestation (SWATT) works under the assumption that the attacker cannot change the target device and its computing capabilities. The attacker is not able to introduce further computing resources, hence the execution time of specific operations is bound. This circumstance allows to measure the response time of the prover: if it takes longer than expected, the system is likely compromised. The attacker cannot forge these results as this would require further operations, so that the response cannot be sent in time [46]. However, software-based remote attestation works upon strict timing assumptions on the response of the prover and the verifier. This induces many requirements upon the implementation of attestation logic and communication [7, 46]. As software-based remote attestation requires no customized hardware, it is commonly used in sensor networks [53].

Hybrid. Hybrid attestation schemes describe a hardware-/software co-design, that performs remote attestation in software supported by custom hardware extensions. These hardware extensions allow to securely store keys or track the execution of commands on the processor. This gives a root of trust, so that the attested device cannot be emulated or replaced by the adversary. There exist various proposals for attestation to use customized hardware to provide remote attestation functionality [10, 20, 34, 40]. SMART is the first attestation scheme that builds upon simple customized hardware [20]. While the attestation is performed in software, the key is protected using custom hardware functionality so that it is not leaked during attestation.

While SMART uses a small extension to protect the keys used for remote attestation, it lacks more sophisticated features like the ability to update the attestation code. Furthermore, SMART requires the attestation to run atomically, which is a major drawback in many application scenarios where the attested device may not be interrupted by the attestation process, e.g., realtime systems. TrustLite does not have these limitations, as it allows to isolate different software modules with hardware modifications of the Memory Protection Unit (MPU) and the exception engine of the processor [34]. TyTan builds upon TrustLite and extends it such that it is able to run applications with realtime requirements [10].

VRASED is a formally verified hardware/software co-design for remote attestation [40] and allows to verify the state of device memory. It has been extended to also verify reset, erasure and update of devices [41] and also attest that code has actually been executed [42].

While these hybrid attestation schemes have many advantages, the required custom hardware extensions are not available on already manufactured legacy devices. The creation of customized hardware is complex and adds significant costs to the manufacturing. Embedded systems often use off-the-shelf microprocessors with generic hardware modules. For the implementation of a hybrid attestation scheme the device itself needs to be replaced. Thus, hybrid attestation is no viable option for legacy devices.

10 CONCLUSION AND SUMMARY

In this paper, we presented RealSWATT, a purely software-based remote attestation framework that allows to attest even systems with realtime constraints. RealSWATT is designed to work on legacy devices in real-world IoT scenarios. We achieve this by introducing continuous attestation, which constantly performs attestation in the background without interfering with normal operation of the system by using a dedicated processor core. In the evaluation, we show that RealSWATT attestation actually has predictable and constant runtime, a mandatory requirement for software-based attestation. We implemented RealSWATT into a syringe pump, a critical medical device with realtime requirements. In an end-to-end experiment, we detected a compromise of the syringe pump via an insecure configuration interface. To show practicability we integrated RealSWATT into ESPEasy, an open-source framework to use on commercial off-the-shelf IoT devices.

APPENDIX

In the appendix, we give implementation details on the prover, verifier and test bed. We provide additional information on the integration of RealSWATT into the syringe pump. Furthermore, we explain the measurements of the power consumption of the syringe pump with and without the attestation to quantify the increase of the power consumption due to the attestation.

A IMPLEMENTATION DETAILS

A.1 Prover and Verifier

The attestation is implemented as two separate tasks, one receiving the attestation requests from the verifier, and one task to perform the actual attestation. Both tasks are pinned to the dedicated and previously unused processor core to ensure no side effects between the attestation and the realtime operation of the attested device. We verify that the attestation runs continuously without distractions from the realtime operation. RealSWATT attestation has the possibility to configure the runtime of the attestation. The runtime needs to balance between delay (compromise detection of the attested device) and communication overhead. This is done by configuring the number of repetitions of the hashing function. More repetitions invoke a longer run time. At the end of each attestation run, the result is sent to the verifier which then checks its validity.

When integrating RealSWATT attestation into a new device, the runtime of the attestation function has to be determined and the number of repetitions has to be configured. In Section 7 we perform a detailed measurement of runtime and communication overhead on the implementation of RealSWATT.

As mentioned the verifier receives the attestation reports and checks its validity. The verifier implements the RealSWATT attestation protocol and sends two nonces to the prover. Each nonce triggers an attestation run. It is the verifier's responsibility to time the transmission of these nonces. The design of the protocol is explained in Section 5.2. In Section 7, we elaborate on how to correctly time these message intervals and determine the thresholds for the attestation.

There are two available implementations for the verifier. First, we used *Python*, later we opted for *C++*. Since *Python* is an interpreter-based programming language the *Python* implementation of the verifier can be used without adjustment across a wide range of devices. The only requirement is that a *Python* interpreter for the device is available. However we assumed a worse network response time than with a native *C* or *C++* implementation. In order to check the influence of the programming language we also implemented the verifier as a native *C++* application. Contrary to our expectations, the programming language had little to no impact on the measured network response times.

B REAL-WORLD DEVICES

B.1 Syringe Pump

In the following, we explain real-world details of the attestation based on the syringe pump use-case implemented on the ESP32. The ESP32 allows for the custom creation of partitions. A developer can define memory sections on the chip in a data structure called partition table [29]. The partition structure depends on the implemented application and the required functionalities. For example if an update mechanism such as over-the-air (OTA) update [28] is used, additional partitions are required. A simple application with no OTA update functionality consists of the following three partitions:

Listing 1: Default ESP32 Partition without OTA [29]

```
# ESP-IDF Partition Table
# Name, Type, SubType, Offset, Size,Flags
nvs, data, nvs, 0x9000, 0x6000,
phy_init, data, phy, 0xf000, 0x1000,
factory, app, factory, 0x10000, 1M,
```

The partitions are by default (1) *factory*, (2) *phy_init* and (3) *nvs*. Partition (1) *factory* contains the application code, i.e., the executable. The second partition (2) *phy_init* contains data required for the physical initialization process of the device. The third partition (3) *nvs* stores the configuration of the actual application.

The syringe pump is implemented with this default partition mapping. The code is saved on the *factory* partition and the configuration data is included in the *nvs* partition. The syringe pump comes with multiple internal and external configuration options. Internal configurations cover physical characteristics of the syringe pump such as the length of the syringe barrel:

Listing 2: Internal configuration of the syringe pump

```
typedef struct {
uint16_t syringe_volume_ml;
uint16_t syringe_barrel_length_mm;
float threaded_rod_pitch;
....
} internal_settings;
```

The internal configuration is required to transpose configured information such as the amount of medicine into the precise amount of rotation steps of the stepper motor driving the threaded rod of the pump. The syringe pump has also its usual medical settings available to the medical personal such as injection intervals and amount of medicine:

Listing 3: Medical configuration of the syringe pump

```
typedef struct {
    uint32_t injections_ms;
    uint16_t dosage_ml;
    uint8_t bolus_step_index;
} medical_settings;
```

In order to attest the syringe pump the data from all three partitions *factory*, *phy_init*, and *nvs* is read and concatenated. Then, we append the nonce and feed this data into the hardware hashing module of the ESP32. The resulting hash value is then repeatedly rehashed and sent to the verifier. Since the verifier knows the original syringe pump code as well as the physical initialization parameters and the configured options, it can verify the correct state of the syringe pump. The verifier can either integrate a device configuration manager or be notified by an external one about legitimate configuration changes. In our use-case we have integrated this functionality into the verifier.

The next nonce is sent by the verifier such that it arrives at the prover just before the end of the expected attestation time, even with worst-case network latency. As explained in Section 5.2, we have chosen 100 repetitions, such that $t_{att} = 1000ms \ggg t_{rtt}$ as Table 1 shows. The verifier is configured to send the second nonce 750 ms after the previous nonce.

B.2 Power Consumption

In order to evaluate the measured power consumption, we have checked the corresponding data sheet [26] of our ESP32 evaluation board. The data sheet provides the expected power consumption in respect to operating mode of the chip for both the processor and wireless module.

The power consumption is dominated by the wireless communication module. Its power consumption varies between transmission mode. It ranges from 95 mA and 130 mA to receive and transmit via Bluetooth; up to 100 mA to receive and 240 mA to transmit via Wi-Fi IEEE 802.11b/g/n. However, this power consumption is only present during the regular send and receive intervals and thus needs to be treated as a peak power consumption.

The power consumption of the CPU is depending on its operating frequency and overall usage of the CPU. For the testing, we have configured the syringe pump to operate at the full CPU clock of 240 MHz. This CPU frequency has the highest difference between no CPU usage (30 mA) and full CPU usage (68 mA). The naive assumption would be at least a 50 percent usage of the CPU (full utilization of the second core) for attestation, resulting in a significantly higher power consumption. However, our implementation used the integrated hardware hashing unit, which is more power-efficient compared to a software-based calculation of the hashes.

We furthermore observed that the additional network traffic for the attestation is negligible for devices which already communicate on a regular basis. The syringe pump in our case study provides a remote command interface. Thus, the wireless communication module of this device is already in use. So sending and receiving attestation messages only slightly increases the overall power consumption.

ACKNOWLEDGMENTS

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—SFB 1119—236615297 within project S2. This work was supported by the DFG Priority Program SPP 2253 Nano Security (Project RAINCOAT—Number: 440059533). Part of this work was conducted within Huawei Open Lab for Sustainable Security and Safety (OpenS3Lab) at Technical University Darmstadt. We thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009). https://doi.org/10.1145/1609956.1609960

[2] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* ACM. https://doi.org/10.1145/2976749.2978358

[3] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. 2008. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th International Symposium on High-Performance Distributed Computing (HPDC-17 2008).* ACM. https://doi.org/10.1145/1383422.1383443

[4] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *IEEE Symposium on Security and Privacy (SP).* IEEE. https://doi.org/10.1109/SP.2019.00013

[5] Anna-senpai. 2017. GitHub - Mirai Source Code. Retrieved 2021-04-09 from https://github.com/jgamblin/Mirai-Source-Code

[6] ARM Limited. 2009. Security technology building a secure system using trustzone technology (white paper).

[7] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. 2013. A security framework for the analysis and design of software attestation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13.* ACM. https://doi.org/10.1145/2508859.2516650

[8] AspenCore. 2019. 2019 Embedded Markets Study. Retrieved 2021-05-07 from https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[9] Kaitlin Boeckl, Michael Fagan, William Fisher, Naomi Lefkovitz, Katerina Megas, Ellen Nadeau, Benjamin Piccarreta, Danna O'Rourke, and Karen Scarfone. 2019. Considerations for Managing Internet of Things (IoT) Cybersecurity and Privacy Risks. https://doi.org/10.6028/NIST.IR.8228

[10] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. 2015. TyTAN: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference.* ACM. https://doi.org/10.1145/2744769.2744922

[11] Ferdinand Brasser, Kasper Bonne Rasmussen, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. Remote attestation for low-end embedded devices: the prover's perspective. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016.* ACM. https://doi.org/10.1145/2897937.2898083

[12] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009.* ACM. https://doi.org/10.1145/1653662.1653711

[13] Chien-Ying Chen, Monowar Hasan, and Sibin Mohan. 2018. Securing Real-Time Internet-of-Things. *Sensors* 18, 12 (2018). https://doi.org/10.3390/s18124356

[14] Boheung Chung, Jeongyeo Kim, and Youngsung Jeon. 2016. On-demand security configuration for IoT devices. In *2016 International Conference on Information and Communication Technology Convergence (ICTC).* IEEE.

[15] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. 2016. A Survey of Man In The Middle Attacks. *IEEE Communications Surveys & Tutorials* 18, 3 (2016). https://doi.org/10.1109/COMST.2016.2548426

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016). http://eprint.iacr.org/2016/086

[17] Sanjeev Das, Wei Zhang, and Yang Liu. 2016. A fine-grained control flow integrity approach against runtime memory attacks for embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 11 (2016). https://doi.org/10.1109/TVLSI.2016.2548561

[18] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *Comput. Surveys* 43, 4 (2011). https://doi.org/10.1145/1978802.1978814

[19] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. 2018. LiteHAX: lightweight hardware-assisted attestation of program execution. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018.* ACM. https://doi.org/10.1145/3240765.3240821

[20] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012.* The Internet Society. https://www.ndss-symposium.org/ndss2012/smart-secure-and-minimal-architecture-establishing-dynamic-root-trust

[21] elm-tech. 2021. GD25Q32 Datasheet. Retrieved 2021-07-26 from https://chipmaster.pro/wp-content/uploads/2019/04/GD25Q32.pdf

[22] elm-tech. 2021. GD25Q32C Datasheet. Retrieved 2021-07-26 from http://www.elm-tech.com/en/products/spi-flash-memory/gd25q32/gd25q32.pdf

[23] Mohammed Faisal Elrawy, Ali Ismail Awad, and Hesham F. A. Hamed. 2018. Intrusion detection systems for IoT-based smart environments: a survey. *Journal of Cloud Computing* 7 (2018). https://doi.org/10.1186/s13677-018-0123-6

[24] Espressif Systems. 2018. Espressif Achieves the 100-Million Target for IoT Chip Shipments. online. Retrieved 2021-05-07 from https://www.espressif.com/en/news/Espressif_Achieves_the_Hundredmillion_Target_for_IoT_Chip_Shipments

[25] Espressif Systems. 2020. ESP32 Technical Reference Manual. online. Retrieved 2021-05-07 from https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

[26] Espressif Systems. 2021. ESP32 Series Datasheet. online. Retrieved 2021-05-03 from https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[27] Espressif Systems. 2021. Memory Capabilities. Retrieved 2021-07-06 from https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/mem_alloc.html
[28] Espressif Systems. 2021. Over The Air Updates (OTA). online. Retrieved 2021-05-06 from https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html
[29] Espressif Systems. 2021. Partition Tables. online. Retrieved 2021-05-06 from https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/partition-tables.html
[30] FreeRTOS. 2021. GitHub - FreeRTOS. Retrieved 2021-05-07 from https://github.com/FreeRTOS/FreeRTOS/tree/master
[31] David Geer. 2005. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer* 38, 5 (2005). https://doi.org/10.1109/MC.2005.160
[32] Dan Goodin. 2016. Brace yourselves—source code powering potent IoT DDoSes just went public. Retrieved 2021-04-09 from http://arstechnica.com/security/2016/10/brace-yourselves-source-code-powering-potent-iot-ddoses-just-went-public/
[33] ITU-T. 2012. *Overview of the Internet of things.* Recommendation Y.2060. International Telecommunication Union.
[34] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: a security architecture for tiny embedded devices. In *Ninth Eurosys Conference 2014, EuroSys 2014*. ACM. https://doi.org/10.1145/2592798.2592824
[35] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak N. Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP.2010.34
[36] Ralph Charles Merkle. 1979. *Secrecy, authentication, and public key systems.* Stanford University.
[37] National Institute of Standards and Technology. 2008. Secure Hash Standard (FIPS 180-3). Retrieved 2021-09-14 from https://csrc.nist.gov/publications/detail/fips/180/3/archive/2008-10-31
[38] National Institute of Standards and Technology. 2015. Secure Hash Standard (SHS). Retrieved 2021-09-12 from https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf
[39] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/hera-hotpatching-of-embedded-real-time-applications/
[40] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium, USENIX Security 2019.* USENIX Association.
[41] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2019. PURE: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in low-End Embedded Systems. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019.* ACM. https://doi.org/10.1109/ICCAD45719.2019.8942118
[42] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2020. APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise. In *29th USENIX Security Symposium, USENIX Security 2020.* USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/nunes
[43] OWASP. 2018. Internet of Things (IoT) Top 10 2018. Retrieved 2021-05-06 from https://owasp.org/www-pdf-archive/OWASP-IoT-Top-10-2018-final.pdf
[44] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. 2017. An Experimental Security Analysis of an Industrial Robot Controller. In *IEEE Symposium on Security and Privacy (SP).* IEEE. https://doi.org/10.1109/SP.2017.20
[45] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. 2015. The shift to multicores in real-time and safety-critical systems. In *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015.* IEEE. https://doi.org/10.1109/CODESISSS.2015.7331385
[46] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. 2004. SWATT: SoftWare-based ATTestation for Embedded Devices. In *2004 IEEE Symposium on Security and Privacy (S&P 2004).* IEEE Computer Society. https://doi.org/10.1109/SECPRI.2004.1301329
[47] Weidong Shi, Hsien-Hsin S. Lee, Laura Falk, and Mrinmoy Ghosh. 2006. An Integrated Framework for Dependable and Revivable Architectures Using Multicore Processors. (2006). https://doi.org/10.1109/ISCA.2006.8
[48] Devu Manikantan Shila, Penghe Geng, and Teems Lovett. 2016. I can detect you: Using intrusion checkers to resist malicious firmware attacks. In *2016 IEEE Symposium on Technologies for Homeland Security (HST).* IEEE.
[49] Kang G. Shin and Parameswaran Ramanathan. 1994. Real-Time Computing: A New Discipline of Computer Science and Engineering. In *Proceedings of IEEE, Special Issue on Real-Time Systems.* IEEE.
[50] Spectra Industrie-PC und Automation. 2021. Embedded Configuration Manager (ECM). Retrieved 2021-07-13 from https://www.spectra.de/cms/splash/embedded-configuration-manager/
[51] Embedded Staff. 2006. Catching the Z-Wave. Retrieved 2021-04-28 from https://www.embedded.com/catching-the-z-wave/
[52] John A. Stankovic and Raj Rajkumar. 2004. Real-Time Operating Systems. *Real Time Systems* 28, 2-3 (2004). https://doi.org/10.1023/B:TIME.0000045319.20260.73
[53] Rodrigo Vieira Steiner and Emil Lupu. 2016. Attestation in Wireless Sensor Networks: A Survey. *ACM Computing Surveys (CSUR)* 49, 3 (2016). https://doi.org/10.1145/2988546
[54] Keith Stouffer, Victoria Pillitteri, Suzanne Lightman, Marshall Abrams, and Adam Hahn. 2015. Guide to Industrial Control Systems (ICS) Security. https://doi.org/10.6028/NIST.SP.800-82r2
[55] Victor van der Veen, Dennis Andriesse, Enes Göktas, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM. https://doi.org/10.1145/2810103.2813673
[56] Bas Wijnen, Emily J Hunt, Gerald C Anzalone, and Joshua M Pearce. 2014. Open-source syringe pump library. *PloS one* 9, 9 (2014).
[57] Reinhard Wilhelm and Jan Reineke. 2012. Embedded systems: Many cores - Many problems. In *7th IEEE International Symposium on Industrial Embedded Systems, SIES 2012.* IEEE. https://doi.org/10.1109/SIES.2012.6356583
[58] Ally Winning. 2019. Number of automotive ECUs continues to rise. online. Retrieved 2021-05-03 from https://www.eenewsautomotive.com/news/number-automotive-ecus-continues-rise
[59] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. 2007. Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In *26th IEEE Symposium on Reliable Distributed Systems (SRDS 2007).* IEEE Computer Society. https://doi.org/10.1109/SRDS.2007.31
[60] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. 2017. Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System. In *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation (IoTDI).* ACM. https://doi.org/10.1145/3054977.3054999
[61] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013.* IEEE Computer Society. https://doi.org/10.1109/RTAS.2013.6531076
[62] Man-Ki Yoon, Lui Sha, Sibin Mohan, and Jaesik Choi. 2015. Memory heat map: anomaly detection in real-time embedded systems using memory behavior. In *Proceedings of the 52nd Annual Design Automation Conference.* ACM. https://doi.org/10.1145/2744769.2744869
[63] Bruno Bogaz Zarpelão, Rodrigo Sanches Miani, Cláudio Toshio Kawakani, and Sean Carlisto de Alvarenga. 2017. A survey of intrusion detection in Internet of Things. *Journal of Network and Computer Applications* 84 (2017). https://doi.org/10.1016/j.jnca.2017.02.009
[64] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. 2017. ATRIUM: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017.* IEEE. https://doi.org/10.1109/ICCAD.2017.8203803
[65] Kim Zetter. 2015. A Cyberattack Has Caused Confirmed Physical Damage for the Second Time Ever. Retrieved 2021-04-09 from https://www.wired.com/2015/01/german-steel-mill-hack-destruction/