SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices

Mahmoud Ammar
imec-DistriNet, KU Leuven
mahmoud.ammar@cs.kuleuven.be

Bruno Crispo
imec-DistriNet, KU Leuven
University of Trento, Italy
bruno.crispo@unitn.it

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

***Abstract*—Remote Attestation (RA) is a security service that detects malware presence on remote IoT devices by verifying their software integrity by a trusted party (verifier). There are three main types of RA: software (SW)-, hardware (HW)-, and hybrid (SW/HW)-based. Hybrid techniques obtain secure RA with minimal hardware requirements imposed on the architectures of existing microcontrollers units (MCUs). In recent years, considerable attention has been devoted to hybrid techniques since prior software-based ones lack concrete security guarantees in a remote setting, while hardware-based approaches are too costly for low-end MCUs. However, one key problem is that many already deployed IoT devices neither satisfy minimal hardware requirements nor support hardware modifications, needed for hybrid RA.**

This paper bridges the gap between software-based and hybrid RA by proposing a novel RA scheme based on software virtualization. In particular, it proposes a new scheme, called SIMPLE, which meets the minimal hardware requirements needed for secure RA via reliable software. SIMPLE depends on a formally-verified software-based memory isolation technique, called Security MicroVisor (SμV). Its reliability is achieved by extending the formally-verified safety and correctness properties to cover the entire software architecture of SIMPLE. Furthermore, SIMPLE is used to construct SIMPLE+, an efficient swarm attestation scheme for static and dynamic heterogeneous IoT networks. We implement and evaluate SIMPLE and SIMPLE+ on Atmel AVR architecture, a common MCU platform.

I. Introduction

The Internet of Things (IoT) is formed by increasing deployment of interconnected embedded devices that touch many aspects of modern life. The vast majority of such devices lack security features (e.g. trusted boot, virtualization) that are present in modern general-purpose computing platforms, creating a large new attack surface for malware, exemplified by Stuxnet [1] and the Mirai Botnet [2]. To mitigate such attacks, it is important to monitor the behavior of embedded devices and detect any malware presence as early as possible. For this purpose, various detection protocols have been proposed.

Remote Attestation (RA) has emerged as an important means to detect the misbehavior of a compromised IoT device. It allows a trusted entity, called a verifier, to securely check the internal state of a remote untrusted device, called a prover. In the last decade, numerous RA protocols with different assumptions and security guarantees have been proposed, first for the single-prover scenario [3]–[11], and, more recently, for groups or swarms [12]–[15]. Early work focused on either pure software (SW)-based [3], [4], [16]–[18] or fully hardware (HW)-based techniques [10], [11]. The latter are relatively expensive as they depend on specialized hardware, i.e. a Trusted Platform Module (TPM) [19], and are thus not suitable for low-end embedded devices.

SW-based RA techniques have been proposed for embedded devices that lack any hardware security features. They rely either on stricttight time constraints [4] or lack of free space to store malicious code [3]. Some subsequent results have shown that SW-based RA techniques depend on assumptions that are very hard to achieve in practice and are still vulnerable to some attacks (i.e. Return-oriented programming (ROP), or Time-Of-Use to Time-Of-Check (TOCTOU)) [20]. Furthermore, SW-based RA techniques are limited to one-hop communication between prover and verifier. To fill the sizeable gap between SW- and HW-based RA techniques, hybrid RA techniques have been proposed. They aim to impose a minimal set of hardware security features to support efficient and secure RA on embedded devices [5]–[7], [9]. All current hybrid RA techniques require prover to have (or support adding) minimal hardware features [21], such as Read-Only Memory (ROM) and Memory Protection Unit (MPU).

Problem Statement. Today, various resource-constrained embedded devices with different resources and capabilities are being connected to the Internet. Securing these devices is extremely important as they play pivotal roles in many application domains, i.e. industry 4.0. The Internet Engineering Task Force (IETF) identifies Class-1 IoT devices with 10 kB of RAM and 100 kB of Flash memory as having the minimal resources necessary to communicate securely over the Internet [22]. A broad range of such devices, including Micaz [23] and MicroPnP IoT platform [24], are widely used in many IoT application domains. However, they do not offer the minimal hardware features needed for hybrid RA. Furthermore, they do not support any hardware modification or extension to their underlying architecture. Consequently, despite their massive deployment, Class-1 IoT devices are still insecure.

Contributions. We propose a novel RA approach that fills the gap between SW-based and hybrid RA techniques. We present SIMPLE, a provably secure hypervisor-based RA scheme for resource-constrained IoT devices, exemplified by IETF Class-1. SIMPLE leverages and extends an open-source

2642-9500/20/$31.00 ©2020 IEEE
DOI 10.1109/ICCPS48487.2020.00036

formally-verified software-based security architecture, called the Security MicroVisor (SμV) [25], which provides trusted MPU-like memory protection. We apply formal verification to guarantee that the entire software architecture of SIMPLE is memory-safe and crash-free. Our goal is to provide the same security guarantees as hybrid RA without hardware modifications, against remote-only attacks.

Main contributions of this paper are:

- In constructing SIMPLE, we relax prior requirements (imposed by hybrid RA) of having ROM and MPU [21] i.e., SIMPLE does not require a hardware-based MPU and does not even depend on the availability of ROM. To the best of our knowledge, SIMPLE is the first RA scheme that provides guaranteed security properties along with efficiency to IETF Class-1 devices, without requiring hardware support.
- We propose SIMPLE+, a lightweight swarm attestation scheme, that extends SIMPLE, for static and dynamic networks, consisting of multiple heterogeneous IoT devices.
- We provide a proof-of-concept implementation and extensive evaluation of SIMPLE(+) on two AVR-based Class-1 IoT platforms, showing its robustness and efficiency.

Organization. The remainder of this paper is organized as follows: Section II reviews the related work. Preliminaries are presented in Section III. Section IV describes SIMPLE in detail, followed by Section V which describes SIMPLE+. Implementation details and evaluation are discussed in Section VI. Section VII concludes the paper, and Appendix A illustrates formally verified properties and their impact on the security of SIMPLE(+).

II. Related Work

Single Device Attestation. Remote attestation (RA) is a security service that enables a trusted party (verifier) to measure the current internal state (i.e. RAM, flash, etc.) of an untrusted remote device (prover) and thus determine whether it has been compromised. Thereby, RA helps the verifier establish a static or dynamic root of trust in the prover. Prior work in single-prover RA falls into hree categories: hardware-based [10], [11], software-based [3], [4], [16]–[18], and hybrid [5]–[9]. HW-based RA techniques [10], [11] depend on establishing a trust anchor on the prover to guarantee the integrity of the attestation code. Typically, the trust anchor is implemented in hardware, i.e., a Trusted Platform Module (TPM) [19]. However, TPMs and other similar HW-based modules are complex and too expensive for low-end embedded devices. SW-based RA techniques [3], [4], [16]–[18] target highly resource-constrained and legacy devices that lack any hardware security features. They assume an ideal environment with reliable communication for exchanging messages, in the presence of a silent (during attestation) remote attacker. As mentioned earlier, SW-based techniques lack concrete security guarantees due to assumptions that are hard to achieve in real-world scenarios, i.e., considering only passive remote attacks, communication without failure or variable delay, and one-hop prover-verifier communication. Hybrid RA techniques [5]–[9] aim to bridge the gap between SW- and HW-based ones. They employ low-cost simple hardware support (e.g., ROM and MPU) on the prover to guarantee the integrity of attestation code and confidentiality of secret keys. They require modifying or extending the hardware of the prover with minimal hardware features, thus increasing the prover's overall cost. To this end, each approach has pros and cons, as discussed in Section I and extensively considered in [26].

Swarm/Group Attestation. Swarm attestation schemes [12]–[15], [27], [28] enable scalable attestation of large groups of embedded devices. These schemes differ in various ways, e.g., methodology followed (tree-based or distributed), topology (dynamic or static), and type of cryptography. SEDA [12] is the first swarm attestation scheme. By using public-key cryptography, SEDA allows neighbour devices to attest each other and then securely aggregate their attestation reports up the spanning tree until the final report is received by the verifier. SANA [13] improves on SEDA by allowing public verifiability of attestation reports and minimizing verification overhead incurred by SEDA by emplpoying a novel aggregate signature scheme. DARPA [28] extends SEDA with an heartbeat-based absence detection scheme to detect physical attacks. SCAPI [27] enhances DARPA by efficiently updating swarm-wide device secret keys and thus detecting physical attacks with a reduced communication overhead, assuming that absent (physically attacked) devices do not receive updates. slimIoT [15] further improves SCAPI in terms of communication overhead and memory footprint by relying on an efficient broadcast authentication scheme using symmetric keys. WISE [14] is a smart swarm RA scheme that deals with heterogeneous devices; it depends on a resource-efficient intelligent broadcast authentication technique based on the Hidden Markov Model. This minimizes communication overhead depending on the tolerated latency of detecting (unlikely) compromised devices w.r.t. the differences and requirements of connected devices in a swarm.

Formally-verified RA. The work in [29] has made the first step towards formal verification of hardware security requirements of RA architectures. Some subsequent research has been proposed to automate verifying hybrid RA properties [30]. Nevertheless, none of the aforementioned research has yielded a fully verified design of RA. HYDRA [31] has followed a different direction by proposing a hybrid design for RA using seL4 [32], a formally verified microkernel. This makes it only suitable for high-end devices. Furthermore, HYDRA does not formally verify hardware modifications or software implementation of attestation code. VRASED [33] is the first fully verified hybrid RA co-design for embedded devices. It aims to provide a secure and sound RA scheme by guaranteeing the correct design and implementation of RA security properties.

Control flow integrity. All aforementioned single-prover and swarm attestation techniques are static and only check whether benign software is still running on the prover device

as initially loaded. This means that such static attestation techniques only ensure the integrity of binaries and not of their execution. Thus, they are vulnerable to runtime attacks that hijack the application's control or data flow. Control-flow integrity (CFI) is a defense mechanism against runtime attacks that reuse the existing code by hijacking the control flow of a program to cause unintended, malicious program behavior [34]. Such attacks have been demonstrated on various platforms and devices, including Class-1 ones [35]. As a countermeasure, several control (and data) flow integrity schemes have been proposed [36]–[38]. The main goal of all CFI schemes is to enforce that a program's control flow behaves as the developer-intended flow. The integrity of the program is maintained by validating for each control flow decision whether the executed path lies within the program's control flow graph. In short, all CFI schemes are complementary to all static RA techniques in the sense that they attest runtime behavior, which is orthogonal to software binaries attestation.

SIMPLE. To this end, we are unaware of any work that provides a verified, reliable, and scalable RA scheme to Class-1 IoT devices without hardware modification. To the best of our knowledge, SIMPLE(+) is the first RA scheme that provides guaranteed security properties to this class of devices, w.r.t remote-only attacks. Furthermore, CFI is implicitly guaranteed in our scheme as explained in the following sections.

III. Preliminaries

The Security MircoVisor (SμV) [39]. SμV is an open-source software-based memory isolation hypervisor that uses selective software virtualisation and assembly-level code verification to isolate a software-based Trusted Computing Module (TCM) from untrusted application software. SμV targets simple microcontrollers that lack MPUs, support global interrupt disabling, are single-threaded, and have sufficient non-volatile memory (e.g. Flash or ROM). The latter is a key hardware requirement in all MCUs, including the ones considered as Class-1 IoT devices [22]. Therefore, we do not consider it as an explicit hardware requirement needed for secure RA. In case of having a Flash memory, SμV turns part of it into a *virtual* ROM to ensure immutability. Furthermore, SμV is aimed at devices that either physically disable, or lack, Direct Memory Access controller (DMA), which accounts for the majority -if not all- of Class-1 IoT devices [22].

SμV provides memory protection by reserving part of the memory for the TCM. This software is installed prior to the deployment of the IoT device using a physical programming device, i.e. JTAG. The remainder of memory refers to the application software and its data. The TCM memory is subject to no restrictions, whereas other parts of memory are strongly restricted as shown in Figure 1, where Data memory holds only data and cannot execute, read, or write any instruction inside the entire memory. Instruction memory can read and write Data memory and Memory Mapped IO (MMIO), jump within its own logical domain, and execute instructions within its own logical domain or from specific entry points in the TCM memory. Access rights are also illustrated in Table I.

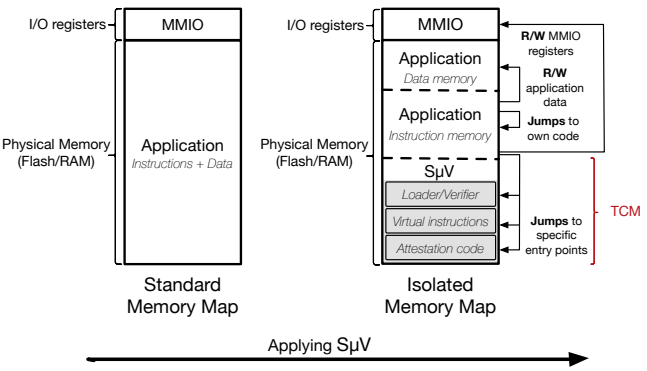


Figure 1: Standard insecure (left) and secure SμV-based (right) memory map.

Table I: Access rights enforced by the Security MicroVisor.

	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.

Restrictions on application code are enforced at the instruction level through two basic mechanisms: (i) application deployment may only occur through SμV, where incoming applications are verified by SμV (TCM) at load-time to ensure that they adhere to the aforementioned rules, and (ii) certain inherently unsafe instructions which are nonetheless essential for normal operation are replaced by safe virtualized instructions through a modified toolchain which substitutes all unsafe dynamic instructions with calls to their secure virtualized equivalents, stored in the TCM memory. The security properties, i.e. strong isolation of SμV, are maintained even when adversaries use their own toolchain or write hand-crafted assembly because load-time verification of applications is done by SμV on the embedded device itself [25]. Applications that contain unsafe instructions are rejected outright. SμV considers an instruction to be *unsafe* if it attempts, in any way, to compromise its memory. Thus, the installed application could be buggy but not malicious (safe). Furthermore, the untrusted application can still write into its own instruction memory with the help of SμV, by invoking the loader/verifier function in the TCM, after placing the entire binary in the non-executable Data memory. The loader/verifier function writes these instructions, even if they are buggy, to the instruction memory of application if they are considered safe w.r.t SμV memory. Thus, dynamic root of trust is still required, motivating the need for SIMPLE.

SμV has been formally verified to be memory-safe and crash-free [39]. As such, considering the verified properties, SμV does not exhibit any undefined behavior as described by the C11 standard [40].

IV. SIMPLE

RA is realized as an interactive protocol, whereby a trusted entity, denoted as verifier, checks the software integrity of

an untrusted remote entity, denoted as prover. The complete set of properties of secure RA has been proposed in [21] and formally verified in [33] to prove that their conjunction implies a sound and secure RA. We first outline our attacker model and describe the design rationale of SIMPLE. Then, we explain how the properties of secure RA are met in SIMPLE by formally verifying certain safety and correctness properties of the entire software architecture, and without relying on the minimal hardware features proposed in [21]. SIMPLE+ is then proposed as an extension of SIMPLE for swarms. Henceforth, we refer to the verifier as υ and the prover as ρ.

A. Adversary model

We consider only remote software-based attacks. Thus, we assume that the adversary has full access to the network and can remotely control the entire software state of the MCU. She can either perform passive (e.g. eavesdrop on communications, or read unprotected memory area) or active (e.g. modify existing code, or inject malware) attacks. Similar to most prior RA schemes, we rule out all kinds of physical, and side-channel attacks [1]. We only consider DMA-free IoT devices (the majority of Class-1 devices are DMA-free).

B. Design rationale of SIMPLE

SIMPLE is a single-prover attestation scheme that is built atop SμV. It is depicted in Figure 2 and realized as follows:

- ρ is initialized with some secret data by a trusted network operator, υ. Prior to deployment of ρ, υ and ρ share two secret keys, K_{auth} and K_{attest}, used for authentication and attestation purposes, respectively. Each one also maintains a counter, $C_\bullet$, initialized to the same value. All keys and counters are stored in a secure memory area (SμV memory).
- Whenever needed, υ sends an attestation request containing: (i) a monotonically updated value of the counter, (ii) a freshly generated *Nonce*, (iii) a valid software state of ρ's memory (VS), and (iv) a keyed-hash message authentication code (HMAC) computed over (i)-(iii) values, using K_{auth}. The counter and *Nonce* guarantee freshness and avoid replay attacks [2]. VS represents the HMAC of static and dynamic memory contents (e.g. Flash, RAM, and registers) at a certain time, computed using K_{attest}.
- Once the attestation request is received, ρ checks whether the sequence number is larger than the stored one. If so and if the entire request is authenticated successfully, ρ computes an HMAC (VS') of its current state using K_{attest}. Then, ρ composes an attestation report consisting of a K_{auth}-based HMAC over C_P, $Nonce$, and 0/1 – a binary outcome flag indicating whether VS matches VS' (1 in case of a match, and 0 otherwise). The attestation report is then sent to υ.
- After receiving and authenticating the attestation report, υ considers ρ to be in a good state if the report value is "1". Otherwise, a compromise is reported and some actions are taken, i.e. secure erasure [41].

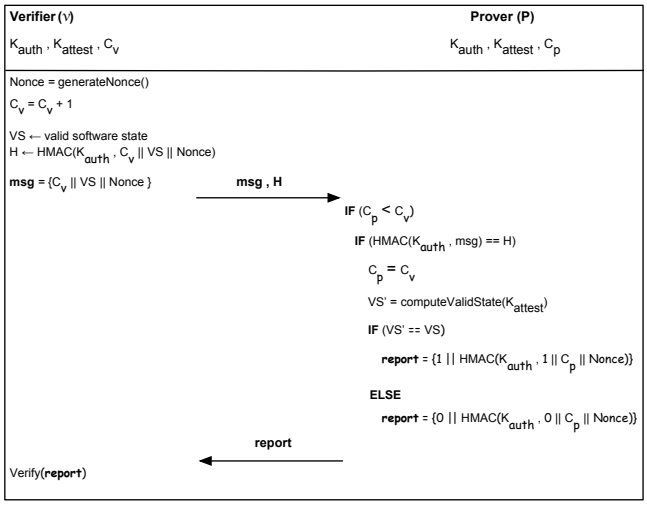


Figure 2: An overview of SIMPLE

We consider a non-wrapping and monotonically increasing counter at the verifier side. This counter is maintained by the prover as a secure variable which can only be changed via TCM. Assuming a 32-bit counter, if the attestation routine is performed once a minute, the counter value needs thousands years to wrap around.

C. Security of SIMPLE

The properties of secure RA, as mentioned in [33], are:

- **Secret Key Protection**
  - **Key Access Control**: The key can only be accessed by the RA routine.
  - **Key confidentiality**: The key is stored in a secure memory area, and not leaked during or after the execution of RA.

  Guarantees by SIMPLE. The attestation code and the secret key(s) are stored inside the secure memory area (SμV memory). Considering the same approach followed to verify SμV [39], we have verified SIMPLE to be memory-safe and crash-free (see Appendix A). Furthermore, we have employed HACL* [42], a verified cryptographic library, thus ensuring also the functional-correctness property. The memory-safety property of entire system guarantees no access to the secret key(s) by

[1]Despite excluding side-channel attacks, the cryptographic library that we use is formally-verified to be secure against timing software-based side channel attacks as we explain later on.

[2]A combination of sequence numbers with nonces is used: (i) to adhere to the best security practices and prevent replay attacks on both sides, and (ii) to provide more reliability in disruptive or malicious networks as explained later in SIMPLE+ (Section V). Neither the verifier nor the prover has to maintain a history of nonces. Holding the last generated nonce at the verifier side is enough as sequence numbers used prevent replaying (old) messages with repeated nonces. On the other hand, SμV can securely control a HW-based clock and make it read-only for other software modules. Thus, authentic timestamps can be used (even as counters) to mitigate DOS attacks, rather than using nonces and sequence numbers. We leave this as an option for devices with internal clocks.

any instruction except the right one inside SIMPLE. Also, the verified functionality makes sure that the state of all temporary memory areas used during the execution of RA is unchanged by erasing all temporary-stored data. Accordingly, along with the atomicity property, where interrupts are disabled during RA, memory-safety and functional-correctness properties maintain the key confidentiality.

- **Safe Execution**
  - **Atomicity**: RA execution cannot be interrupted. This prevents leakage of secret key(s), ROP attacks, and roving malware from evading detection.
  - **Immutability**: The attestation code should not be modified by any untrusted code or malware in order to guarantee the validity of responses and the non-leakage of secret keys.
  - **Controlled Invocation**: The attestation should always start execution from the first instruction, which is disabling interrupts, and end up with the correct computed digest of memory after erasing all temporary stored data and enabling interrupts.
  - **Functional Correctness**: RA implementation must always demonstrate the correct and expected behavior of ρ whenever υ asks for the attestation report. Furthermore, the attestation process must always finish in a finite time.

Guarantees by SIMPLE. Atomic execution is guaranteed by disabling all global interrupts before executing any function, including SIMPLE, residing in SμV memory, and enabling them upon return. **Immutability** is enforced after initial deployment of SμV using a physical programming device (e.g. JTAG), where all incoming applications cannot be installed without passing through the loader/verifier function inside SμV. This ensures that no application instruction can read, write, or execute any part of SμV memory (more than immutability), except for hard-coded specific entry points; see Table I. Thus, **Controlled Invocation** is also guaranteed (considering the verified memory-safety property; see Appendix A). The combination of verified atomicity and absence of crashes (see Appendix A) properties preserves the bounded execution time property. Given the bounded execution time and memory-safety properties, functional correctness is ensured in SμV since the entire code is predictable and deterministic. In other words, all instructions of the TCM (SμV) are statically allocated conforming to the specifications of the memory-safety property. Considering that SIMPLE is a statically-allocated code built atop SμV, its functional correctness depends on the correctness of the HMAC primitive used. Thus, in line with [33], we employ HACL* HMAC-SHA256 function [42]. This function has been formally verified to be memory-safe, functionally-correct, and cryptographically-secure against timing side channel attacks. Hence, the **functional correctness** of SIMPLE is verified.

That is, as long as the IoT device is securely initialized with SμV (e.g. using JTAG) by a trusted party, where SμV is verified and guaranteed to execute correctly after any reboot (see Appendix A), then SIMPLE is a secure RA scheme.

D. SIMPLE vs. other RA approaches

Although existing hybrid RA techniques [6], [8] rule out all kinds of physical attacks, they are more immune to some physical attacks than SIMPLE. An adversary with physical access to the IoT device can simply re-flash (remove) the MicroVisor that is the core of providing the dynamic root of trust, whereas this is not the case in hybrid RA techniques, which do not depend on any hypervisor or kernel. Furthermore, retrieving secret keys from hardware-protected modules in hybrid RA is more time-consuming than software protection in SIMPLE. Nevertheless, SIMPLE has the advantage of not requiring any hardware modification or extension to the millions of already deployed Class-1 IoT devices.

While traditional (*most likely insecure*) SW-based RA approaches do not depend on any verified kernel or some existing hardware properties in MCUs (e.g. Flash), SIMPLE fills a gap between SW-based and hybrid RA families. To the best of our knowledge, SIMPLE is the first pure software RA that provides guaranteed security properties for any device capable of running such verified software architecture w.r.t remote-only attacks. Figure 3 shows the position of SIMPLE w.r.t the spectrum of RA techniques.

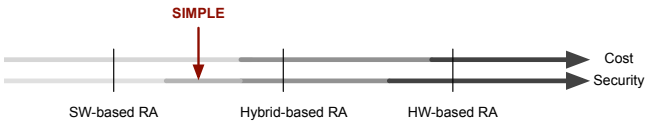


Figure 3: The position of SIMPLE w.r.t the spectrum of RA techniques.

Please note that due to the limited space and to keep consistency of the paper, we briefly describe the approach followed in verifying the various properties of SIMPLE in Appendix A. Further details can be found in [39].

V. SIMPLE+

In the majority of IoT application domains, IoT devices are deployed in massive numbers, forming self organizing mesh networks or swarms. For this purpose, we extend SIMPLE by proposing SIMPLE+, a collective attestation scheme that efficiently and securely verifies the software integrity of a group of devices.

System Model. We consider a swarm of devices, where the swarm topology can be either static or dynamic. We assume that minimal connectivity is maintained in the swarm even during motion, and there is at least one device within the range of the verifier. Devices can have different software configurations and possess heterogeneous hardware capabilities, taking into account that each device should satisfy the minimal requirements needed for secure RA, that are specified either in

this paper (see Section IV-C), or in [21]. In particular, Class-1 IoT devices, that lack the hardware-based MPU, should employ a trusted software-based architecture, such as the formally-verified SμV [39].

Overview. SIMPLE+ is a lightweight swarm attestation scheme that consists of three phases. First, the initialization phase (Section V-A) takes place once before the deployment, where all devices in the network are initialized by a trusted network operator, i.e. υ, with a bunch of private and public data. Second, the attestation phase (Section V-B), in which, whenever needed, the verifier asks the devices to check their software integrity and compute their attestation reports. This phase can be repeated more than one time before collecting the attestation reports. Last, the collection phase (Section V-C), in which the verifier collects the attestation reports efficiently, without relying on any, possibly expensive in terms of memory footprint and computation overhead, aggregation scheme. SIMPLE+ is informative in the sense that after collecting the aggregated attestation reports, the verifier is able to distinguish between healthy and compromised devices.

A. Initialization phase

υ initializes the secure memory area of all devices with several secrets. First, devices store two group-wide secrets: K_{auth} and K_{col}. K_{auth} is used to authenticate υ and other swarm-connected devices. K_{col} is a session key that is used in the collection phase and is updated securely in all devices after every attestation request, by computing its hash value, i.e. $K_{col} = HASH(K_{col})$. Second, each device maintains a counter, C_p, and is equipped with one device-specific key, K_{attest}, for the attestation purpose [3]. All devices are deployed in a healthy status, and thus they securely store an initial attest value equal to one (i.e. $attest = 1$). Finally, each device stores its own identifier (ρ_i).

B. Attestation phase

Whenever needed, υ broadcasts an attestation request, $Attest_{req}$, consisting of a monotonically increased value of C_v, a $Nonce$, a set of valid software states (VSS), and the HMAC of all attached values, computed using K_{auth}, to all nearby devices (see ① in Figure 4). $Attest_{req}$ is then further propagated by all receivers in the *possibly dynamic* network via broadcasting too, after being successfully authenticated (see ② in Figure 4). Afterward, each receiver computes its current software state value (VSS') using its own K_{attest} [4]. If the computed digest (VSS') equals to one of the valid software states in VSS, the attest value will equal to the result of a bit-wise AND operation between its old value ($initially, attest = 1$) and "1". Otherwise, the attest value will equal to "0". The attest value is stored in the secure memory area. At the end of this phase, υ updates the common session secret key, K_{col}, as shown in Figure 4. Likewise, each ρ increases its C_p to be equal to the latest updated and authenticated sequence number (C_v) of υ. Furthermore, K_{col} is updated securely for later collection purposes. If the attestation phase is performed multiple times before collecting the attestation reports, and some of the provers have received the latest attestation request while missing some previous ones (due to potential lossy

[3]For the sake of simplicity, we further proceed describing SIMPLE+ by assuming that K_{attest} is unified on all provers (considering only remote-only attacks in this paper).

[4]VSS could contain a description of what should be measured and included in the HMAC computation, i.e. part of or full Flash memory, RAM, etc.

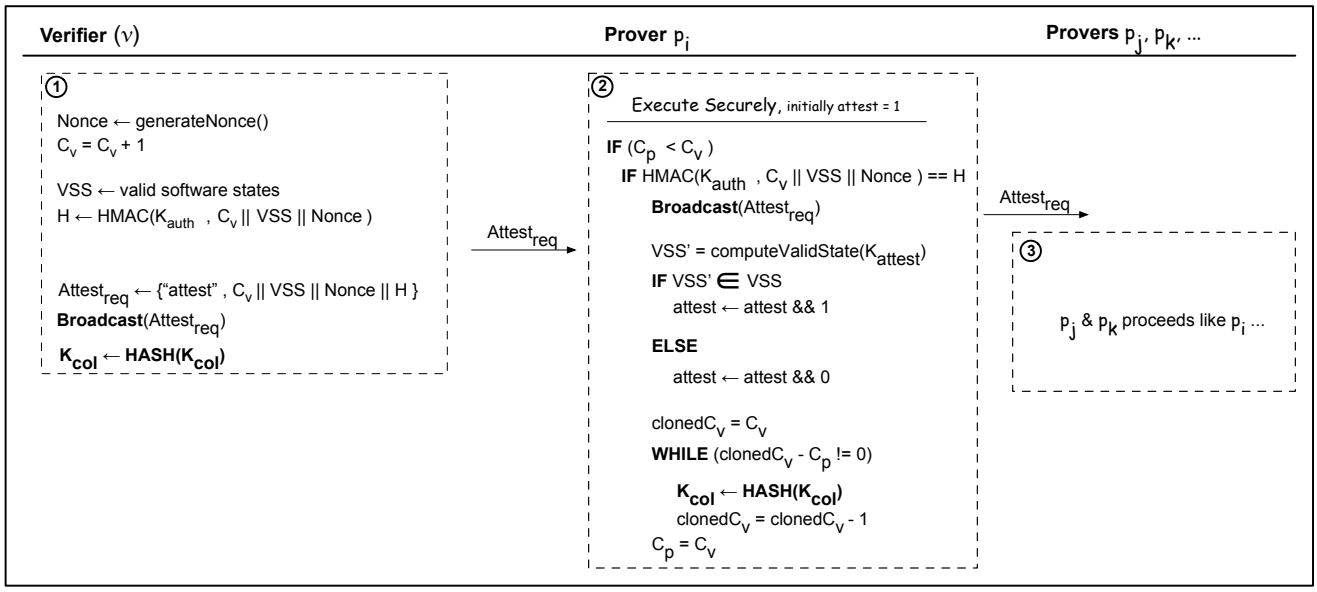


Figure 4: An overview of the attestation phase of SIMPLE+

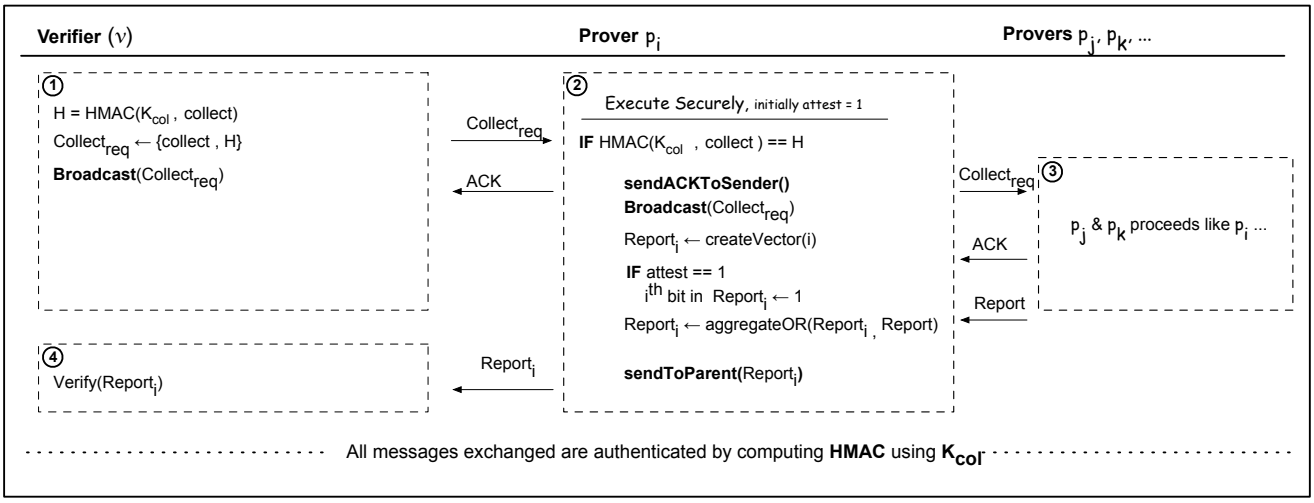


Figure 5: An overview of the collection phase of SIMPLE+

transmission of wireless communications), they update K_{col} a number of times equal to the difference between C_v and C_p, before updating the value of C_p (see Figure 4).

C. Collection phase

After running the attestation phase one or more times, v can collect the attestation responses by emitting a collection request, $Collect_{req}$, via broadcasting to all devices in the vicinity. $Collect_{req}$ is authenticated only using the latest updated collection key, K_{col} (see ① in Figure 5). All receivers that manage to successfully authenticate $Collect_{req}$, re-broadcast it to other devices and send an acknowledgment to the sender for notification purposes. The number of received acknowledgments determines the number of subsequent *aggregated* attestation reports that should be received from the same devices. Thus, a *virtual* spanning tree is formed for this short time period (in practice, it should be in terms of milliseconds)[5]. Each receiver ρ_i, creates an n-bit vector, with zero values as defaults, and sets the i^{th} position to be equal to the stored *attest* value, where n is greater than or equal to i, and n % 8 = 0. Marking the i^{th} position with 1 means that the corresponding prover, ρ_i, is healthy, whereas 0 means that it is compromised. Aggregating attestation reports occurs through a simple OR operation that would require a negligible overhead on even very simple devices. For example, the attestation report of a healthy prover of ID 10, ρ_{10}, will equal to 0000001000000000, whereas the attestation report of a healthy prover, ρ_7, will equal to 01000000. Aggregating both reports at any of these provers will result in an aggregate of 0000001001000000 (see ② in Figure 5). The verification process at v side is simple. After authenticating and ORing all received aggregates, the positions of zero values indicate compromised provers in the swarm.

[5]The verifier can include an amount of time in the collection request to be used by the provers to maintain timers, thus determining the upper bound of waiting time for children to send their attestation reports.

Both attestation and collection phases start securely by disabling global interrupts at the beginning and enabling them at the end. This preserves atomicity and prevents leakage of secrets and ROP attacks (as explained in Section IV-C).

D. Features of SIMPLE+

One of the key advantages of using sequence numbers (counters) in SIMPLE+ is reducing the number of false-positive cases, i.e. healthy devices regarded as compromised ones. The decoupling of attestation and collection phases helps temporarily disconnected devices (due to abnormal conditions in the communication protocol) in teaming up and synchronizing with the swarm again when receiving a new attestation request by observing the difference between the latest value of the verifier's counter and the current local counter. Accordingly, they update the common session collection key and their local counters. Also, this decoupling is advantageous in various ways. First, it maintains a high level of efficiency in attesting heterogenous static/dynamic networks, where the differences of hardware capabilities do not prevent powerful and probably time-sensitive devices from doing their normal tasks after performing self-attestation, rather than waiting other devices to compute and send their attestation reports. Second, this maintains a high level of immunity against a roving[6] malware, as detecting it at least once would never change the attest value (due to the use of the AND operation, i.e. 0 AND anything = 0), even if it succeeds to evade detection in the subsequent attestation periods.

E. Security of SIMPLE+

The security of SIMPLE+ at the edge side is inherited from the verified secure RA properties of SIMPLE (see Section

[6]A roving malware is a kind of malware that is always aware of the attestation schedule of the IoT device and thus it is only active between any two successive attestation routines which deletes itself at the beginning of the attestation to evade detection.

IV-C). Therefore, we only analyze the security of SIMPLE+ at the network level.

The goal of any swarm attestation scheme is for a verifier v to distinguish between healthy and compromised devices in a swarm S, where limited false positive [7] cases are accepted but not vice versa. This is formalized by the following adversarial experiment $ATT^{n,c}_{adv}(j)$, where adv is an adversary interacting with n devices, and compromises up to c devices in S, where $c \leq n$. Considering that adv is computationally bounded to the capabilities of devices deployed in S, adv interacts with the devices a polynomial number of times j, where j is a security parameter. Upon verifying the aggregated attestation reports, v outputs a decision as 0 or 1. Denoting the decision made as A, A = 1 means that the attestation routine is finished successfully and all compromised devices are detected, or A = 0 otherwise. According to the definition of secure swarm attestation given by [12], we summarize the security of SIMPLE+ at the network level with an informal proof sketch.

Theorem 1 (Security of SIMPLE+)**.** *SIMPLE+ is a secure scalable attestation protocol if $Pr[A = 1 \mid ATT^{n,c}_{adv}(j) == A]$ is negligible for $0 < c \leq n$, if the PRNG and HMAC schemes used are secure and selective forgery resistant.*

Proof. Considering remote-only attacks (see Section IV-A), authenticating messages during the entire lifetime cycle of RA (using HMACs), and using formally-verified cryptographic primitives that are secure and selective forgery resistants, SIMPLE+ is also secure at the network level. □

VI. Implementation and Evaluation

A. Implementation

A prototype of SIMPLE has been implemented using two different AVR-based IoT platforms: Arduino Uno [43] and MicroPnP [24]. SμV is employed on both platforms since they lack the hardware-based MPU. The Arduino Uno offers an 8-bit AVR ATmega 328p microcontroller running at 16MHz with 2kB of SRAM and 32kB of Flash memory, whereas the MicroPnP platform provides an 8-bit AVR ATmega 1284p microcontroller running at 10MHz with 16kB of SRAM and 128kB of Flash. The IEEE 802.15.4e Time-Slotted Channel Hopping [44] radio transceiver is used for wireless communication. We used HACL* HMAC-SHA256 [42] as a keyed-hash message authentication code. We have extended the implementation in SIMPLE+ to accommodate a testbed of five devices, where two of these devices belong to Arduino Uno, whereas the remaining ones are MicroPnP platforms. Similar to all existing approaches, we have used the experimentally measured values to simulate scalability using OMNET++ [45].

B. Evaluation

1) Execution time: Computing the HMAC of the entire Flash memory consumed about 17.41 and 2.64 seconds in MicroPnP and Arduino Uno platforms respectively. Table II shows the detailed execution time of the various cryptographic operations. An average propagation delay of 17ms has been measured between the two neighboring nodes (v and ρ) with a throughput of no more than 60kbps.

Table II: Cryptographic Runtime Measurements in SIMPLE

Runtime Measurements	Arduino Uno ATmega328P	MicroPnP ATmega1284P
Computing HMAC of 10 KB of memory	0.89 s	1.48 s
Nonce authentication	44.74 ms	71.68 ms
Creating attestation report along with its HMAC	44.75 ms	71.87 ms
ORing two vectors of length 1000 Bytes (8000 devices)	1.7 ms	2.2 ms

2) Memory footprint: SIMPLE requires no more than 80 bytes of permanent storage to store the secrets and the prover identifier, whereas this space grows up to 120 bytes in SIMPLE+. The core functions of SμV consume 1070 bytes of Flash memory. The verified implementation of HACL* HMAC-SHA256 occupies about 6kB of Flash memory. No more than 1.5kB of temporary storage (RAM) is required in each device connected to a swarm of up to 10k devices to create and aggregate attestation reports.

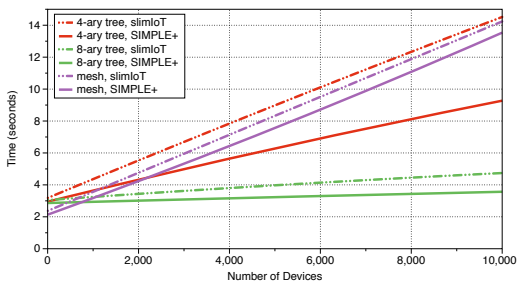
3) Power consumption: We have only measured the impact factor on the power for the MicroPnP IoT platform as it consumes more power than Arduino Uno. MicroPnP IoT platform consumes 3.54mA when operating on 10MHz in the active mode, and 54.5μA in the idle mode. Every MicroPnP platform is powered by a standard 3000mAh battery pack. The baseline battery lifetime, if the MCU is in the sleeping mode constantly, is 6.5 years. Considering these values, Figure 6c shows the estimated lifetime of the battery when running SIMPLE+ at various time rates. For example, the battery lifetime would last for no less than 5 years when attesting devices three times and then collecting reports every 3 hours.

4) Scalability and Efficiency: OMNET++ [45] is used to simulate large networks with different topologies and configurations. The computational and network delays are adjusted according the experimentally measured values (see Table II).

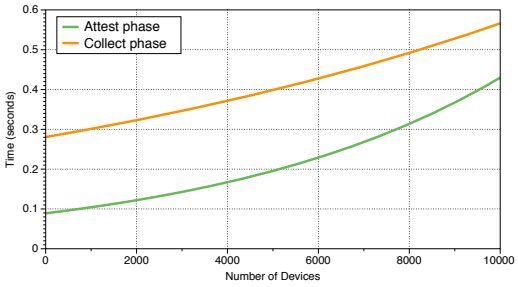
SIMPLE+ vs. slimIoT. Figure 6a compares the total runtime of SIMPLE+ with slimIoT [15], a state-of-the-art swarm attestation scheme for resource-constrained IoT devices, in various static and homogeneous network topologies consisting of Arduino Uno-based nodes. To ensure objectivity, we have implemented slimIoT using the same cryptographic functions and underlying hardware as SIMPLE+. In SIMPLE+, both attestation and collections phases are performed once and directly after each other with no waiting time in between. SIMPLE+ performs better than slimIoT in all topologies since the number of packets exchanged in any topology is always less than what exchanged in slimIoT, where slimIoT requires the verifier to emit at least 4 messages before collecting the attestation reports. Furthermore, slimIoT requires loosely-time synchronized devices in the network, increasing the communication overhead.

heterogeneous and dynamic networks. Figure 6b shows the communication overhead of each of the attestation and

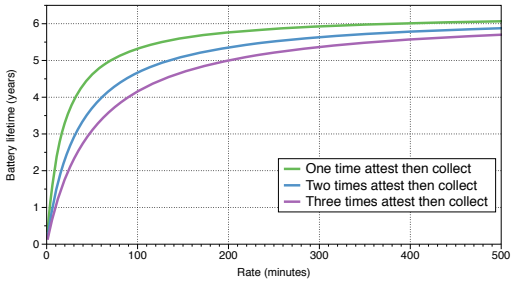
[7] A false positive means that a healthy device is regarded as compromised, i.e. due to absence because of disruption in the network.



(a) Runtime of SIMPLE+ Vs. slimIoT



(b) Communication overhead in each phase.



(c) The battery lifetime (MicroPnP).

Figure 6: An overview of the various evaluation factors of SIMPLE(+)

collection phases in SIMPLE+. The measurements are reported in a heterogeneous mesh network consisting of different number of devices (50% Arduino Uno and 50% MicroPnP), where 25% of them are moving randomly with a speed of 10m/s and a communication range of 50 meters. Emitting and authenticating an attestation request in a network of 10k devices requires no more than 0.45 second, whereas propagating, authenticating, and aggregating attestation reports in the collection phase demands less than 0.6 second for the entire network.

VII. Conclusion

This paper describes SIMPLE(+), the first reliable and scalable remote attestation scheme that provides guaranteed security properties as hybrid-based techniques without requiring hardware support or modification, and w.r.t remote-only attacks. SIMPLE(+) proposes secure RA on resource-constrained MPU-free IoT devices by leveraging the Security Micro-Visor (SμV), a software-based memory isolation technique. SIMPLE(+) is reliable in the sense that the entire software architecture is formally verified to be memory-safe, crash-free, and functionally-correct. Evaluation results demonstrate that SIMPLE(+) is lightweight and very efficient to be used in static and dynamic networks consisting of multiple heterogeneous IoT devices.

Future Directions. As a future work, we aim to achieve a fully-verified design and implementation of SIMPLE(+) using different cryptographic schemes. Furthermore, we would like to go beyond the AVR architecture and port this work to other MCU architectures such as Von Neumann.

References

[1] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, *et al.*, "Understanding the mirai botnet," in *USENIX Security Symposium*, pp. 1092–1110, 2017.

[3] A. Seshadri, M. Luk, and A. Perrig, "SAKE: Software Attestation for Key Establishment in Sensor Networks," in *Distributed Computing in Sensor Systems*, (Berlin, Heidelberg), 2008.

[4] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: software-based attestation for embedded devices," in *Proceedings of the IEEE Symposium on Security and Privacy*, IEEE, 2004.

[5] 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*, (New York, New York, USA), p. 6, ACM, june 2015.

[6] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust.," in *19th NDSS Symposium*, The Internet Society, 2012.

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

[8] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the 9th European Conference on Computer Systems*, (New York, NY, USA), p. 14, ACM, 2014.

[9] 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.

[10] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *Dependable Systems & Networks, 2009. DSN'09.*, IEEE, 2009.

[11] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Security and privacy (SP), 2010 IEEE symposium on*, pp. 414–429, IEEE, 2010.

[12] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "Seda: Scalable embedded device attestation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 964–975, ACM, 2015.

[13] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, "Sana: secure and scalable aggregate network attestation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 731–742, ACM, 2016.

[14] M. Ammar, M. Washha, and B. Crispo, "Wise: Lightweight intelligent swarm attestation scheme for iot (the verifier's perspective)," in *Proceedings of the 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, IEEE, 2018.

[15] M. Ammar, M. Washha, G. Sankar Ramachandran, and B. Crispo, "slimiot: Scalable lightweight attestation protocol for the internet of things," in *Proceedings of the 2018 IEEE Conference on Dependable and Secure Computing (DSC)*, IEEE, 2018.

[16] 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," *ACM SIGOPS Operating Systems Review*, vol. 39, oct 2005.

[17] Y. Li, J. M. McCune, and A. Perrig, "SBAP: Software-Based Attestation for Peripherals," in *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, (Berlin, Heidelberg), 2010.

[18] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM workshop on Wireless security*, pp. 85–94, ACM, 2006.

[19] Trusted Computing Group, "TPM Main Specification Level 2 Version 1.2." http://www.trustedcomputinggroup.org/tpm-main-specification/, 2011. [Online; accessed 13-February-2017].

[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, ACM, 2009.

[21] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 244, 2014.

[22] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," tech. rep., 2014.

[23] N. A. Ali, M. Drieberg, and P. Sebastian, "Deployment of micaz mote for wireless sensor network applications," in *Computer Applications and Industrial Electronics (ICCAIE), 2011 IEEE International Conference on*, pp. 303–308, IEEE, 2011.

[24] N. Matthys, F. Yang, W. Daniels, W. Joosen, and D. Hughes, "Demonstration of micropnp: the zero-configuration wireless sensing and actuation platform," in *Sensing, Communication, and Networking (SECON)*, IEEE, 2016.

[25] W. Daniels, D. Hughes, M. Ammar, B. Crispo, N. Matthys, and W. Joosen, "S μ v-the security microvisor: a virtualisation-based security middleware for the internet of things," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, 2017.

[26] R. V. Steiner and E. Lupu, "Attestation in wireless sensor networks: A survey," *ACM Computing Surveys (CSUR)*, vol. 49, no. 3, p. 51, 2016.

[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, ACM, 2017.

[28] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "Darpa: Device attestation resilient to physical attacks," in *Proceedings of the 9th ACM WiSec Conference*, pp. 171–182, ACM, 2016.

[29] G. Cabodi, P. Camurati, C. Loiacono, G. Pipitone, F. Savarese, and D. Vendraminetto, "Formal verification of embedded systems for remote attestation," *WSEAS Transactions on Computers*, vol. 14, pp. 760–769, 2015.

[30] F. Lugou, L. Apvrille, and A. Francillon, "Smashup: a toolchain for unified verification of hardware/software co-designs," *Journal of Cryptographic Engineering*, vol. 7, no. 1, 2017.

[31] 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*, 2017.

[32] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009.

[33] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "{VRASED}: A verified hardware/software co-design for remote attestation," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1429–1446, 2019.

[34] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[35] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 15–26, ACM, 2008.

[36] 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, ACM, 2016.

[37] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter, "Diat: Data integrity attestation for resilient collaboration of autonomous systems.," in *NDSS*, 2019.

[38] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: Lightweight hardware-assisted attestation of program execution," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2018.

[39] M. Ammar, B. Crispo, B. Jacobs, D. Hughes, and W. Daniels, "Sμv - the security microvisor: A formally-verified software-based security architecture for the internet of things," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 5, pp. 885–901, 2019.

[40] I. Jtc, "Sc22/wg14. iso/iec 9899: 2011," *Information technology—Programming languages—C. http://www. iso. org/iso/iso_catalogue/catalogue_ tc/catalogue_detail. htm*, 2011.

[41] M. Ammar, W. Daniels, B. Crispo, and D. Hughes, "Speed: Secure provable erasure for class-1 iot devices," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018.

[42] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1789–1806, ACM, 2017.

[43] Y. A. Badamasi, "The working principle of an arduino," in *Electronics, computer and computation (icecco), 2014 11th international conference on*, IEEE, 2014.

[44] T. Watteyne, M. Palattella, and L. Grieco, "Using ieee 802.15. 4e time-slotted channel hopping (tsch) in the internet of things (iot): Problem statement," tech. rep., 2015.

[45] A. Varga, "The omnet++ discrete event simulation system (http://www. omnetpp. org). european simulation multiconference (esm2001), prague," 2001.

[46] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *ACM SIGOPS Operating Systems Review*, vol. 27, pp. 203–216, dec 1993.

[47] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: fully verified software fault isolation," in *Proceedings of the ninth ACM international conference on Embedded software*, pp. 289–298, ACM, 2011.

[48] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for c and java," in *NASA Formal Methods Symposium*, Springer, 2011.

[49] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008.

[50] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Logic in Computer Science*, IEEE, 2002.

APPENDIX A
FORMAL VERIFICATION

A. Methodology and Verified Properties

In principle, errors in either the design or implementation of any software module may violate the safety and security properties. Therefore, trust must be a key requirement for any security-related service based on software (or even hardware). Ensuring reliability and high-assurance of SIMPLE involves formally verifying both the design and implementation to prove certain safety and correctness properties. A verified design provides a proof of semantic correctness, while a verified implementation ensures safety and functional correctness. The design of SμV is equivalent to the software fault isolation approach (SFI) [46], whose correctness has been formally verified in [47], whereas the design of SIMPLE is inline with the recently proposed formally-verified correct design of secure RA in [33]. Therefore, in this paper, we assume that our design is correct and focus only on verifying certain safety and correctness properties of implementation of the entire software architecture. Furthermore, we assume that the underlying MCU architecture is correctly implemented and adheres to its design specifications. In comparison to the work in [33] that follows the Model Checking approach to verify certain properties, we follow the Theorem Proving approach to verify safety properties, leveraging VeriFast [48], a separation logic-based program verifier that relies on the Z3 theorem prover [49].

We briefly introduce VeriFast and then elaborate on the properties that have been verified in both SμV and SIMPLE and their implications on trustworthiness and reliability.

VeriFast [48] is a formal verification tool that checks and verifies certain safety properties of C programs. It is built on top separation logic that extends Hoare logic [50]. The verification process starts with human-assisted annotations of C code. The annotations include pre- and post-conditions expressed as separation logic assertions, ghost data structures, and ghost lemmas. The body of each function of the C program is executed symbolically, starting from a symbolic state identified by the function's preconditions, going through checking permissions for each statement inside the function, updating the symbolic state, and ending up with a symbolic state that should meet the function's postconditions. Certain safety properties are guaranteed if both pre- and post-conditions and any additional annotations are satisfied during the symbolic execution of that function.

Verification. The verification process begins with writing formal specifications in terms of annotations to indicate what should each function do. These annotations consist of contracts (pre- and post-conditions), predicates to describe data structure, and lemmas (i.e. ghost functions). After annotating all function headers and their bodies, the verifier in VeriFast will then check whether the code complies with these annotations, providing proof steps that are automatically generated. Thus, any illegal access or operation will be detected by VeriFast, reporting an error. Listing 1 shows a snapshot of verified statements inside a function body, where the function belongs to the architecture of SIMPLE. All annotations in VeriFast are written inside special comments (/*@ ... @*/ or //@ ...) which are ignored by the C compiler. As shown, the function contract consists of pre-conditions, specified by the keyword **requires**, and post-conditions, specified by the keyword **ensures**. The pre-conditions describe the permissions that the function requires to execute successfully, whereas the post-conditions describe the permissions that are transferred from the function to its caller when the function is done with the execution.

To abstract over the set of permissions required by the function, permissions can be grouped and hidden via *predicates*. In Listing 1, we use a **predicate**, i.e. named $state\_pred$, in the function contract to hide the complexity of formalizing permissions needed. For each loop, VeriFast requires identifying a **Loop Invariant** to verify an arbitrary sequence of loop iterations by verifying the loop body once, starting from the initial symbolic state.

A number of user-defined along with built-in lemmas (i.e. $uchars\_split$, etc.) have been used in the verification process. A lemma is a way of writing theorems and proofs in the form of ghost C-like functions with the exception that they do not perform field assignment or call regular C functions. For example, in Listing 1, $XOR\_NoUnderflow$ lemma is used to verify that there is no arithmetic underflow once the bit-wise XOR operation takes place in the ch function. The body of the lemma represents the proof of the theorem, as shown in Listing 1.

Verified Properties. By formally-verifying the software architecture of SIMPLE using VeriFast, we prove that it is **memory-safe**, which does not exhibit any undefined behavior as described by the C11 standard [40]. In particular, the entire system is free from the following run-time errors: division by zero, integer overflow/underflow, buffer overflow/underflow, invalid pointer dereferences, out-of-bounding array indexing, illegal memory access, double free, use after free, problematic bit shifts, type conversions that would overflow the destination, and memory leaks. The freedom from the aforementioned run-time errors guarantees the **absence of crashes** property since there is no non-terminating recursions or loops, no segmentation faults (i.e. attempting to write read-only memory), or exceptions (i.e. division by zero). Considering atomicity (which is a feature ensured by disabling interrupts), the absence of crashes property preserves **bounded-execution time** property, where all functions are guaranteed to finish execution in a finite period. In line with the work in [33], [42], **functional correctness** is verified, given the aforementioned verified properties along with the deterministic, predictable, and statically allocated code in the memory (no dynamic memory allocations used through the entire software architecture) along with the usage of a functionally-correct HMAC primitive.

Implications on SIMPLE. SμV aims at providing strong software-based isolation guarantees. This can be achieved by verifying *memory safety* and *control flow integrity* properties. The former has been verified as explained above. All SμV functions start execution by disabling all interrupts and end

up enabling them. The combination of atomicity and memory-safety properties guarantees the control-flow integrity property, as execution may not scape a predetermined control flow graph. Consequently, **reliable software-based memory isolation** is guaranteed, where the data-access and control-flow policies shown in Table I are fulfilled in the presence of an untrusted *modified* compiler. Thus, considering its formally-verified properties and being built atop the formally-verified MPU-Like SμV, SIMPLE is proven to be secure and sound RA.

B. Formal verification overhead

The entire software architecture of SIMPLE consists of about 580 lines of code. 441 lines of VeriFast annotations (or about 0.76 line of annotation for every line of code) have been added to verify the aforementioned properties. The classification of these annotations is as follows: function contracts needed 66 lines of annotations (there were 33 pairs of requires/ensures, one pair for each function), only 1 predicate is defined throughout the entire code base, 19 loop invariants, and 42 specific lemma's are identified and employed in the verification process. Furthermore, various built-in predicates and lemmas are used in verifying the code. In terms of development overhead, extending the verified properties of SμV along with verifying SIMPLE consumed no less than 120 man-hours.

```
...
/*@
        predicate state_pred(sha_ctx_t *state,
            uint32_t *p, uint32_t h0, uint32_t h1,
            uint32_t h2, uint32_t h3, uint32_t h4,
             uint64_t theLength) = p == state->h
            &*& *p |-> h0 &*& *(p+1) |-> h1 &*&
            *(p+2) |-> h2 &*& *(p+3) |-> h3 &*&
            *(p+4) |-> h4 &*& state->length |->
            theLength;
@*/
...
static void
write_page(uint8_t *page_buf, uint32_t offset)
        //@ requires [?f]uchars(page_buf, 256, _)
            &*& 0 <= offset &*& offset <= UINT_MAX;
        //@ ensures [f]uchars(page_buf, 256, _);
{
        uint32_t pageptr;
        uint8_t i;
        /* Erase page */
        boot_page_erase(offset);
        boot_spm_busy_wait();

        /* Write a word (2 bytes) at a time */
        pageptr = offset;
        //@ div_rem(PAGE_SIZE, 2);
        i = (uint8_t)(PAGE_SIZE/2);
        //@ div_rem(PAGE_SIZE, 2);

        //@ uint8_t *page_buf0 = page_buf;
        //@ uint8_t offsetsDone = 0;

        while (i > 0)
                /*@ invariant [f]uchars(page_buf0,
                    256, _) &*& 0 <= i &*& 0 <=
                    offsetsDone &*& 0 <= pageptr
                    &*& pageptr <= UINT_MAX
                    &*& offsetsDone <= 256 &*&
                        page_buf == page_buf0 +
                        offsetsDone &*& i <= (256 -
                        offsetsDone) ; @*/
        {
        // @uchars_split(page_buf0,offsetsDone);
                uint16_t w = *page_buf;
                page_buf++;
                //@ assume(offsetsDone+1 <=
                    256-offsetsDone-1);
        ...
        }
        ...
}
...
/*@
...
lemma void XOR_NoUnderflow_char( uint8_t x,
    uint8_t y)
        requires 0 <= y &*& 0 <= x &*& x <= 255
            &*& y <= 255;
        ensures ((uint8_t)(x ^ y)) >= 0;
{
        Z zx = Z_of_uint8(x);
        Z zy = Z_of_uint8(y);
        bitxor_def(x, zx, y, zy);
        Z zt = Z_of_uint8(0);
        Z_of_uint8_gtOReq(Z_xor(zx,zy), zt);
}
...
@*/

uint32_t ch(uint32_t x, uint32_t y, uint32_t z)
        //@ requires 0 <= x &*& 0 <= y &*& 0 <= z;
        //@ ensures 0 <= result &*& result <=
            UINT32_MAX;
{
        //@ produce_limits(x);
        //@ produce_limits(y);
        //@ produce_limits(z);
        //@ AND_NoUnderflow(x,y);
        //@ AND_NoOverflow(x,y);
        //@ N_AND_NoUnderflow(z,x);
        //@ N_AND_NoOverflow(z,x);
        //@ XOR_NoUnderflow(x&y,z& ~x);
        //@ XOR_NoOverflow(x&y,z& ~x);
        uint32_t result = ((x&y)^(z& ~x));
        return result;
}
```

Listing 1: A snapshot of verifying software architecture of SIMPLE.