Bachelor Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Aggregatable Remote Attestation for IoT

Vasil Alistarov
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr. Alexandra Dmitrienko
First Reviewer

Prof. Dr. Alexandra Dmitrienko
First Advisor

BSc. Ala Eddine Ben Yahya
Co-advisor

Submission
27. December 2020

www.uni-wuerzburg.de

Abstract

With the recent spike of Internet of Things (IoT) and "smart" devices, there has also been an increase in the amount of attacks on IoT networks. Wide-reaching attacks such as the one from the Mirai botnet in 2016 [1] show how crucial it is to know that a device can be trusted before initiating communication.

Remote Attestation (RA) is a proven method for asserting that a device is in a benign state. It is a challenge-response process between two parties, where the first checks the trustworthiness of the second. However, it is characterisable with low scalability – a critical issue in the IoT sector. In our work, we model a new RA protocoll, called *Aggregatable Remote Attestation*, which would allow a device to process multiple RA challenges simultaneously. We base it on the already existing SIMPLE architecture [2] and implement it as a Proof-of-Concept (PoC) by modifying the code of the Security MicroVisor (SμV) – the core component of SIMPLE.

We evaluate our work in terms of security and performance and show that it greatly outperforms the underlying SIMPLE. We discuss the relevance of our design in relation to the IoT sphere and denote a small set of potential topics for future work and research.

Zusammenfassung

In den vergangenen Jahren wurde ein rapider Anstieg der Anzahl der Internet of Things (IoT)-Geräte beobachtet, welcher sich auf die rasante Digitalisierung zurückführen lässt. Allerdings hat dies ein unerwünschtes Seiteneffekt mit sich gebracht: die Frequenz der Angriffe auf IoT-Netzwerke ist ebenso gestiegen und diese werden immer umfangreicher. Es ist damit klar ersichtlich, wie wichtig es ist, die Sicherheit und den vertrauenswürdigen Zustand einer IoT-Entität überprüfen beziehungsweise nachweisen zu können.

Unter den berühmtesten Mitteln zur Feststellung der Vertrauenswürdigkeit eines Geräts ist die sogenannte Remote Attestation (RA) [3] – ein Challenge-Response-Verfahren zwischen zwei Perteien, Verifier (VRF) und Prover (PRV) genannt, und den gutartigen Zustand des Letzteren zu überprüfen. Dies ist insbesondere bei kleinen IoT-Geräten relevant, die oft keine sonstigen Sicherheitsmechanismen besitzen [4]. Unglücklicherweise lässt sich die RA, als ein eins-zu-eins-Prozess, relativ schlecht skalieren, was ein enormer Nachteil im IoT-Sektor darstellt. In unserer Arbeit konzipieren wir eine sogenannte *Aggregatable Remote Attestation*, die es einer PRV-Entität ermöglicht, mehreren RA-Herausforderungen gleichzeitig zu verarbeiten.

Diese Aggregatable RA bauen wir auf SIMPLE [2], einem bereits existierenden RA-Protokoll, auf. Wir modifizieren dieses und implementieren unser Konzept als Proof-of-Concept (PoC) mithilfe des Quellcodes vom Security MicroVisor (SμV), dem Kernelement von SIMPLE. Anschließend evaluieren wir unser Design bezüglich Performanz sowie Sicherheit und zeigen, dass diesen keine weiteren schwerwiegenden Sicherheitslücken im RA-Prozess schafft. Wir diskutieren über die Relevanz unseres Designs in Relation zur IoT und erörtern dazu auch einige eventuelle Themen für zukünftige Forschung.

Contents

1. Introduction

In the past years, a drastic increase of the amount of Internet of Things (IoT) devices has been observed. Not only has there been a spike in the total number of "smart" devices used and produced, but also many formerly analogue systems have been updated and digitalised, making them part of the IoT world. Now, its elements can be found in all spheres of our lives, reaching from the homes to as far as the medicine and the industry [5].

However, this also entices numerous malicious adversaries willing to infiltrate IoT networks, by widening their attack possibilities. Indeed, with the expansion of the IoT sector, there has also been an upsurge of attacks that have occasionally proven to be remarkably costly or to affect devices on a huge scale [1, 6, 7]. Hence, one must always strive to assert the security and benignness of any entities they communicate or interact with. This is a critical issue especially in the area of resource-constrained IoT devices, as those often do not come with any built-in security mechanisms [4]. Furthermore, classical applications and approaches are unsuitable there due to the limitations in terms of memory or computational power.

Among the best-known techniques for a (small) "smart" device to show its trustworthiness is the Remote Attestation (RA) [3]. It is an one-to-one process between a trusted and an untrusted entity, called Verifier (VRF) and Prover (PRV), respectively. RA follows a challenge-response model, with VRF sending a challenge with specific parameters, called Attestation Request (AR), such that PRV can only answer correctly if and only if it is currently in a benign state (i.e., not infected by malware) [3]. Research has already been conducted in the sphere of RA; indeed, multiple architectures have already been created, and there is vigorous ongoing research in that sphere.

Still, RA in its classical form does exhibit remarkably low scalability. That is a crucial issue in the IoT sphere, where there may be hundreds or thousands of devices communicating. A device may, for instance, need to store a large amount of symmetric keys for the attestation to work safely, or be faced with numerous ARs at once after being inactive for some period of time. To the best of our knowledge, no RA architecture exists such that it solves these issues.

Goals and Contribuions.

This gap is also what inspires our work. Here, we will attempt to construct a RA protocol targeting low-end devices which, unlike the classical RA, can function on an one-to-many

basis and is hence significantly more efficient. In particular, we hope to enable the PRV party to process multiple ARs at the same time, while also producing a report that is verifiable to all VRF parties. An architecture with this property will go a long way towards improving the efficiency of RA in the IoT sector, and will therefore be a crucial addition to the vast spectrum of currently existing RA techniques.

Our work will take the format of a Bachelor's thesis, thus concluding our studies of Computer Science towards the degree of Bachelor of Science (BSc.) at the Julius-Maximilians-University of Würzburg. Moreover, our work is also motivated by the Secure Internet of Things Management PLatform (SIMPL) project [8], at which the JMU is involved, among other parties. In SIMPL, trust assessment of devices can be achieved with the assistance of RA. Notably, the participating entities have access to a Blockchain (BC), for example, for logging or storing sensitive data. This allows a VRF to, e.g., query the BC in order to fetch the states of a PRV that are considered valid, and match them against the one PRV sent as a response to the attestation challenge. We aim to show that the trust management framework of SIMPL provides the flexibility to offer secure and functional one-to-many RA as a novel solution.

Outline.

The rest of our work is organised as follows. The next Chapter 2 concerns itself with the general security and trustworthiness in the IoT sphere; furthermore, it provides an introduction to the concept of RA, giving us a base to build upon. Chapter 3 presents an overview of a set of existing RA architectures, grouping them by types. The subsequent Chapter 4 forms the main part of our work. There, we discuss the RA process in an one-to-many scenario and show its inefficiency. We afterwards design the so-called *Aggregatable Remote Attestation* which builds on a selected one of the techniques presented in the previous Chapter 3. Thereupon, in Chapter 5, we present in detail our implementation of the Aggregatable RA as a Proof-of-Concept (PoC). In the following Chapter 6, we evaluate our design in terms of both security and performance, showing its efficiency even in the case of a large amount of RA challenges. Finally, Chapter 7 concludes our work by summarising it and also suggesting some points for future research.

2. Background

Ever since a modified vending machine at the Carnegie Mellon University became the first (non-computer) device to be connected to the Internet [9], the number of such appliances has been constantly on the rise. Some 17 years later, in 1999, the term IoT was proposed by the British technology pioneer Kevin Ashton as a description for a system of interconnected devices [10]. The devices-per-capita ratio would then go on to increase from 0.08 in 2003 to 1.84 in 2010 [11].

Nowadays, IoT refers to the concept of having intelligent physical objects, dubbed "Things", that autonomously share data and communicate with each other over the Internet, as well as integrating them to the global Internet infrastructure [9]. Thus, a vast information system is created. Every year, new "smart" gadgets emerge, and digital, computerised components are being added to previously analogue systems, integrating them into the IoT world. Its elements can be currently found almost everywhere, starting from our homes (heat and light sensors, printers, smart TVs) and reaching up to the medicine and the military (robotic assistants, drones) [5]. For this reason, it is no wonder that IoT has a stable and very relevant position in the business and economics of today. The online statistics portal Statista has estimated that, as of 2030, there will be over 50 billion IoT devices worldwide [12]. For the purpose of comparison, that number denotes an amount over twice as large than the one approximated for 2018.

While the very nature of IoT encourages user-defined applications, it does also mean that (in some fields more than in others) the applied security measures may be rather generic. This is supported further by the above mentioned exponential growth of IoT over such a short period of time: just a couple of years ago, probably few would have expected that a scenario of billions of interconnected devices worldwide is plausible; therefore, the security aspect might not have been considered of great relevance during design phase.

Alas, the rising amount of embedded and IoT devices, as well as their applications, also turns them to rather attractive targets and thus widens the attack possibilities of adversaries willing to infiltrate IoT networks. This is splendidly exemplified by well-known cases such as the Stuxnet worm [6] from 2010 and the Mirai botnet [1] from 2016, where the infection of critical devices has significantly endangered various systems, sometimes even on a global level. More recently, in 2018, the developer platform GitHub[1] suffered the largest Distributed Denial of Service (DDoS) attack in history, peaking in 1.35 Tbps [7] – way over the previous record, which was held by Mirai. One can therefore easily see how

[1] https://github.com/

crucial it is to have reliable, proven means to establish the trustworthiness of an IoT entity before initiating any sort of communication or cooperation whatsoever.

2.1. Establishing Trust

Breaches in security can often be traced back to flaws of the applications or the Operational System (OS) run by the compromised device [3]. The most straightforward solution would be, without doubt, to design and write applications and OSes in such a way that they are bug-free and exploit-proof. Alas, this is barely possible in theory, let alone in practice. Modern OSes have an extremely high complexity degree even without considering the myriad of drivers and applications that exist and introduce new possible attack vectors. Consequently, any rigorous testing methods that can be found are also suboptimal in terms of complexity, but also affordability. On the other hand, cheaper and simpler solutions would probably not suffice as an assertion of a system's security. Even if the opposite were the case, yet another challenge then emerges by the users as unpredictable actors who may, through their actions, introduce security holes into a system themselves.

Therefore, it is clearly visible that no absolutely secure OS or application can exist, such that it simultaneously conforms to the modern standards regarding performance and functionalities [3]. This has forced developers and researchers aiming to create trustworthy platforms to opt for (or come up with) alternatives with wide applicability which are also plausible to implement in practice.

Among the most wide-spread methods working towards that goal in current days is the usage of Trusted Computing. That concept gained popularity with its introduction to the general public in 2005 by the Trusted Computing Group (TCG), successor of the Trusted Computing Platform Alliance, which informally defines a "trusted system" as one that "behaves in the expected manner for a particular purpose" [3]. Still, descriptions of what is called a Trusted Platform Module (TPM) can be found in works dating as far as 2001, as discussed by S. Pearson [13], and architectures like AEGIS [14] were created even earlier.

The TCG sets three major requirements to any Trusted Platform [3]:

- shielded locations and protection capabilities onto the platform that are safeguarded by means of hardware or software,
- the ability to measure (and store measurements of) the own integrity and state (that is, before all, the content of the memory), and
- mechanisms for *attestation*, i.e., asserting the benign state of the device.

Those categories, whilst listed separately, often make use of each other to fulfill their goals. Secure storage is more often than not provided onto the platform by the manufacturer, for example, in the form of (Programmable) Read-Only Memory (ROM), or PROM. Self-measurement is not difficult to achieve by simply deploying code with that functionality. In this work, we are however especially interested in the 3rd category – the attestation. In particular, we will be looking into what is known as RA. Other attestation types include attestation by the TPM, platform authentication, etc.

As for the TPMs themselves, their main tasks include (pseudo-)random number generation (such as nonces, or derived keys), data sealing (by using the protected storage), and hashing of given information (such as the contents of the device's memory) [15]. TPMs are often regarded as *roots of trust*, that is, the first piece in a device's chain of trust.

2.2. (Remote) Attestation

One of many definitions of RA, as given in [16], is "the activity of making a claim to a 3rd party about the properties of an entity by supplying evidence which supports that claim" – or, in other words, proving that an entity is not behaving in an unexpected, malicious way. RA inside a given network demands the existence of at least one device in a benign state, i.e., a device that is sure not to be infected by malware; such device is called a VRF. VRF *attests* an untrusted entity, called PRV, using a challenge-response principle. The challenge is modelled in such a way that PRV can generate a valid response to it if and only if it is currently not infected by malware [3]. RA is not limited to the IoT world only – while there it can be applied to assert that a device is safe and behaving as expected, one can also use it in different contexts. For instance, software developing institutions can use it to check for unauthorised changes of their product, or whether the copy of the product is genuine.

At this point we consider it important to give a more detailed overview of a standard RA process, for better understanding. For visualisation, the reader should refer to Figure 2.1. There, an example of RA depicting a use in the IoT sphere is given, where a new device may, e.g., wish to join an IoT network and needs to be attested beforehand. As mentioned earlier, that process includes two parties – VRF and PRV. Understandably, the exact implementation will vary between the different architectures; yet, the base principle will remain the same.

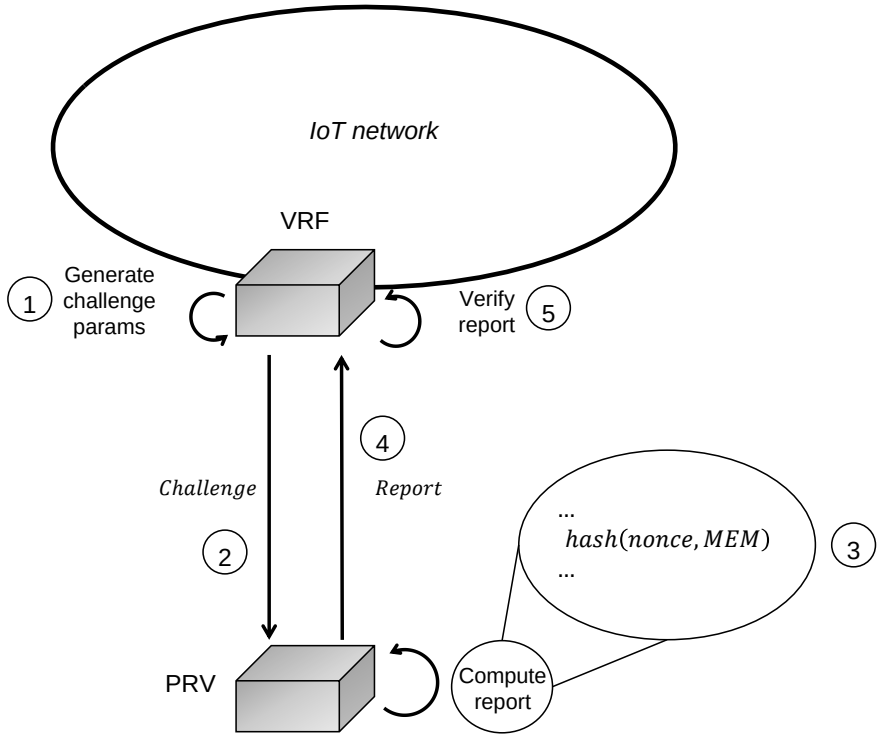

Figure 2.1.: Classical Remote Attestation process with a Verifier and a Prover. An example from the IoT sphere.

Often, VRF and PRV are expected to share a secret symmetric key K_{att}. In both parties, this key must be protected from unauthorised access, for example, in a memory region safeguarded by means of hardware. At some point, VRF will initiate the RA process. It

begins by generating the challenge parameters specific for that cycle (1), such as the nonce or the boundaries of the exact memory region which is to be attested. It will then send an attestation request to PRV containing those parameters (2). The parameters may be also encrypted with K_{att}. Once PRV receives them, it performs a call to the instance of the RA architecture deployed onto it (3). That architecture should, for example, compute a cryptographic hash of the memory region defined by the received boundaries using the secret key and the nonce[2]. Any other data required by VRF is also hashed. The resulting report is then sent back to VRF (4), who verifies its validity (5) also with the help of the key. This can be done by VRF either by computing the expected values itself, or by comparing the received report with a list of valid values[3]. Should the result be evaluated as valid, PRV is declared to be in a benign state and therefore trustworthy. Note once again that some architectures might have slight deviations from this description. A crucial consideration is that without the shared secret which is K_{att}, an attacker could impersonate PRV. This is possible by listening to the network, thus getting the challenge parameters, and then, e.g., computing the (now keyless) hash of the memory region and the other parameters (as an adversary is supposed to have access to all non-protected memory).

The classical RA is hence an one-to-one process as it includes only two actors. In particular, the necessity to use shared symmetrical keys between the two parties is also a significant issue, since for a single entity, the number of keys it needs to store exhibits linear growth in relation to the total number of entities in the IoT network. This also points to the low scalability of the classical RA as nowadays networks can easily contain thousands or even millions of devices.

IoT networks often include various low-end devices, such as sensors, with very limited computational and memory resources on their disposal. As an example, consider a simple light sensor in a Smart Home environment. Moreover, for such platforms, security may be an aspect that has not been given high priority by the manufacturers either, for instance, due to economic incentives [4]. These limitations are critical to consider when choosing an appropriate RA scheme.

RA can be performed through techniques that can be separated in four distinct groups – *hardware-based, software-based* and *hybrid*[4] – as described in [18]. The first type includes RA performed in terms of ROM, secure co-processors and additional secure hardware attached to the platform. The deployment of such hardware elements over a large network of devices, while possible, could eventually prove rather costly.

Thus, *hardware-based solutions* of this sort are at times rather unwell suited for utilisation in the IoT context. On the other hand, *software-based* RA architectures often rely on very restrictive assumptions [19], and various attacks against these have already been demonstrated [20]. An alluring alternative is created by opting to also include elements such as ROM and a Memory Management Unit (MMU) to assert the correct workflow of the attestation process, resulting in a *hybrid* solution. This basic idea can also be applied to lower-end devices – a property likewise valid for some hybrid architectures, i.e., ones that rely on hardware with some assistance by software. That flexibility is what motivates many other researchers as well. Indeed, research and both theoretical and practical work has already been done on the topic of designing such manageable techniques that are able to operate with minimal requirements – as presented in Chapter 3.

Finally, worth mentioning for the sake of inclusion are some RA subtypes with various properties. Note however that we do not consider them well suited for our use cases, and

[2]Recall that hashing, along with secure storage, is one of the requirements for TPMs by the TCG.

[3]These could be, e.g., directly stored inside VRF, or fetched from a database.

[4]An alternative grouping of RA techniques, based on their exact approach, as well as an analysis of those groups' properties, is presented in [17]. Note that this grouping is out of scope for this work.

will therefore leave them, for the most part, out of the scope of this work.

- *runtime attestation* concerns itself with generating attestation evidence during runtime. Examples include the work in [21] and ATRIUM [22].
- *control flow attestation* can be considered a type of runtime attestation and works by asserting that an application's control flow path is correct – possibly without requiring the source code. Notable mentions are C-FLAT [23] and LO-FAT [24].
- *RA with resilience to physical attacks* also exists, although most RA architectures exclude this scenario. Notable examples include DARPA [25], EAPA [26] and SCAPI [27].

3. Related Work

RA is a sphere of growing relevance in the IoT sector, as was previously discussed. It is, therefore, no wonder that there has been vigorous ongoing research focused on perfectionating the existing RA architectures and developing new, even more robust ones. In this section, a non-exhaustive overview of selected RA schemes for low-end IoT devices is presented, together with their requirements, main benefits and disadvantages. It is important to note that, unless mentioned otherwise, most of those techniques have been designed with the same – or very similar – adversarial model in mind, namely an adversary that has control over both the platform of PRV and the communication channels. In other words, it can freely load and unload malicious code, read unprotected memory, eavesdrop, inject packages, etc. However, it is assumed that the adversary does not (or cannot) tamper with the hardware itself, so as to, e.g., physically extract stored secrets.

3.1. Hardware-based RA

Hardware-based schemes are ones that modify the platform's hardware (or introduce additional pieces of hardware) to provide dynamic attestation. They utilise modules like secure co-processors, memory management units and ROM. Additional CPU instructions can optionally be introduced as well. A vast number of devices, such as laptops and even smartphones, already employ similar procedures.

TrustLite

One of the two architectures that build up a "base" for RA schemes on small devices is TrustLite [4]. It relies on a programmable ROM (PROM) and a Memory Protection Unit (MPU) – a lightweight version of a standard MMU. TrustLite provides its functionalities based on four main components: an execution-aware MPU (EA-MPU) that controls access based on both queried and calling code addresses, a Secure Loader setting the appropriate rights in the EA-MPU during boot, a custom Interrupt Service Routine (ISR) protecting against data leakages during interrupts, and a "trustlet" table with information about every trusted application. The authors point to the architecture's flexibility as its main benefit. They state that it can easily be modified to perform different tasks, depending on what is needed. Moreover, they mention that modifying a device engine to run TrustLite has, in their experience, proved rather easy. The main disadvantage of TrustLite remains its static initialisation: no guarantees on security can be made in case an application is deleted or a new one is added during runtime.

TyTAN

The above-mentioned problem is exactly what TyTAN [28] aims to solve. The TyTAN architecture, which is similar to TrustLite, allows for tasks to be loaded and unloaded during runtime while still maintaining functioning RA capabilities. Like TrustLite, it assumes that the device has an MPU and a ROM unit at its disposal. However, a real-time operational system (RTOS) is also needed to meet the real-time guarantees that the authors aim to provide. Similarly to TrustLite, TyTAN makes use of an EA-MPU and a secure boot component. A set of keys is available as well, so as to enable secure Inter-Process Communication (IPC). Another crucial element is its EA-MPU driver, which modifies the EA-MPU dynamically when a task is loaded or unloaded. Finally, instead of a custom ISR, a trusted Interrupt Multiplexer (Int Mux) is applied before calling the interrupt handler. The tasks are separated in *secure* and *normal* tasks. Even the OS has no access to the latter. The TyTAN architecture has been implemented as a PoC. While little data has been provided regarding the overall performance of TyTAN, the general understanding that the memory overhead is significant.

Sancus 2.0

A well-known architecture, Sancus [29], has been redesigned and improved to form Sancus 2.0 [30]. Targeted devices are resource-constrained ones similar to a MSP430. Sancus 2.0 can be used for module isolation, RA, as well as secure communication. A set of cryptographic functions and safely stored keys is used, as well as a custom Memory Access Logic circuit for program counter-based memory protection. The authors also extend the core with additional instructions, e.g., for encryption and decryption used in RA. There, the contents of the device are encrypted and sent to VRF. A nonce and various keys guarantee freshness and authenticity. The authors also provide a PoC and argue about the security of Sancus 2.0's functionalities. They show that the overhead of using Sancus 2.0 is acceptable.

3.2. Software-based RA

Architectures implemented entirely in code also exist. While this can be considered an advantage, they often rely on assumptions regarding, e.g., the adversary model which are rather unrealistic. Among the most common ones are "adversarial silence" during the RA process [31], as well as tight time or storage space constraints [2].

SWATT

This scheme [32] relies roughly on the same ideas as Pioneer [33], the first software-based RA architecture. It is designed for small devices, with the authors providing a PoC on an 8-bit microcontroller. While no dedicated hardware is required, VRF is assumed to have exact knowledge of PRV's hardware specifications. SWATT uses (pseudo-)random traversal of PRV's memory to calculate its checksum. The crucial element is that the calculation is made in such a way that even a single additional `if`-command injected by malware would add a noticeable time overhead. The authors point that SWATT can also be used for virus checking; however, the assumptions they set are not realistic (VRF needing to know exact hardware specifications of PRV, etc.).

SIMPLE

The authors of SIMPLE [2] aim to provide formally verified RA. This work forms a bridge between software-based and hybrid RA as it requires no additional hardware modules (except for, e.g., Flash memory which is considered standard. Hence our decision to place

it in this Section), but makes use of a Security MicroVisor (SμV)[34] (software-based). It is its duty to load applications, verifying them beforehand. Unsafe applications trying to compromise the memory are denied access; unsafe, but necessary operations are replaced by virtualised safe ones stored in SμV's memory. RA is achieved by hashing the memory and makes use of an authentication and an attestation key shared between VRF and PRV. The authors argue formally that SIMPLE is secure if SμV is the first application to be loaded.

3.3. Hybrid RA

Those RA schemes that rely on a combination of software and hardware security features are also known as *hybrid* ones. In particular, they relax the unrealistic assumptions associated with software-based approaches by also allowing hardware modifications or additional hardware elements. Hybrid schemes are proven to be applicable on low-end devices as well, making them attractive for this work.

SMART

Together with TrustLite, SMART [31] builds up the base of many hardware-based and hybrid RA schemes. The prerequisites for SMART most notably include a shared secret key K between VRF and PRV, a so-called "Attestation ROM", and a secure storage inside the CPU accessible only from SMART code. Two compiler enhancement tools, Deputy and CQUAL, are applied in the SMART procedure to tackle possible memory leakages. The authors argue that, based on their PoC, the only possibly relevant overhead is the calculation of the checksum. The main benefit of SMART is that it is not only easily implemented, but also multi-functional due to the optionally executed code. This allows it to be used for, e.g., attested measurement reading. Dynamic application (un-)loading is not considered.

SMARM

The authors of SMARM [35] aim to solve the problem represented by *roving malware*, that is, malware that can relocate itself. Their work is heavily based on SMART and has the same requirements, with the addition of a reliable read-only clock. SMARM works by attesting PRV's memory segment-wise using SMART, with events such as adversary allocation and device functionalities being allowed to occur only in between two block attestations. The measurement order is a random permutation defined at the beginning and kept inside a protected storage. Since SMARM is non-deterministic, its malware detection rate is not 100% as shown by the authors. It is however possible, by repeating the process multiple times, to reduce the failure probability. SMARM's biggest strength is allowing the device to function during RA time. However, this also represents a weakness as it allows the adversary to avoid detection by roaming through the memory during the RA process.

HYDRA

This is a relatively new hybrid RA technique [18] that does not explicitly require any MPU or ROM; however, it makes use of the separate microkernel seL4 [36] as its only prerequisite. Note that seL4 itself does require some minor hardware modifications of the microcontroller it is deployed on. HYDRA borrows some key elements from SMART such as attesting the content inside some boundaries given by VRF. It does not, however, offer code execution at a given address, or secure inter-process-communication. Thus, its general functionality consists of verifying the validity of the RA request to avoid possible

Denial of Service (DoS) attacks, and then computing a cryptographic checksum of the code segment in the given boundaries. For access control, ROM and secure storage, HYDRA relies on seL4. The seL4 microkernel is also the main drawback of HYDRA: while it does remove the need for ROM and MPU, these are very often available in embedded devices by default. Thus, seL4 represents an additional piece of software which is not needed for other purposes. Moreover, the authors of HYDRA explicitly state that their architecture does not aim at really small devices, but at rather more sophisticated ones. The main benefit is that only little modifications are required on the original device platform.

HEALED

HEALED [37] is novel in that it provides not only attestation but also recovery of infected devices to benign state. Required are only a ROM and a MPU. HEALED functions by making a random device try to attest another one at a random interval. PRV's memory is divided into ordered segments and attested by hashing in a tree-like manner, creating a Merkle Hash Tree. If PRV turns out to be infected, VRF seeks a third device H from the same type as PRV. After successful attestation by PRV, H finds the corrupted code inside VRF by using the Merkle Tree and tries to patch it. HEALED suffers from major drawbacks in the sphere of scalability, as an exponentially rising number of devices need to share symmetric keys. On the other hand, healing is a crucial benefit that is currently rarely considered in RA schemes and thus should not be ignored. The authors provide PoC implementations over SMART, TrustLite and a drone testbench, and show that HEALED weakens the performance in a negligible range.

VRASED

The next hybrid RA scheme discussed here is VRASED [38], an architecture with security "verifiable-by-design". Hardware requirements are minimal: VRASED focuses on low-end IoT devices with ROM, SRAM, a Flash, and an additional small hardware module. In VRASED, the platform is divided into smaller components. They are then represented as Finite State Machines in *(i)* Verilog HDL, and *(ii)* the model-checking language SMV. The functionalities and the properties themselves are then formulated as Linear Temporal Logic formulae, and each sub-module is checked against the formulae subset it is supposed to cover. Once the sub-modules are verified, the same is done for the whole platform. While the formal verifiability is an obvious benefit, it also significantly increases the process' complexity. The authors provide an implementation on a Basys3 Artix-7FPGA board and suggest that it is portable to other platforms with minimal modifications.

APEX and PURE

To conclude the hybrid schemes, we mention APEX [39] and PURE [40], two architectures relying heavily on VRASED. Aside from RA, APEX is capable of generating formally provable evidence that specific code has been executed, as well as a proof that the produced result is as intended. The authors also provide an implementation on an OpenMSP430. While the memory overhead is negligible in comparison with VRASED, the additional computation time is considerable. PURE on the other hand allows for (also formally provable) system-wide reset, software update and memory erasure. Its overhead in both computation and memory is insignificant, as asserted by the authors. An implementation is also available.

3.4. Application of Attestation

Here, we present approaches which are still relevant to the RA topic, but cannot be categorised as completely new or different RA primitives. They are rather novel ways to utilise already existing RA protocols.

ERASMUS

The ERASMUS architecture [41] belongs in a separate category since its authors do not present a completely new attestation scheme. Instead, what they contribute with is the use case. In particular, they review the case where malware can infect and leave the device between the attestation requests. This scenario is addressed by making the device self-attest at regular intervals, and save all measurements until a request from VRF arrives. Then, all saved measured states are sent and removed from the memory. While this scheme has no particular hardware requirements by itself, it demands that another attestation technique is deployed over the device (and thus inherits that technique's own requirements). The authors suggest implementations over a SMART+ and a HYDRA instance. The logs need not be stored in secure memory, as any changes inflicted by malware can be detected by VRF upon log collection. One benefit of ERASMUS, other than having a self-measurement log, is the fact that attestation requests by VRF cause no further drains of computational power than those needed to simply send all saved measurements. Furthermore, ERASMUS can be extended to provide on-demand attestation or self-attestation at irregular intervals.

SARA

We consider the SARA protocol [42] to be worthy of a mention, even though it is not an actual RA architecture. This work concerns itself with asynchronous RA. VRF and PRV use a synchronised vector clock which is updated after each event. SARA operates based on a publish-subscribe approach. VRF sends RA challenges to the Publisher, and the Subscriber (PRV) eventually uses them to attest itself. Once VRF sends a RA request to PRV, the Subscriber service will send the RA result. It is also possible for VRF to subscribe to the Publisher to directly receive updates about PRV. The authors demonstrate the efficiency of SARA through a PoC. They also note that the main benefits are asynchrony, selective attestation and historical evidence.

3.5. Summary

A large array of RA architectures and approaches tailored for resource-constrained devices already exists, and more will eventually be developed and presented in the near future. Here, we discussed a selected subset of those architectures. Some of them also offer additional features aside from RA, as we will elaborate in more details in Section 4.3. However, those are all in their essence variants of the classical RA presented in Section 2.2. Hence they inherit most of the properties typical for RA.

An exemplary issue which is not addressed is the one-to-one property observable in RA. All architectures we considered, regardless whether hardware-based, software-based, or hybrid, work with it as a base and do not attempt to improve it; however, as we will discuss in the next Chapter 4, this leads to performance issues when a PRV entity is faced with many ARs and also paves the way for some attack vectors.

Important to note here is that the SARA protocol does make progress in a similar, albeit not the same direction, by introducing the Publish-Subscribe approach. In particular, SARA does not consider the use case where there are multiple ARs coming from different VRF entities. Moreover, it still intends for PRV to process those ARs sequentially, thus not solving the performance issue. This problem will also be what our work revolves around.

4. Aggregatable Remote Attestation

One of the most crucial details about the classical RA is the observation of it being an one-to-one process between a Verifier VRF and a Prover PRV entity. Namely, VRF would send a challenge, also known as AR, that contains specific parameters to PRV. PRV would then be able to calculate an answer to this challenge *if and only if* it is not infected by malware. As a consequence of its one-to-one property, this process was shown to be characterisable as one with very low scalability – a crucial detail in the IoT sphere, where a single network can easily consist of well over a million smart devices. In this Chapter, we will inspect the scalability issue a bit more closely, and attempt to craft a solution by creating the so-called *Aggregatable Remote Attestation*.

4.1. Problem Statement

One could argue that the one-to-one property is not a real issue. After all, PRV just accepts the challenge parameters, executes the attestation process, and returns the report. However, in reality this is not that simple. The one-to-one approach can, for instance, easily be misused by adversaries to create a DoS scenario by bombarding PRV with ARs since PRV will be doing nothing but computing attestations. Moreover, the number of symmetric keys that need to be stored grows with the size of the network and can thus easily cause memory issues.

The whole problem starts to get clearer once one considers the circumstances and events surrounding the attestation. The first crucial point is that a lower-class device is often not reachable directly. An (IoT) network can be visualised as a complex graph, with the nodes representing the separate devices and the edges being the simple (i.e., with hop count = 1) connections between them. Depending on the exact architecture, a set of nodes could, for example, need to pass their whole communication with the outside world through some higher-class mediator device – a hub, a switch, or some kind of broker. A broker in this case is essentially a server that accepts messages from the above mentioned set of nodes and forwards them to their destinations, or alternatively forwards messages to the set of nodes. The underlying reason for this network design choice may vary: different possibilities include reduced coupling within the sub-network, communication control, protection against diverse attacks, etc. Regardless of the case, especially lower-end entities (the ones we are interested in) such as sensors are often "hidden" behind such a broker (cf. Figure 4.1). This implies that any ARs sent to them are first accumulated at the broker before being passed to the device.

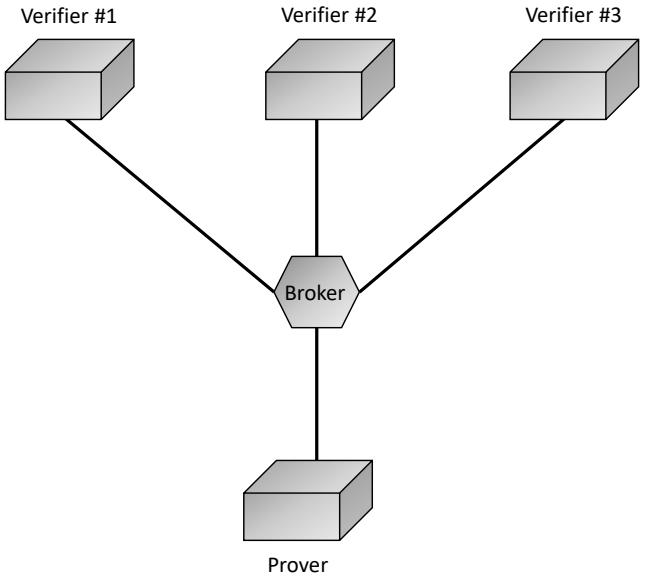

Figure 4.1.: An IoT network including multiple devices, a broker, as well as a distinguished Prover.

Second, it is important to note that devices, regardless of their type, are not constantly available. There are (not necessarily on a regular, foreseeable base) events like maintenance, software/hardware updates, but also, for example, spontaneous downtimes during which the entity is not reachable by external traffic. Ultimately, the device might be merely "sleeping" or executing its standard tasks and due to this reason not be available for any other purposes. It is hence not completely unlikely that an incoming AR (or some other request, in that context) is not processed immediately. To tackle the issue represented by this, such messages are often accumulated at the previously mentioned broker entity. It would then wait for the device to "wake up" (or become available again) in order to send the whole list of pending messages and requests.

It can thus be clearly seen that the scenario of a device receiving multiple AR at once is not only possible, but also plausible. Naturally, more reasons exist for this aside from the broker simply collecting a set of requests. Even if we were to remove the broker entirely, it is not unfeasible that multiple entities, each acting separately as a VRF, decide to perform a RA onto the target device – maybe even simultaneously. Furthermore, a single VRF entity may due to various reasons send a second AR to the same PRV before the first has been processed. To exemplify why this can happen, examine the case where that very VRF is acting as a gateway for a larger subsystem[1]. With different entities from the subsystem needing to verify the trustworthiness of PRV, the gateway entity may send separate ARs. Alternatively, a subsequent AR might be triggered by some unexpected event, while the first one is part of a routine. Whatever the case, it is safe to assume that the different ARs carry different challenge parameters, and therefore need to be processed separately.

What exactly does happen inside PRV when multiple different challenges arrive? To the best of our knowledge, the currently existing RA architectures simply proceed to process those challenges sequentially. The separate ARs may also be checked for validity beforehand, but that is irrelevant for our considerations. In summary, the process goes as follows:

1. PRV wakes up/becomes reachable.

[1]See the *Facade* Structural Design Pattern described by the Gang of Four [43].

2. The broker entity sends all ARs to PRV.
3. While there is an AR in the queue:

3.a) PRV attests itself with this AR

3.b) PRV sends report to VRF through Broker

3.c) Return to 3.

4. The results are sent back to the broker, who forwards them to the VRFs.

For visualisation, one can also refer to the sequence diagram on Figure 4.2.

An important element to note would be the repetition of the hashing of the own memory during 3.a) (as a remark, we have opted for this description since many RA architectures, as seen in Chapter 3, do indeed revolve around memory hashing). This is also the most problematic point: however small the memory size inside a low-end device might be, computing a hash over the whole memory still does take a significant amount of time and computational resources (at least in comparison to the other tasks). The addition of the nonce to the hash can be ignored in the efficiency considerations as all nonces share the same constant size and are significantly smaller than the memory. Regardless, the computation-intensive chaining of hashes could eventually result in longer periods of time where the device is bound to the attestation process and cannot proceed with its default tasks. Hence arises the opening for the DoS attack mentioned above.

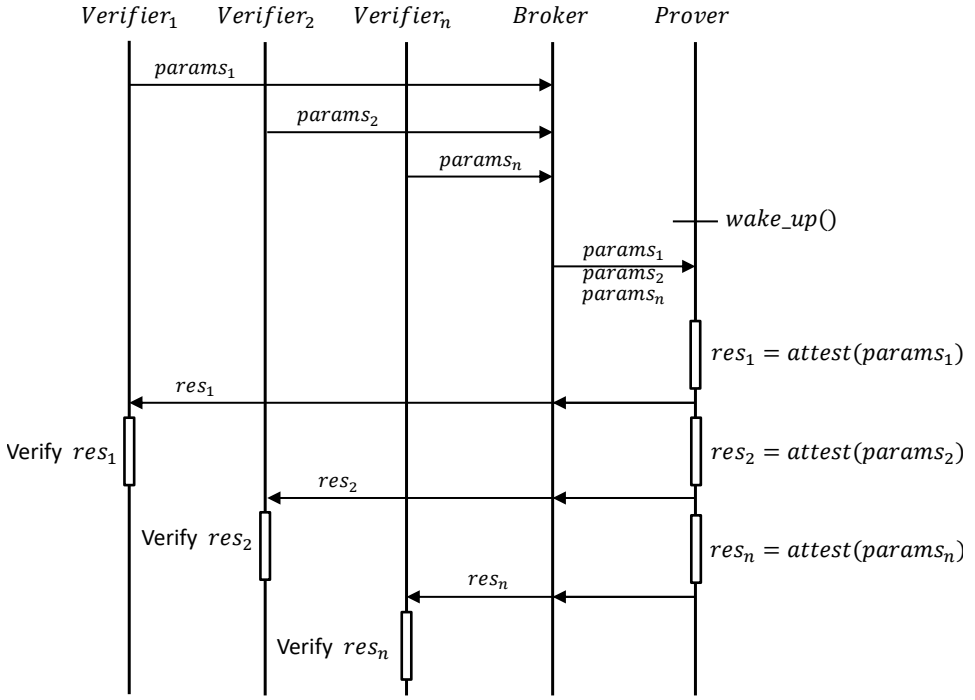

Figure 4.2.: Remote Attestation process with multiple Verifiers, a Broker and a Prover.

4.2. Approach

We do believe that this process can indeed be optimised.

For the optimisation, it is only logical to target the underlying reason of the problem, namely the one-to-one property. An upgrade to an one-to-many attestation process would be most advantageous and go a long way towards making RA far better suited for the IoT

sphere. For instance, if PRV was able to somehow aggregate all of the pending ARs, it would then merely need to calculate a solution to the challenge aggregate and send it back to the Verifiers.

This is also the goal we aim for: A RA protocol with the capabilities to (i) aggregate challenge parameters to a single one, and (ii) produce a publicly verifiable report. In other words, the mere contents of the report should suffice as an assertion of PRV's benign state, particularly without needing any shared secrets or other non-public elements of that manner.

With meticulous considerations, a modified version of the standard RA process, as described in Chapter 2, has to be crafted. This new technique, called *Aggregatable Remote Attestation*, should allow a PRV to collect a set of ARs, iterate over them and create a single challenge, whose parameters are derived from the ones of the individual requests. PRV will then be able to compute a sole response to the challenge aggregate and return that report to the broker entities (or the Verifiers, depending on the case). Great care is to be exercised though, as that construct needs to have the property of being verifiable especially by each and every single one of the Verifiers which sent the initial ARs. This in turn entails high probability that not only the self-measurement of the PRV will need to be modified, but the verification process done by the VRF as well.

In the previous Chapter 3, we offered an overview of a selected set of RA architectures, grouped by their type: hardware-based, software-based, and hybrid. Each scheme was briefly analysed in terms of prerequisites, main benefits and disadvantages, as well as functionality. We saw that many of those schemes do indeed also offer other features aside from RA – healing of infected devices, protection against roving malware, etc.

The first part of our contribution will thus consist of the post-processing of this overview. We are to evaluate the properties of the presented schemes in accordance with our use case and present an aggregated list. Thenceforth we shall select a single architecture which we deem to be best suited as a base for the Aggregatable RA. That is, in the main part of our work we will modify the VRF and PRV algorithm of the architecture in order to convert it to an one-to-many process. In this sense, our approach bears some similarities to ERASMUS and SARA that were discussed earlier in Section 3.4, as they too employ another RA protocol at their core.

Afterwards, having picked a basis scheme, we will describe the exact modifications and how they help us achieve our goal. We argue that those changes do not introduce almost any additional security holes and attack vectors that were not present before. The only exception is that we discard the symmetric key and hence the authentication of VRF. Finally, we will discuss eventual topics for future work which, if implemented, would make the whole construct more robust.

4.3. Architecture Evaluation and Selection

Providing RA capabilities is what unites all techniques discussed in Chapter 3. Many of them do, however, also go beyond that and offer additional functionalities and benefits. As a first step, we are going to present those (at least as far as they are mentioned by the authors of the corresponding works):

Dynamic applications:

This denotes the capability of an architecture being able to guarantee functioning RA even if applications are loaded or unloaded during run-time. TyTAN [28] is a primary example, and SIMPLE [2] allows additionally for verification of the application safety prior to loading.

Asynchronous RA:

This means that the attestation is allowed to happen not immediately after the device receives the AR, but also afterwards when it is available. Notable implementor is SARA [42] which utilises a publish-subscribe approach.

Code execution:

After the self-measurement of the memory is complete, the device executes the code stored at an address passed with the challenge parameters and hashes the result. This is one of the most valuable properties of SMART [31], but is also available in APEX [39].

Protection against roving malware:

Roving malware is one that can re-allocate itself to another memory location and thus eventually avoid standard RA approaches. Pseudo-randomness-based protection is offered by SMARM [35].

Recovery:

If a device is found to be infected, performing a clean-up procedure is recommended. The only architecture offering this out-of-the-box is HEALED [37]. A relaxed property – applying memory erasure and system-wide reset – is also available in PURE [40].

Formal verifiability:

Many authors argue about the security of their architectures. Few do however prove it *formally*. The most notable schemes with this property are SIMPLE [2], VRASED [38] and PURE [40].

Our scheme of choice, after some considerations, is ultimately SIMPLE. While it is a really novel technique, it does indeed already appear very promising. We will now present briefly our line of thought that led to this decision.

As already noted, SIMPLE, in the form in which it was presented, can be verified formally by various means described by the authors. This provides a stable basis for extensions such as the ones we wish to apply. It does also not require any additional hardware to deploy – as the authors show, SIMPLE takes a very specific spot bridging the software-based and the hybrid architectures (cf. Figure 4.3).

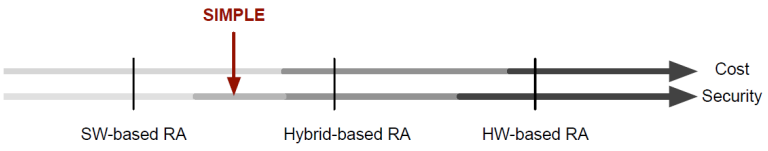

Figure 4.3.: Position of SIMPLE on the spectrum of RA architectures [2].

Another positive aspect is that SIMPLE targets resource-restricted devices, exactly as the ones we are considering. Moreover, as remarked above, it operates correctly even in the case of applications being removed or new ones being loaded during run-time. As a concluding point, the authors have implemented it as a PoC to show its real-world applicability. Its main component, SμV, is open-source on GitHub[2] and can therefore be downloaded and extended freely.

[2]An implementation in C can be found on GitHub under https://github.com/m3mmar/verified_SuV/.

4.4. Functionality of SIMPLE

We will adapt SIMPLE to our use case and make the necessary modifications to convert it to an one-to-many architecture. However, first we will go into more details regarding SIMPLE to provide better insights about its functionalities, as we consider having understanding of its internals important for the rest of this work.

As discussed, SIMPLE falls under the software-based category as it completely relies on the Security MicroVisor SμV. It provides the same security guarantees as hybrid-based approaches, without requiring neither hardware support in the form of a ROM or a MPU, nor hardware modifications. Moreover, SIMPLE implicitly guarantees Control Flow Integrity, as asserted by the authors.

The MicroVisor SμV is an open-source software-based hypervisor. It provides trusted MPU-like memory protection and isolation. SμV uses selective software virtualisation as well as code verification on assembly level to isolate a software-based Trusted Computing Module, or TCM (similar to a TPM). A necessary and sufficient prerequisite for SμV is having a simple, single-threaded microcontroller that lacks an MPU, supports global interrupt disabling, and has sufficient non-volatile memory (e.g., Flash or ROM). This is standard even in small IoT devices [44] and is thus not considered an explicit hardware requirement.

SμV needs to be installed on the device before any other application. It reserves a part of the memory for the TCM, which is immutable and has non-restricted access. The rest is split to Data Memory and Instruction Memory (cf. Figure 4.4). In particular, Data Memory is not allowed to read, write, or execute any instructions; it merely holds application data. Contrary to that, Instruction Memory can read and write data, jump freely within itself and call specific entry points of TCM memory.

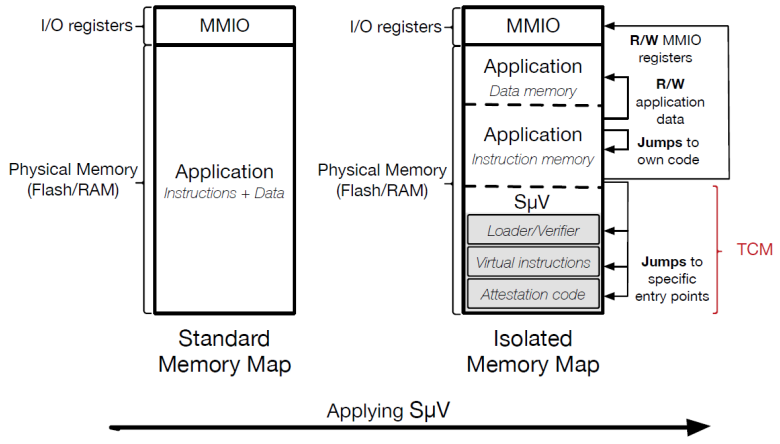

Figure 4.4.: Memory mapping without and with the Security MicroVisor [2].

Figure 4.5 illustrates the memory access rules described above. Most notably, SμV has unrestricted capabilities within the whole program memory. On the other hand, any untrusted applications are allowed to use specific access points in the secure memory, and freely execute instructions in the non-secure program memory.

Application deployment may only occur through SμV, which verifies incoming applications at load time. A crucial element is that unsafe instructions, which are still essential for normal operation, are replaced by safe virtualised instructions stored in the TCM memory. Applications attempting to compromise the memory are rejected by SμV at boot time.

	Program Memory		Data Memory / MMIO
	Secure Area SμV memory	**Non-secure Area Instruction memory**	
SμV	rwx	rwx	rw-
Untrusted App	x*	x	rw-

*Execution is only available from specific entry points.

Figure 4.5.: Memory access rules set by the Security MicroVisor [2].

RA with SIMPLE functions as follows. At the beginning, PRV and VRF need to share two keys, K_{auth} and K_{attest}. Moreover, they have monotonically updateable counters C_v and C_p. The counters and the keys are safely stored inside the TCM memory. First, VRF increments its clock and uses K_{attest} to calculate the expected state VC of PRV. That state is sent to PRV along with the clock and a nonce, as well as an HMAC digest with K_{auth} of those elements for authentication. PRV uses the digest to verify the message and updates its own clock accordingly. Afterwards, PRV computes the digest of its current state and compares it with the expected one received from VRF to a 1/0 result. Then, for authentication, it generates the HMAC digest of that result along with the own clock C_p and the nonce, and sends the digest together with the 1/0 result back to VRF. VRF checks the authenticity of the result by using the digest, and makes a conclusion regarding PRV's benignness from that result.

The nonce and the counter are needed as a protection against replay attacks. For visualisation the reader may also refer to the scheme on Figure 4.6.

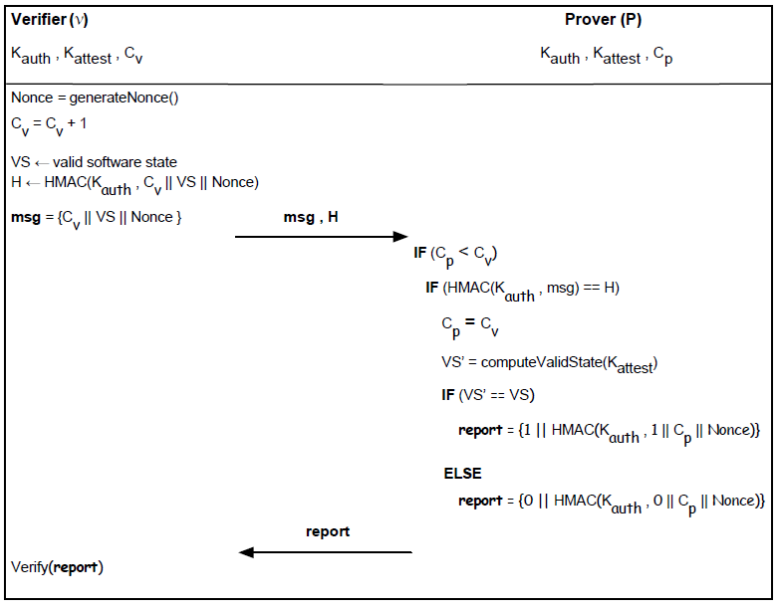

Figure 4.6.: Remote Attestation with SIMPLE [2].

For the hash function, a keyed HMAC-SHA1 is used. Security of the architecture and the algorithm is asserted as follows:

- SμV erases all data that was temporarily stored during RA .
- At the very beginning of the algorithm, all interrupts are disabled.
- Thus, the algorithm is executed atomically. This is a protection against leakages, return-oriented programming and roving malware.

- The algorithm always finishes in finite time, as verified through model checking.
- SμV code is immutable, therefore the results are always reliable.

That proves that if the IoT device is initialised with the installation of SμV, then SIMPLE is secure and guaranteed to provide secure and correct RA.

4.5. Modifications

As elaborated in Section 4.2, we conduct a set of adjustments to the original SIMPLE architecture. In particular, we carefully analyse the individual issues that emerge from the way it was designed initially, and take action to adjust each of them while also attempting not to deviate from the archetype more than is necessary.

For our model, we will be assuming the same adversary characteristics as the authors of SIMPLE – an attacker able to eavesdrop, inject packages, and modify *unprotected* areas of PRV's memory, but not to physically tamper with the device so as to, e.g., gain access to protected memory. Moreover, the adversary has enough computational power to attempt, for example, collision attacks. As mentioned in Chapter 3, this model is typical in the RA sphere; hence, using it sets our approach on the same level as the majority of other schemes.

The first problem, which was in fact already briefly addressed, is related to performance: if the whole attestation is executed once for every VRF and with every nonce, this would take significant time and also creates an opening for a potential DoS attack. Therefore, once the RA process begins, we collect the important data – the nonces n_i – from all pending ARs and construct a single one by combining them, e.g., to a hash chain. Afterwards, the device proceeds to perform the rest of the RA with the new, aggregated nonce n'. In particular, n' is also hashed together with the current state (memory) to a digest we denote as res. When sending the result back to every VRF (or simply to the broker entity), we also append the whole list of nonces N collected from the individual ARs. This way, VRF can

1. check that its own nonce (or all of them, if VRF has sent multiple ARs) is included in the list, and
2. compute the hash chain himself.

VRF will then use the self-computed chain to verify the received result. It can, for instance, compare it with a list of valid states stored in a database or, in the case of SIMPL, a BC. One could try to argue that sending this additional data over the communication channel represents a new security threat. However, this is indeed not a vulnerability since an eavesdropping adversary could have learned the single nonces by the time they were received by PRV.

Secondly, we wish to break away from the one-to-one property of standard RA whilst still making the result verifiable for all VRF entities. An important issue here is the number of symmetric keys that PRV, a resource-constrained IoT device, has to hold in a large network. In order not to require secure key exchange beforehand (or storing symmetric keys), we replace the keyed HMAC function, currently used to hash the memory, with a keyless hashing function, such as a simple SHA-1. We argue in Section 6.2 that this does not create a new attack vector. Additionally, we opt for asymmetric cryptography and applying a digital signature on the result after computing it. For this, PRV will use a single private key K_{priv} stored in protected memory. Once the current state *state* is collected, PRV signs it with K_{sign} to a result $Sign$. It then sends $\{Sign||res||N\}$. Any entities having access to PRV's public key and a list of valid states for the device can verify

that the result is correct. Any VRF who has sent a RA request can thus also verify the validity of the result and the new nonce as described above.

Finally, VRF sending a valid software state VS (even encrypted) to PRV is unnecessary. Instead of PRV, VRF will be the one checking whether PRV's state is benign or not. While it is still possible for PRV to compare its own to the expected state and return a signature of a 1/0 result together with the nonce, this is rather impractical. The rationale behind this is that eventually there can be a vast amount of benign states, and saving and comparing them inside PRV would be rather unfeasible.

For a visualisation of the modifications, we point to the sequence diagram on Figure 4.7.

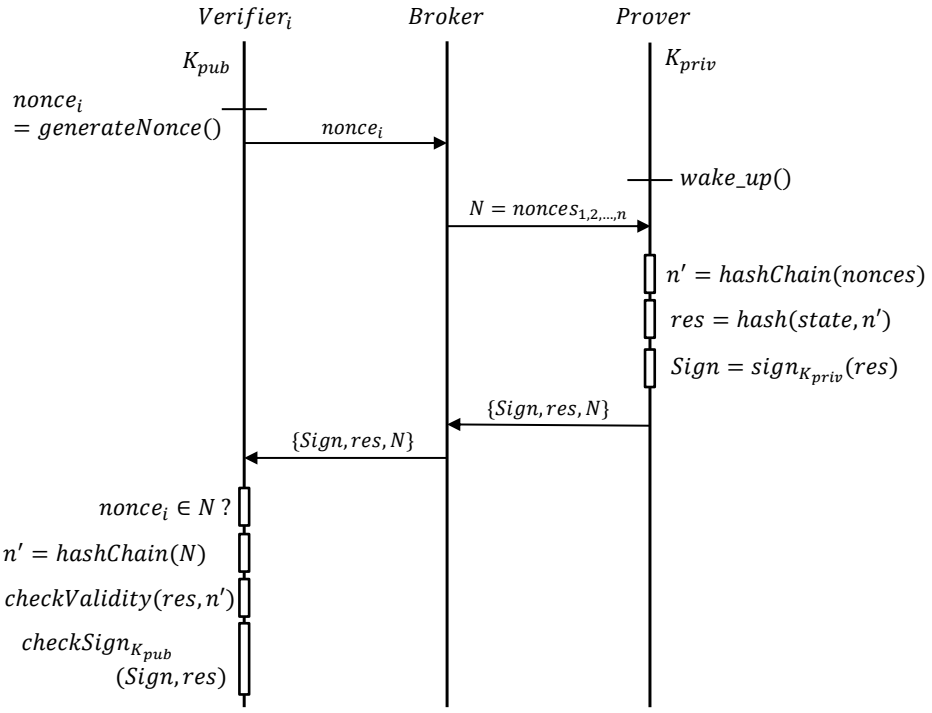

Figure 4.7.: Modified version of SIMPLE.

4.6. Further Considerations

We conclude this chapter by briefly presenting some additional points regarding the modified RA architecture. Note that considerations regarding the security and the eventual attack vectors can be found in Section 6.2.

4.6.1. Retention of Dynamic Functionality

A point which was not examined thoroughly was the property of SIMPLE allowing for valid RA in a dynamic environment, i.e., when applications are loaded and/or unloaded during run-time. The authors of SIMPLE argue that their implementation allows this [2]. Albeit our modifications do not concern themselves with the processing of the memory (or, more accurately: the exact memory locations which are processed), this use case has still not been tested appropriately. Hence, no guarantees are provided for the proper functionality of the modified RA, should applications be (un-) loaded after boot. This remains a subject of potential works in the future.

4.6.2. Formal Verifiability

The final sphere of discussion remains the capability of SIMPLE's RA to be verified formally in terms of functionality and security. This is a similar aspect to the one exhibited by VRASED [38], APEX [39] and PURE [40]. It is also one of the main focal points of the original SIMPLE architecture. Not unlike the dynamic environment, it is not impossible that our changes to the archetype SIMPLE have rendered the means of formal verifiability no longer usable. Again, this is a case that has not been inspected as part of our work. We estimate this aspect to be one of high complexity and far above the reach of a Bachelor's Thesis. Consequently, it can also be considered a niche for future work and research.

5. Implementation

To demonstrate the applicability of our design given in the previous Chapter 4, we also write an implementation as a PoC. The implementation is based on SIMPLE's core component, the Security MicroVisor SμV, as we derived our model of the so-called *Aggregatable RA* from SIMPLE. Previously, we discussed the various benefits of different architectures aside from RA, and declared SIMPLE [2] as one most suited for modifications.

5.1. Microcontroller Testbench

Since our work is concentrated on small devices, an appropriate one needs to be selected for the implementation. Our choice lies ultimately with the Atmel MEGA-1284P Xplained[1] (ATmega1284P-XPLD). The underlying ATmega1284P is a low-power 8-bit microcontroller based on the AVR enhanced RISC architecture [45]. It features 128K bytes of programmable flash memory allowing for concurrent read- and write-operations. Moreover, at its disposal are 32 general-purpose working registers, 4K bytes EEPROM, 16K bytes SRAM, as well as I/O capabilities through a SPI serial port and a JTAG test interface. A close-up view of the board is shown on Figure 5.1.

Those properties allow the ATmega1284P-XPLD to be classified as a low-end device. More accurately, it belongs to the Class 1 devices, following the definitions given by Bormann et al. [44]. This is particularly convenient for our use case as the Class 1 devices do not offer any security mechanisms out-of-the-box. Hence, showing that our Aggregatable RA can be deployed on such a device will be an important statement in regard to its applicability and usefulness.

5.2. Security MicroVisor

Recall that one of our arguments behind the choice of SIMPLE as the scheme to be extended did consist of SIMPLE's main underlying component, the Security MicroVisor SμV, being available open-source on GitHub as mentioned in Section 4.3. The MicroVisor code is written in the C programming language and targets AVR architectures specifically, as described on the GitHub page[2] – another reason supporting our choice of the ATmega1284P-XPLD microcontroller.

[1] https://www.microchip.com/DevelopmentTools/ProductDetails/ATMEGA1284P-XPLD
[2] Once again, the repository is available at https://github.com/m3mmar/verified_SuV/.

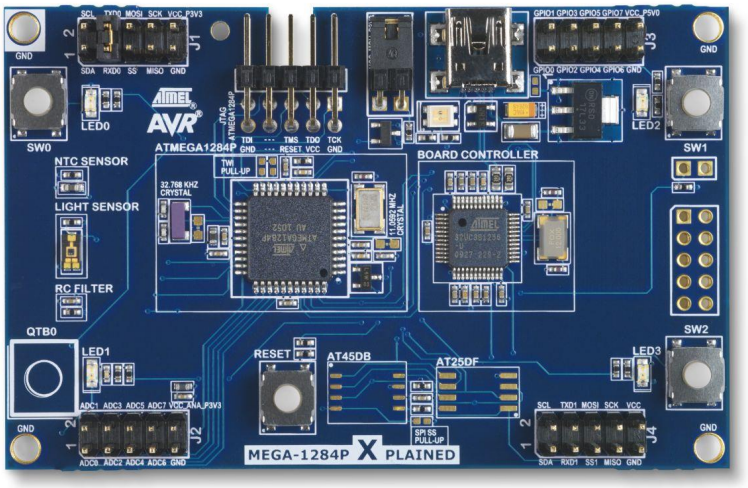

Figure 5.1.: The ATmega1284P-XPLD.

The code of the MicroVisor already includes RA capabilities for PRV as well as a simple VRF entity written in Python. VRF creates and sends a single nonce. Afterwards, PRV computes the already discussed hash of the memory and the nonce, implemented through a HMAC-SHA1. The exact HMAC implementation, after some research, appears to originate from the AVR-Crypto-Lib[3]. Finally, the result is sent back to VRF. VRF has computed a digest itself, and compares it with the one received from PRV. The communication is conducted through a serial port. The VRF code, runnable on any device that supports Python, accesses the serial port by using the pySerial library[4].

5.3. Modifying the Prover

The first element we need to change is the nonce reading. Since we are collecting an unknown number of nonces instead of a single one, we need a flexible way to store them. Currently, the single nonce is written to a `uint8_t` array of length 20. Instead, we model a double-linked list, defined as given on Listing 5.1[5]. Each element of the list contains pointers to the previous and the next element of the list (or `NULL` if no such exists) as well as a pointer to the actual nonce (again, stored as a `uint8_t` array). The full header of the list can be found in Appendix B. In the RA code of SμV, we create a list instance and append an element for each nonce received over the serial port.

```
1 typedef struct node {
2     uint8_t * val;
3     struct node * next;
4     struct node * prev;
5 } node_t;
```

Listing 5.1: Definition of the custom list data structure.

Having saved all nonces, the next step would be to aggregate them. For this, as mentioned in Section 4.5, we can, for example, compute a hash chain (or use any other compression

[3]Can be found at https://wiki.das-labor.org/. Link to a repository with the functions is also included.
[4]See documentation at https://pyserial.readthedocs.io/.
[5]For the list, inspiration was taken from https://www.learn-c.org/.

function). A simple solution is offered by the AVR-SHA1 library[6], whose code we put in the `core/crypto` directory in the place of SμV's HMAC-SHA1. So we define a function

```
1 void aggregate_nonces(sha1_ctx_t *ctx, void *src, void *dest)
```

that accepts pointers to a SHA-1 context (`ctx`), the list with nonces (`src`), as well as a destination (`dest`). The function iterates through the list, feeding the single nonces to the context in a chain-like manner, and subsequently writes the digest in the `dest` parameter. At the end, `dest` contains a single 20 bytes long nonce aggregate, or n' as per Section 4.5 notation. Afterwards, the (already exiting) procedure `remote_attestation(buf)` is called with the aggregate.

Next, we move on to the main MicroVisor code, stored in `core/microvisor.c`, and more precisely to the `remote_attestation(uint8_t *n)` function. There, a HMAC-SHA1 context is created and initialised with K_{attest}. Afterwards, the device memory, block by block, is fed to the context, effectively computing the HMAC. Once again, we replace the HMAC-SHA1 context with a simple SHA-1 one. Note that utilising a collision-safe one-way cryptographic hash function is not entirely necessary, as we will show in Section 6.2 and the referenced Appendix A. The SHA-1 digest is placed in the buffer previously containing the nonce aggregate, in order to return it to `main.c`. With these changes, we have effectively implemented $\frac{2}{3}$ of our modifications. The only remaining element (on the PRV part) is the digital signature.

Due to the relation of our work to the SIMPL project [8], as mentioned in Section 1, the signature process we opt for needs to be compatible with the one on SIMPL's BC. There, this operation is performed by using an Elliptic Curve Digital Signature Algorithm (ECDSA). The curve of choice for the algorithm is the same one as on the Bitcoin network, namely secp256k1 [46]. Hence, in order to reach compatibility, we must apply signature with the same curve.

A C library with the required capabilities is, for instance, micro-ecc[7]. The authors assert that it is deliberately designed to be small and fast, and so as to be applicable for 8-bit processors. We thus include the library's functional files in `core/crypto`. Since the ECDSA algorithm needs a source of randomness, we also add a lightweight Pseudo-Random Number Generator (PRNG)[8]. We use the micro-ecc library to generate a key pair. The private key is placed in secure memory, namely in the same field which used to store the key for the HMAC function. We also save the public key inside VRF. In the body of PRV's `remote_attestation(uint8_t *n)` function, we create a signature context and sign the hash of the memory and the nonce. To return the signature digest to the main function, we add a second `uint8_t *sign` parameter and pass a `uint8_t sign[64]` to it.

From there, we simply send the list of nonces, as well as the digest and then the signature, over the serial port. We utilise special characters for separating the three elements and also for denoting the end of the message. For exact description of the data format, the reader may refer to Appendix C.

5.4. Modifying the Verifier

In the previous section, we described our changes to the SμV code. However, recall that changes to the VRF side also need to be made for it to properly process the new RA results.

[6] Available at https://github.com/.
[7] Available at https://libraries.io/. GitHub repository is linked.
[8] As presented in https://blog.podkalicki.com/.

In the initial phase, we need to generate the nonce(s). For testing purposes, we configure the process to accept an additional numeric parameter k so as to emulate multiple different ARs. The original code creates a single one with `os.urandom(20)`; we call that function multiple times and store the four nonces in a list. Once the generation phase is complete, we log the nonces on the console and also send them to PRV over the serial port[9]. We utilise some special characters as delimiters to make it simpler for PRV to process the input. The reader may refer to Appendix C for a description of the data format.

A similar procedure is applied to the received report. Recall that as per design, $report = \{Sign, res, N\}$ (see Appendix A). Hence, once the report arrives, we first check whether our k nonces are included in N and then build the nonce aggregate n'. The Python script already employs the hashlib library[10] to compute the original HMAC-SHA1. Instead, we will utilise the simple SHA-1 function. Namely, we define a method

```
1 def hash_chain(list)
```

accepting a list and returning the SHA-1 digest yielded by hashing the elements of the list sequentially. We will use this method not only for the nonces, but for the memory compression as well.

For the next phase, we want to check the validity of the received res, that is, the compression of n' and the memory. In a real scenario, such as the SIMPL project, the valid memory state would be fetched from a database or some other type of storage. Since our testbench does not have such an entity at its disposal, we instead compute the expected memory state of PRV by utilising the flashed `.hex` file and the intelhex library[11]. We then use this variable together with the computed n' to calculate their SHA-1 compression (`hash_chain([intelhex, n'])`) and compare it with res.

Subsequently, we need to check the digital signature. For this, we need a library providing ECDSA with the secp256k1 curve, exactly like in the case of PRV. Our library of choice is ultimately python-ecdsa[12]. Once again, in a real-world scenario, PRV will have distributed its ECDSA public key before the initiation of the RA process; however, here we have it statically stored in a variable `public_key`. This allows us to check the received $Sign$ against the received res (which we validated in the last step), and ultimately log the result on the console.

If any of the described checks fails, we log an error on the console and stop the RA process. Otherwise, should all of them succeed, we log that the attestation was successful.

5.5. Running the Project

The entire code is included on the disc provided with this work. In order to run it, the following is needed:

- An ATmega1284P-XPLD microcontroller. We provide no guarantees for other microcontrollers due to eventual pin differences that may raise the need for code modifications.
- A programmer in order to load the code. Our programmer of choice is the Atmel ICE (cf. Figure 5.2).

[9] For the communication, VRF utilises the pySerial library. Documentation at https://pyserial.readthedocs.io/.
[10] Documentation at https://docs.python.org/.
[11] Documentation at https://pypi.org/.
[12] GutHub repository at https://github.com/tlsfuzzer/python-ecdsa.

- A device offering a serial port and capable of executing Python code.
- A cable with USB and USB-mini connectors to power the board.

We conducted our implementation and testing on a computer with the Linux operational system. For the performance evaluation later, it is crucial to note that the machine was equipped with an Intel® Core™ i5-4570 processor with 4 CPUs, as well as 8 GB of RAM.

Figure 5.2.: The Atmel ICE.

Both the Atmel ICE and the board are connected to the computer. In addition, the board is connected to the AVR port of the programmer. Note that the computer needs to provide sufficient power for the board (1.8V).

On the computer, the code needs to be compiled by `avr-gcc` and flashed to the board. It is hence necessary that an `avr-gcc` toolchain is available and functioning. Moreover, `avrdude` needs to be installed and callable as well.

For the compilation, a Makefile with all the necessary commands is already available. Open a console in `apps/remote_attest` and execute `make all`. This will use the toolchain to build the project, and then use `avrdude` to flash it onto the board. Once this process completes, the Python VRF can be started by executing `sudo python3 verifier.py microvisor.hex <PORT> <k>`. Here, `<PORT>` refers to the serial port the microcontroller is connected to, e.g., `/dev/ttyS0` or `/dev/ttyACM0`, and `<k>` is the number of nonces to generate (for k ARs). This will start the RA process. Note that the superuser permission is needed for the Python script to access the serial port. The generated nonces are logged on the console; then, VRF also logs any eventual errors as well as a status notice in the case of success or failure.

Finally, to clean up the testbench, `make clean` can be executed.

6. Evaluation

We successfully designed and implemented the Aggregatable RA, a novel technique based on SIMPLE that aims to provide one-to-many RA capabilities to resource-constrained devices. Our implementation in the C programming language, as described in Chapter 5, was deployed on an ATmega1284P-XPLD microcontroller as a testbench. Now we will evaluate our work in the two most crucial terms – performance and security.

6.1. Performance Evaluation

Achieving better performance than classical RA architectures especially in an one-to-many scenario is one of the main goals of our work. In theory, we achieved this by increasing the overhead (by aggregating the nonces) but reducing the amount of memory digests for multiple challenges to one. In order to evaluate the performance of our approach in practice, we performed a series of tests on our implementation. In this Section, we present the results of our testing. As a base for comparison, we took the original implementation of SμV. Both the VRF and PRV performance were tested.

6.1.1. Prover Performance

In the first test, we compared the performance of our PRV with the one of SμV. In particular, we compared the time needed to process up to 10 challenges (= nonces). Since SμV is not capable of processing more than one nonce per execution, we conducted multiple runs and summed up the separate times. The time was measured as the time span between the sending of the last nonce from VRF and receiving the last part of the report. Figure 6.1 shows a chart with the resulting data.

The test shows that while our implementation is inferior when dealing with up to two challenges, it exceeds the performance of SμV in any subsequent scenarios. Namely, the run-time of our architecture remains (almost[1]) constant at 17s. SμV on the other hand exhibits linear growth with the number of nonces it needs to process, with roughly 6s being added for each further nonce. The lesser performance of our architecture in the one or two nonces scenario can be attributed to the nonce aggregation and especially the digital signature, since the latter is known to need a non-trivial amount of computational power. Later on, however, the multiple repetitions of SμV prove to be too costly in respect of

[1]Not absolutely constant. See Appendix D.

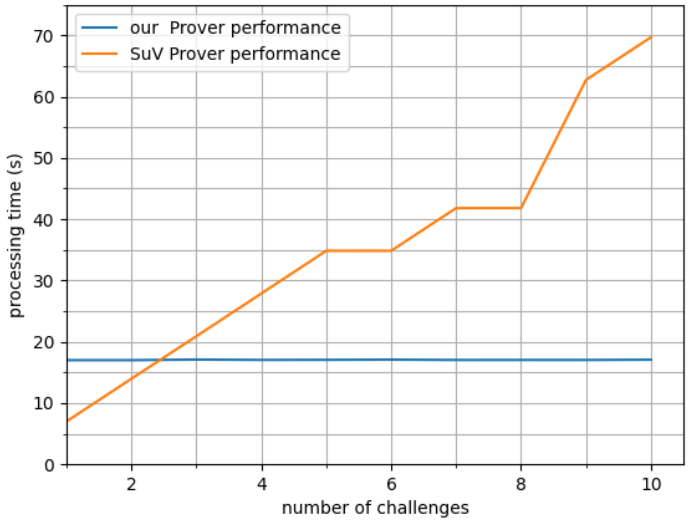

Figure 6.1.: Performance comparison of the Provers of SμV and our implementation. Measurements in seconds.

time. As this shows the general superiority regarding run-time of our implementation in comparison to SμV, in the further PRV tests we concentrated on our code.

Next, in order to get a better visualisation of the efficiency and the exact run-time of the Aggregatable RA, we continued incrementally increasing the number of nonces that were sent. For better scaling, after 20 nonces we began incrementing the input size by 10, and after reaching 250 nonces, we started incrementing by 25. The results are presented on Figure 6.2.

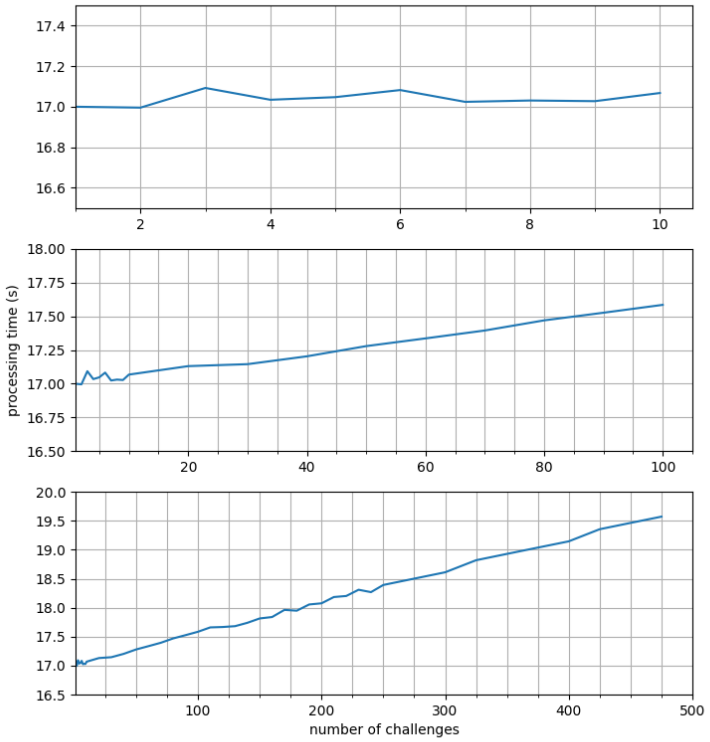

Figure 6.2.: Performance of our implementation with different amounts of nonces. Measurements in seconds.

The 18s threshold was reached with 190 nonces. While it is rather questionably realistic that any IoT device, regardless of its type and resources, needs to process a larger amount at the same time, we continued increasing the input size for the sake of completeness. The ATmega1284P-XPLD stopped responding after around 475 challenges. We assume this to be due to memory limitations since, as per design, all nonces need to be stored so that they are sent back. Even with that number of challenges, the 20s threshold was not reached. The raw data from all tests can be viewed as a table in Appendix D.

6.1.2. Verifier Performance

While our work aims to improve the efficiency of PRV, it is also important to know how our modifications interact with VRF. Many may deem the trade-off suboptimal if our approach increases the burden of VRF by a too significant amount. For this, we also conducted a set of tests on VRF, similar to the ones done on VRF. In particular, we measured the time span starting with receiving the last part of the report from PRV and ending with the check of the signature. As discussed in Chapter 4, the scenario where a single VRF needs to verify multiple reports is improbable, yet possible. Figure 6.3 shows the test results in seconds.

Similarly to what was observed with PRV, the performance of our implementation is inferior to SμV in the single challenge scenario, albeit with an insignificant amount. After that, as the SμV VRF processes the results sequentially, it gets outperformed by our implementation. However, unlike PRV, there is almost no observable difference (0.01s) when comparing the time to verify one and a couple of hundred of responses, as seen on Figure 6.4. Here, we emphasise again that the VRF performance is strongly dependent on the specifications of the machine running the Python script.

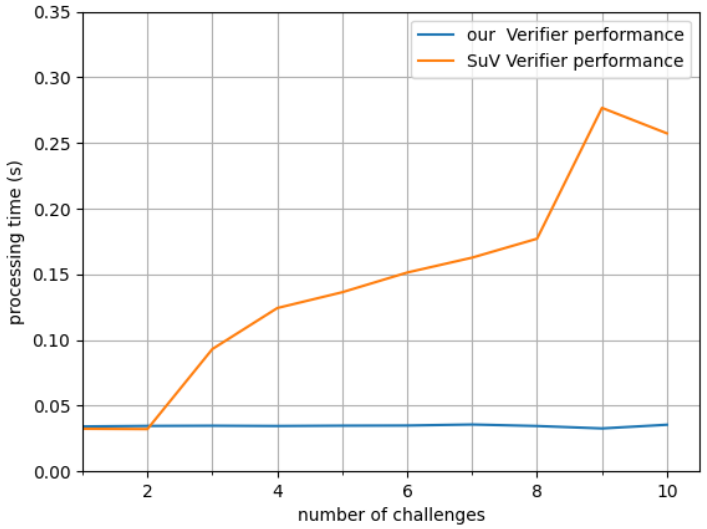

Figure 6.3.: Performance comparison of the Verifiers of SμV and our implementation. Measurements in seconds.

6.1.3. Size Comparison

Our concept is intended to apply to resource-constrained devices. Hence, besides the performance, it is crucial to also observe the size of our implementation. The most optimal way to measure it is by checking the size of the `.hex` file produced by the `avr-gcc` compiler. The `.hex` file of the original SμV has a size of 9644 bytes. However, our implementation presents an almost sevenfold increase, with 67068 bytes for the `.hex` file. It is possible that this metric can eventually be optimised. The increased size is with high probability due to the various libraries for randomness, signature and hashing which we had to add.

6.2. Security Evaluation

To the original algorithm of SIMPLE, multiple changes were made surrounding the used functions and parameters. We will now argue that our algorithm remains secure, that is, it does not allow an adversary to act as PRV. The attacker model, as described in Section 4.5, allows the adversary to eavesdrop, manipulate the network, and read/modify the contents of unprotected memory. Still, the modifications that we made are only in the RA procedure of SIMPLE. Any other functionalities and capabilities, such as memory division, etc., remain untouched.

6.2.1. Comparison to SIMPLE

We note initially that the challenge parameter(s) passed by VRF are not encrypted and thus easily available to an eavesdropping adversary. With our modifications, this equals to the list of nonces sent by the different VRFs. Hence, the nonce aggregation function needs not have the one-way property available in hashing functions to obscure the nonces. Any aggregation function from the space $\mathbb{N}^k \rightarrow \mathbb{N}$ (where k is the nonce length) will suffice. Still, to reduce implementation difficulty, we opt for a hashing function (see Chapter 5).

The second major change was utilising a keyless hashing function for the memory compression. The original SIMPLE uses a keyed HMAC-SHA1. Instead, we opt for the underlying SHA-1. We are indeed aware of the low security and preimage resistance of the SHA-1 function. Public attacks were documented in 2005 by Wang et al. [47], 2009 by Aoki et

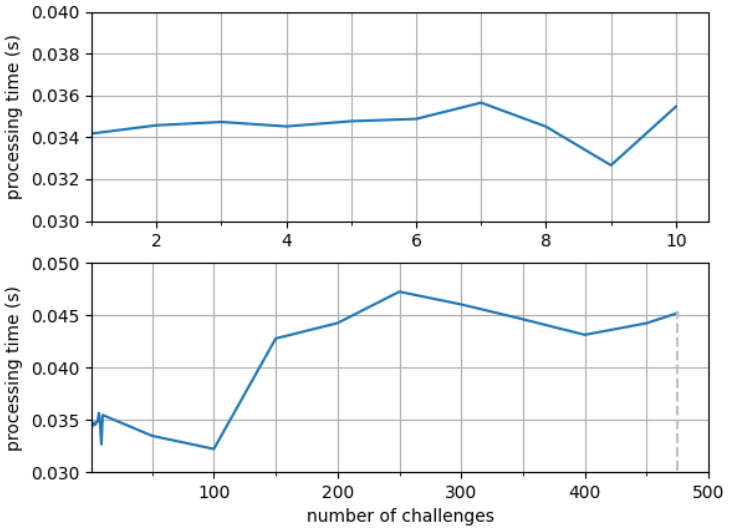

Figure 6.4.: Performance of our Prover implementation with different amounts of nonces. Measurements in seconds.

al. [48] and 2017 by Stevens et al. [49]. However, to compensate for that, we introduce a new element: the digital signature. Recall that a signature utilises a pair of asymmetric keys. This allows one to write the private key in PRV's secure memory (which is also read-protected by SμV) and send the public key (even over insecure channels) to any external entities.

An adversary cannot act as PRV. To achieve this, they would need a signature of the hash of the current nonce aggregate together with a valid state. However, the nonces differ in each AR (due to their nature as nonces) and the private key K_{priv} used for signature is safely stored. Therefore, forging such a signature is not possible. At most, the adversary can have access to a list of hashes of valid states together with a used nonce (or a nonce aggregate), as well as the signatures of those hashes[2]. This is, ultimately, not helpful.

6.2.2. Other Attack Vectors

As per our analysis, our changes to SIMPLE do not introduce any new opportunities for malicious entities or adversaries to act as PRV. However, we do remark that by removing parameters such as the synchronised clocks and the symmetric key, PRV is no longer able to authenticate VRF. This leads to no issue on PRV's side as PRV can now process a significant amount of challenges efficiently.

It is a fact that the underlying design of SIMPLE is not perfectly protected against every sort of attack. That is exemplified by a Spoofing/DDoS attack: while PRV can now process multiple ARs in parallel, this is not true for VRF. This allows an attacker to send a multitude of ARs to the same or different devices (or the corresponding brokers), impersonating the victim. They will all be processed, and the victim will be flooded with attestation results.

The reason for this is the simplistic nature of the challenge parameters in our case (a simple nonce), meaning that the sender of an AR is in fact not verified to PRV. PRV will handle all ARs it receives. We do however deem the construction of a protection mechanism against this sort of attacks rather non-trivial, and therefore also not suited for

[2]Refer to Appendix A for a more detailed overview.

a Bachelor's Thesis. Hence, we leave it out of the scope of our work, and denote it as a possible point for future research.

7. Conclusion

Being able to prove oneself's trustworthiness is crucial nowadays in the IoT sector due to the rising amount of high-impact attacks from adversaries. RA is well-known as a proven, established mechanism for achieving this goal. Unfortunately, the RA approach also has one major drawback in the sense of its scalability, as its classical version is an one-to-one process between VRF and PRV. This reduces its fitness for all scenarios where a PRV entity may need to process multiple ARs at once, and thus for the IoT sphere in particular. Notably affected by this limitation are low-end entities that otherwise do not offer security mechanisms out-of-the-box.

In our work, we designed the so-called Aggregatable RA, novel in that it converts the RA process from one-to-one to one-to-many. There, PRV will not process every single AR sequentially, but rather construct a challenge aggregate and work with it instead. The produced result will then be verifiable to all VRF entities. With this, we aim to improve the efficiency of RA while still maintaining its security and full functionalities. To the best of our knowledge, that is the first architecture with this crucial property, such that it is also applicable to resource-constrained IoT devices.

For the design, we chose to extend a currently existing RA architecture to make it provide the aforementioned properties. In order to achieve this, we first had to analyse the state of the art in this area. After presenting an overview of a sample of such techniques, we found that many of them also provide additional unique properties such as healing, formal verifiability, etc. We evaluated those in order to select an optimal one to build upon. Our final choice would ultimately lie with SIMPLE, a relatively new hybrid RA technique that allows for dynamic application loading and unloading and can also be verified formally.

We afterwards described an one-to-many RA protocol based on SIMPLE that introduced three major changes. The first one was the nonce aggregation: after receiving a set of challenge parameters (nonces), we constructed an aggregate from them as discussed above. Then, we observed that SIMPLE uses a keyed HMAC-SHA1 function for compressing the memory. To eliminate the need for symmetric keys, we replaced it with a simple keyless compression function. The final step was to introduce digital signature of the result computed in step 2. At the same time, we also adjusted the VRF process to make it accept the new report. VRF would assert that its own nonce is contained in the aggregate, and then check the validity of the state by consulting a remote database (or, in the case of SIMPL, a BC). Finally, VRF will use PRV's public key to confirm the digital signature's validity and authenticity. The whole process is, once again, visualised as a sequence diagram on Figure 4.7.

In the next phase, we proceeded to implement our approach as a PoC. In particular, we opted for an ATmega1284P-XPLD board as our testbench. This is an 8-bit microcontroller with very limited memory, so it falls in the device category we target. The SIMPLE architecture provides the code of its core component, the Security MicroVisor SμV, freely on GitHub, allowing us to extend or adapt it at will. From the modifications described above, we opted for SHA-1 as both the nonce aggregation and the memory compression function. We defend this decision and base it primarily on the SHA-1 code being already available with the SμV code, due to the original usage of HMAC-SHA1. Moreover, to provide compatibility with the BC of SIMPL, we selected a library with ECDSA capabilities such that it utilises the secp256k1 curve. The VRF code was also changed to reflect the process we discussed in the previous paragraph.

Finally, we presented an evaluation of the Aggregatable RA. After extensive testing, it was shown that it exceeds the performance of the original SμV in all but a negligible amount of cases. This holds for both VRF and PRV. In particular, we conducted testing with up to 475 nonces and observed a slowdown of only around 10% in comparison to a single nonce. The processing times of PRV thus ranged from 17 to 19.57s. In contrast to that, the original SμV code exhibits a linear growth in relation to the number of nonces. In terms of security, we argued that the probability of an adversary being able to act as PRV is very limited as it relies on them finding a collision with very specific properties. Since the only potentially unsafe choice is the SHA-1, one could simply opt for another keyless compression function, such as the SHA-256 or SHA-512. Regardless of the choice, it holds that conducting a successful attack would have a negligible probability as long as the utilised compression function is known to be collision resistant.

Still, there exist some use cases which have not been explicitly taken care of in this thesis as they are only indirectly connected to the Aggregatable RA. For instance, the original SIMPLE architecture could be formally verified and proven as secure. We deem the proof process, however, to be non-trivial and hence better suited as a separate topic of future research. Similarly, it is possible to conduct correct and secure RA with SIMPLE even after (un-)loading applications during run-time. We see no reason why this should not be the case with our RA protocol anymore, but it has also not been inspected closely. Therefore we provide no guarantee for the correct functionality in this scenario, and leave the inspection for other works. Finally, eventual research can be made in the future to try and remove some currently persisting attack vectors, such as the danger of a spoofing or a DDoS attack.

Overall, our work shows that constructing a RA architecture for resource-constrained IoT devices that allows them to process multiple ARs simultaneously is by no means impossible. Quite the contrary, it can be easily designed in theory and implemented in practice by simply modifying an already existing RA technique (in our case, SIMPLE). We consider this to be a result of high importance in the IoT sector. We do believe that the insights gained from our work, but also the provided implementation which proves the applicability of our idea, open the door for potential researchers in the future, paving the way for them to perfectionate the described concept even further.

List of Figures

List of Tables

Listings

Acronyms

IoT Internet of Things

SIMPL Secure Internet of Things Management PLatform

TCG Trusted Computing Group

TPM Trusted Platform Module

BC Blockchain

RA Remote Attestation

VRF Verifier

PRV Prover

BKR Broker

AR Attestation Request

PKI Public Key Infrastructure

PoC Proof-of-Concept

SμV Security MicroVisor

OS Operational System

ROM Read-Only Memory

ISR Interrupt Service Routine

MMU Memory Management Unit

MPU Memory Protection Unit

IPC Inter-Process Communication

MitM Man-in-the-Middle

DoS Denial of Service

DDoS Distributed Denial of Service

ECDSA Elliptic Curve Digital Signature Algorithm

PRNG Pseudo-Random Number Generator

Bibliography

[1] B. Herzberg, I. Zeifman, and D. Bekerman, "Breaking down mirai: An iot ddos botnet analysis." https://www.imperva.com/blog/malware-analysis-mirai-ddos-botnet/. Accessed: 2020-05-20.

[2] M. Ammar, B. Crispo, and G. Tsudik, "Simple: A remote attestation approach for resource-constrained iot devices," in *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 247–258, IEEE, 2020.

[3] C. Mitchell, *Trusted computing*, vol. 6. Iet, 2005.

[4] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *Proceedings of the Ninth European Conference on Computer Systems*, pp. 1–14, 2014.

[5] M. Banerjee, J. Lee, and K.-K. R. Choo, "A blockchain future for internet of things security: A position paper," *Digital Communications and Networks*, vol. 4, no. 3, pp. 149–160, 2018.

[6] J. Vijayan, "Stuxnet renews power grid security concerns." https://www.computerworld.com/article/2519574/stuxnet-renews-power-grid-security-concerns.html. Accessed: 2020-05-20.

[7] "February 28th ddos incident report." https://github.blog/2018-03-01-ddos-incident-report/. Accessed: 2020-11-09.

[8] "Secure iot management platform. overview." http://simpl-project.de/. Accessed: 2020-05-20.

[9] M. U. Farooq, M. Waseem, S. Mazhar, A. Khairi, and T. Kamal, "A review on internet of things (iot)," *International journal of computer applications*, vol. 113, no. 1, pp. 1–7, 2015.

[10] K. Ashton *et al.*, "That 'internet of things' thing," *RFID journal*, vol. 22, no. 7, pp. 97–114, 2009.

[11] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," *CISCO white paper*, vol. 1, no. 2011, pp. 1–11, 2011.

[12] "Number of internet of things (iot) connected devices worldwide in 2018, 2025 and 2030 (in billions)." https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/. Accessed: 2020-11-07.

[13] S. Pearson, "Trusted computing platforms, the next security solution," *HP Labs*, 2002.

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

[15] T. C. Group *et al.*, "Tpm main part 1 design principles," *TCG White Paper*, 2011.

[16] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, 2011.

[17] M. Alam, T. Ali, S. Khan, S. Khan, M. Ali, M. Nauman, A. Hayat, M. Khurram Khan, and K. Alghathbar, "Analysis of existing remote attestation techniques," *Security and Communication Networks*, vol. 5, no. 9, pp. 1062–1082, 2012.

[18] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Hydra: hybrid design for remote attestation (using a formally verified microkernel)," in *Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks*, pp. 99–110, 2017.

[19] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1–12, 2013.

[20] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 400–409, 2009.

[21] S. Hristozov, J. Heyszl, S. Wagner, and G. Sigl, "Practical runtime attestation for tiny iot devices," in *Proceedings of the 2018 Workshop on Decentralized IoT Security and Standards, San Diego, CA, USA*, vol. 18, 2018.

[22] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 384–391, IEEE, 2017.

[23] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 743–754, 2016.

[24] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "Lo-fat: Low-overhead control flow attestation in hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[25] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "Darpa: Device attestation resilient to physical attacks," in *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pp. 171–182, 2016.

[26] W. Yan, A. Fu, Y. Mu, X. Zhe, S. Yu, and B. Kuang, "Eapa: Efficient attestation resilient to physical attacks for iot devices," in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pp. 2–7, 2019.

[27] F. Kohnhäuser, N. Büscher, S. Gabmeyer, and S. Katzenbeisser, "Scapi: a scalable attestation protocol to detect software and physical attacks," in *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pp. 75–86, 2017.

[28] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "Tytan: tiny trust anchor for tiny devices," in *Proceedings of the 52nd Annual Design Automation Conference*, pp. 1–6, 2015.

[29] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pp. 479–498, 2013.

[30] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for iot devices," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–33, 2017.

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

[32] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.*, pp. 272–282, IEEE, 2004.

[33] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, pp. 1–16, 2005.

[34] M. Ammar, "Sμv - the security microvisor: A formally-verified software-based security architecture for the internet of things." `https://github.com/m3mmar/verified_SuV`. Accessed: 2020-05-30.

[35] X. Carpent, N. Rattanavipanon, and G. Tsudik, "Remote attestation of iot devices via smarm: Shuffled measurements against roving malware," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 9–16, IEEE, 2018.

[36] "The sel4 microkernel." `https://sel4.systems/About/`. Accessed: 2020-05-30.

[37] A. Ibrahim, A.-R. Sadeghi, and G. Tsudik, "Healed: Healing & attestation for low-end embedded devices," in *International Conference on Financial Cryptography and Data Security*, pp. 627–645, Springer, 2019.

[38] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "Formally verified hardware/software co-design for remote attestation," *arXiv preprint arXiv:1811.00175*, 2018.

[39] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "{APEX}: A verified architecture for proofs of execution on remote devices under full software compromise," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[40] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems," in *International Conference On Computer Aided Design*, 2019.

[41] X. Carpent, G. Tsudik, and N. Rattanavipanon, "Erasmus: Efficient remote attestation via self-measurement for unattended settings," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1191–1194, IEEE, 2018.

[42] E. Dushku, M. M. Rabbani, M. Conti, L. V. Mancini, and S. Ranise, "Sara: Secure asynchronous remote attestation for iot systems," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3123–3136, 2020.

[43] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, "Elements of reusable object-oriented software," *Reading: Addison-Wesley*, 1995.

[44] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," *Internet Engineering Task Force (IETF): Fremont, CA, USA*, pp. 2070–1721, 2014.

[45] "Atmega1284p datasheet." `https://ww1.microchip.com/downloads/en/DeviceDoc/doc8059.pdf`. Accessed: 2020-11-27.

[46] H. Mayer, "Ecdsa security in bitcoin and ethereum: a research survey," *CoinFaabrik, June*, vol. 28, p. 126, 2016.

[47] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full sha-1," in *Annual international cryptology conference*, pp. 17–36, Springer, 2005.

[48] K. Aoki and Y. Sasaki, "Meet-in-the-middle preimage attacks against reduced sha-0 and sha-1," in *Annual International Cryptology Conference*, pp. 70–89, Springer, 2009.

[49] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full sha-1," in *Annual International Cryptology Conference*, pp. 570–596, Springer, 2017.

Appendix

A. Adversary Knowledge after SIMPLE Modifications

Let

- $l =_{def}$ the length of a single nonce in bytes
- $state =_{def}$ PRV's current memory
- $N =_{def} \{n_1, ..., n_k\}$ for nonces n_i where $i \in \{1...k\}$
- $n' =_{def} aggregate(N)$ where $aggregate$ is from the space $\mathbb{N}^{k \times l} \rightarrow \mathbb{N}^l$
- $res =_{def} compress(state, n')$
- $Sign =_{def} sign_{K_{priv}}(res)$
- $report =_{def} \{Sign, res, N\}$

From eavesdropping, an adversary A can learn the following:

- N
- a compression res of a valid $state$ with the current N
- the $Sign$ corresponding to this res

By recording multiple RA sessions, A creates a list of each of those elements. In order to impersonate PRV, A needs to forge a valid $report$, i.e., a valid $state$ compressed with the *current* nonce (aggregate), as well as the signature of that compression. We assume that $aggregate$ and $compress$ may not have the one-way property, or collision attacks may have been shown.

Let the current list of nonces be N_{new}. Can A can compute the following:

- N_{new}: yes, it is available directly.
- n'_{new}: yes, by applying $aggregate$ to N_{new}.
- a valid $Sign_{new}$: only if $res_{new} = res_x$ for some res_x from the list mentioned above. Signatures cannot be created without K_{priv}.
- a valid $res_{new} = compress(some\_state, n'_{new})$: ?

Analysis: *compress* is not necessarily an one-way function, so we assume A can find some random $state_x$ and n'_x such that $compress(state_x, n'_x) = res_x$. However, in our case, $n'_x = n'_{new}$ is fixed. Depending on the exact *compress*, the probability of finding such a specific *state* may be negligible. Recall that A also does not know any valid, uncompressed *state*-s. As a conclusion, it cannot forge a valid res_{new} including n'_{new} that is also equal to some old res_x in order to use the corresponding $Sign_x$. Thus, A cannot act as PRV.

B. Custom List Implementation in C

```c
#include <stdint.h>

#define NONCELEN sizeof(uint8_t)*20

#ifndef LIST_H
#define LIST_H

typedef struct node {
    uint8_t * val;
    struct node * next;
    struct node * prev;
} node_t;

node_t * l_create(const uint8_t * val);

void l_append(node_t * head, const uint8_t * val);

void l_prepend(node_t ** head, const uint8_t * val);

uint8_t * l_pop(node_t ** head);

uint8_t * l_truncate(node_t * head);

int l_len(node_t * head);

uint8_t * l_extract(node_t ** head, int n);

void l_delete(node_t * head);

#endif
```

Listing 7.1: Full header of the custom list data structure.

C. Data Format in the Aggregatable RA

The data is transferred byte-wise over a serial port. Since both the challenge and the response consist of multiple parts, special delimiters are utilised to differentiate between them, as well as to denote the end of the transmission.

Challenge delimiters:

- `','` (or `0x2c`) between each pair of nonces.
- `'.'` (or `0x2e`) at the end of the nonce list.

Response delimiters:

- `';;'` (or `0x3b, 0x3b`) between each part of the response (signature/hash/nonce list).
- `','` between each pair of nonces from the nonce list.
- `'..'` at the end of the nonce list.

The nonces are deliberately generated in such a way that they do not contain any of those characters, as well as some ASCII control characters (`0x0a, 0x0d`) that may interfere with data parsing. Namely, we simply keep generating a new nonce with `os.urandom(20)` until it no longer contains any of them.

This results in the following data formats:

- For the challenge:

```
n(1 1)n(1 2)...n(1 20) , ... , n(k 1)n(k 2)...n(k 20) .
```

- For the response:

```
Sign(1)...Sign(64) ;; res(1)...res(20) ;;
    n(1 1)...n(1 20) , ... , n(k 1)...n(k 20) ..
```

Since the bytes/characters produced by the hash function and the signature algorithm are impossible to control, some of the delimiter or control characters we had forbidden in the nonces may occur in `Sign` or `res`. This may eventually lead to an unparseable response. However, in our tesing experience, this happens relatively seldom.

D. Raw Testing Data

Here, we present the raw data produced by our testing of the VRF and PRV code of our implementation and of SμV. For the VRF entities, we considered the time from the receiving of the last part of the report to the end of the attestation. For the PRV entities, we measured the time inside VRF between sending the last nonce and receiving the first part of the report. For the SμV implementation, we summed up the corresponding times for i executions, and for our code, we simply sent i nonces to be processed. All measurements are in seconds. A "-" indicates that the measurement was not made.

Nonces	Our PRV	SμV PRV
1	17.00004482269287	6.999865770339966
2	16.99561834335327	13.967623233795166
3	17.092677354812622	20.89748740196228
4	17.03445029258728	27.866657495498657
5	17.047333002090454	34.841463804244995
6	17.08233141899109	34.842437744140625
7	17.02419090270996	41.802367210388184
8	17.03079390525818	41.80990028381348
9	17.027580976486206	62.71474742889404
10	17.067793369293213	69.67570805549622
11	17.035887002944946	76.65568685531616
12	17.13425850868225	83.60884690284729
13	17.114075660705566	83.61706876754761
14	17.11038851737976	97.53844952583313
15	17.106674194335938	97.5536961555481
20	17.13041591644287	-
30	17.144960165023804	-
40	17.202955961227417	-
50	17.278524160385132	-
60	17.335693359375	-
70	17.394194841384888	-
80	17.46912956237793	-
90	17.52569055557251	-
100	17.584004640579224	-
110	17.65950608253479	-
120	17.666281700134277	-
130	17.680712461471558	-
140	17.738770961761475	-
150	17.81421732902527	-
160	17.838453769683838	-
170	17.961848258972168	-
180	17.948129415512085	-
190	18.055432081222534	-
200	18.07551884651184	-
210	18.183959245681763	-
220	18.20200538635254	-
230	18.30968403816223	-
240	18.268444061279297	-
250	18.39233946800232	-
275	18.501898527145386	-
300	18.611114978790283	-
325	18.819324016571045	-
350	18.92877697944641	-
375	19.037516355514526	-
400	19.145596504211426	-
425	19.354893922805786	-
450	19.464022636413574	-
475	19.572144746780396	-

Table D.1.: Performance comparison of both Provers in seconds.

Nonces	Our VRF	SμV VRF
1	0.0341792106628418	0.03239941596984863
2	0.034575462341308594	0.03215432167053223
3	0.03473520278930664	0.0930938720703125
4	0.034523725509643555	0.12431907653808594
5	0.03477191925048828	0.13636088371276855
6	0.03487992286682129	0.1513535976409912
7	0.03565573692321777	0.16269826889038086
8	0.03451204299926758	0.17711472511291504
9	0.03266406059265137	0.2767360210418701
10	0.035470008850097656	0.25733399391174316
50	0.033478736877441406	-
100	0.032219886779785156	-
150	0.04277539253234863	-
200	0.044252634048461914	-
250	0.047249555587768555	-
300	0.046036720275878906	-
350	0.0446171760559082	-
400	0.04313254356384277	-
450	0.04423999786376953	-
475	0.04521036148071289	-

Table D.2.: Performance comparison of both Verifiers in seconds.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 27. December 2020

.....

(Vasil Alistarov)