FlashAttest: Self-attestation for Low-end Internet of Things via Flash Devices

Zheng Zhang[1], Jingfeng Xue[1], Weizhi Meng[2], Xu Qiao[1], Yuanzhang Li[1] and Yu-an Tan[3]
[1]School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
[2]School of Computing and Communications, Lancaster University, United Kingdom
[3]School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China
Corresponding Author: Yu-an Tan Email: tan2008@bit.edu.cn

***Abstract*—Remote Attestation (RA) is an effective security service that allows a trusted party (verifier) to initiate the attestation routine on a potentially untrusted remote device (prover) to verify its correct state. Despite their usefulness, traditional challenge-response remote attestation protocols suffer from certain limitations, such as challenges in scaling attestation collection and the forced suspension of normal operation during attestation. Self-attestation tackles these issues by enabling the prover to measure its own state asynchronously with the verifier's attestation request. Existing self-attestation methods rely on hybrid architectures to provide the required security properties, which may not be compatible with low-end Internet of Things (IoT) devices due to hardware limitations. In addition, these protocols currently lack formal verification of design correctness. In this paper, we present FlashAttest, a formally verified self-attestation protocol for low-end IoT devices. FlashAttest leverages the flash device to fulfill the security properties required by self-attestation, eliminating the requirement for hardware modifications. In particular, FlashAttest allows the prover to initiate the attestation routine and guarantee the trustworthiness of the results based on the verified software-based security architecture. By collaborating with the flash device during attestation to generate timestamped reports, FlashAttest enables the verifier to collect and verify the legitimacy of the attestation results. More importantly, FlashAttest achieves strong security guarantees supported by a formally verified design using the Tamarin prover. We implement and evaluate FlashAttest on MSP430 architecture, showing a reasonable overhead in terms of memory footprint, communication overhead, runtime and power consumption. Compared with state-of-the-art self-attestation schemes, our approach achieves similar runtime overhead, low energy consumption, and reasonable memory overhead while eliminating the need for hardware modifications. The results confirm the suitability of FlashAttest for low-end devices.**

***Index Terms*—IoT security, remote attestation, flash device, memory isolation.**

I. Introduction

THE advancements in wireless network technology has given rise to the prosperity of Internet of Things (IoT), promoting the proliferation and application of embedded IoT devices in different domains. Most IoT devices depend on continuous network connections to communicate with each other and share data collected by embedded sensors. According to the Statista [1] report, there will be 18 billion connected devices by 2024 and 75 billion devices by 2033. Among them, quite a few are Class-1 IoT devices [2] that connect to the Internet with the minimum resources (with less than 10kB of Random Access Memory (RAM) and 100kB of flash memory). These devices tend to be vulnerable to remote attacks because the vast majority of such devices lack necessary security features. Indeed, these devices are generally limited in size and computing power due to cost and energy considerations, which make them inevitably become a favorite target for cyberattacks. Malicious attackers would cause massive threat to both privacy and security of the compromised device. The attack forms involve hijacking the device's control flow, injecting the malicious code in the memory, and even manipulating it to launch Denial of Service (DoS) attacks on other devices.

Remote attestation is a promising security service to defend against malicious infestation on embedded devices. It allows a trusted entity (*verifier*) to attest the trustworthiness of one (or multiple) untrusted *prover*. Traditional remote attestation techniques operate under an on-demand challenge protocol, where the attestation routine on the prover is initiated upon receiving an attestation request issued by the verifier. The normal operation of the prover has to be suspended during attestation, which can be exploited by DoS attackers to maliciously invoke attestation routines on the prover, interfering with the performance of its primary tasks. The non-interruptibility also makes remote attestation protocols between the verifier and a single prover face scalability challenges when scaling to collective protocols.

Self-attestation protocol [3]–[5] is a promising solution to overcome these drawbacks by decoupling the verifier's challenge requests from the prover's attestation computations. Self-attestation protocols generally consist of a measurement phase and a collection phase, which are executed asynchronously. Since the constraints of synchronous interaction between the verifier and the prover are released, the self-attestation protocol need to verify both the device trustworthiness (i.e., the device is not compromised) and the legitimate operations (i.e., the protocol should provide timestamped attestation result) [5]. To obtain the necessary security features, exiting approaches all build atop of hybrid architectures [6]–[8] that attempt to achieve fairly strong security guarantees based on minimal hardware requirements. However, the availability of hybrid architectures in low-end IoT devices is challenging due to several reasons. First, hybrid schemes are not compatible with legacy microcontroller due to the requirement of hardware modifications. Consequently, already deployed devices with off-the-

shelf microprocessors need to be replaced, which significantly adds to manufacturing costs. Second, hardware modifications to the underlying architecture of devices require changes in the vendor production lines, which is not always easy and incurs additional costs. Third, technical differences between different vendors may cause compatibility and flexibility challenges for hybrid schemes that rely on specific hardware extensions.

Flash devices have potential benefits in providing security features required for enabling self-attestation on low-end devices that lack security extensions by considering the unique properties. First, the flash device's non-volatile storage which is physically isolated from the host device, can be used for secure storage of protocol keys. Second, the flash controller's timer makes it possible to provide the timestamp required for self-attestation. Third, the firmware space of the flash device can be used to maintain and run custom code related to the protocol. In addition, the plug-and-play feature of flash devices enables seamless integration with the host device. Ideally, we wish to utilize the flash device to achieve the functional correctness of self-attestation, while enforcing the security of the protocol.

To this end, we present FlashAttest, a novel self-attestation protocol in conjunction with flash devices. FlashAttest leverages an external flash device of the prover to provide the necessary security properties required for self-attestation, eliminating the requirement for hardware modifications. In particular, FlashAttest is built on top of the verified software-based security architecture to guarantee the trustworthiness of state measurement on the prover device. Besides, flash device with modified firmware provides necessary security properties for ensuring legitimate operations. Due to the portability of flash memory devices, FlashAttest achieves high flexibility and availability, making it well-suited for low-end IoT devices that lack security mechanisms. In a nutshell, this paper brings the following contributions:

1) We present FlashAttest, a novel self-attestation protocol in conjunction with flash devices. To the best of our knowledge, this is the first remote attestation protocol that utilizes the prover's external flash device to provide the necessary security requirements for self-attestation, eliminating the need for hardware modifications to the prover device.
2) We define several security goals that are required in FlashAttest and design the protocol based on them. To verify the design correctness, we model the protocol and formally analyze its security using Tamarin prover [9], a symbolic protocol analysis tool.
3) We implement a prototype of FlashAttest on top of MSP430 architecture and a commercial TransFlash (TF) card. Then we evaluate it in terms of memory footprint, communication overhead, runtime overhead and power consumption.

The remainder of this article is organized as follows: Section II presents the research related to our work. Section III introduces the preliminary situation related to flash device and self-attestation schemes. The threat model, assumptions, and required security goals of our design are described in Section IV. The details of our design are provided in Section V. Analysis of the security of our approach is provided in Section VI. Evaluation and comparisons are presented in Section VII, and the paper is concluded in Section VIII.

II. Related Work

In this section, we review previous studies that inspired our work. Specifically, we discussed various remote attestation methods and explained their respective advantages and limitations. We discussed the contributions of our work compared to state-of-the-art (see Table I). We also discuss other important work relatively relevant to flash-assisted security.

A. Remote Attestation

Existing attestation schemes can be clustered into three types: (1) Software-based attestation; (2) Hardware-based attestation; (3) Hybrid attestation.

Hardware-based attestation. Hardware-based attestation executes the attestation routines in a secure environment provided by the trusted hardware component such as Trusted Platform Module (TPM) [10]. Trusted Execution Environment (TEE) [11], [12] is another widely used trusted computing technology suitable for enabling remote attestation. TEEs can be used to securely execute checksum computing in the secure world to isolate potential threats from an untrusted system. The related approaches [13], [14] are more suitable for high-end modern processors due to cost and performance considerations. In low-end IoT devices, however, these trusted hardware components are often too costly to be deployed.

Software-Based Attestation. Software-based attestation relies on the software mechanism to attest the memory of the untrusted device, making it suitable for legacy devices and low-end embedded devices. Early software-based attestation schemes [15]–[18] mostly rely on time-based checksums. SWATT [18] is an example that detects additional timing overhead caused by malicious tampering through instruction-level optimization of the attestation procedure. However, strong assumptions (e.g., extreme optimization of the checksum function) required by these techniques limit their availability. Moreover, the security of these schemes has also been challenged [19], [20].

Recently proposed software-based attestation techniques [21]–[24] attempt to address the shortcomings of early methods through more complex designs. Software-based security architecture [25], [26] is an important category of solutions to achieve memory isolation by enforcing software-based access control. The key idea is to establish a software root of trust from the boot. A Trusted Computing Module (TCM) is deployed in the bootloader area to isolate its memory address space from untrusted applications. Other security services such as remote attestation [24], secure erasure and reset can be built on top of TCM to provide strong security guarantees.

Hybrid attestation. Hybrid schemes implement a hardware/software co-design with minimal hardware extensions such as Read-Only Memory (ROM) and Memory Protection Unit (MPU) to meet the requirements for secure remote attestation. The early hybrid schemes aimed to protect the

immutability of the attestation code [6]–[8], [27]. The attestation routine is implemented as ROM-resident code to prevent malicious modification. Most of these techniques require the attestation procedure to run without interruptions, which may affect the normal operation of the device.

State-of-the-art schemes focus on solving the security challenges faced by hybrid attestation, such as key leakage vulnerabilities [28] and Time-Of-Check-Time-Of-Use (TOCTOU) attacks. IDA [29] addresses these issues by monitoring and authenticating hash computation instead of leveraging a key during attestation. However, this approach relies on a customized hardware module to provide required security features, which limits its flexibility. RATA [30] addresses the TOCTOU problem by detecting when any memory location is written, using a hardware module capable of monitoring a set of CPU signals. RATA is built upon the formally verified hybrid attestation architecture [31], which guarantees the correctness of its security properties.

Another category of schemes focuses on self-attestation for real-time requirements or to avoid unnecessary suspension of the normal operation of the device. self-attestation is another effective method. To guarantee the legitimacy of the operation (i.e., the attestation routine is initiated as expected), most schemes rely on hybrid architectures to provide hardware features such as ROM [5], read-only clock [4] or hardware trigger circuit [3]. While SIMPLE [24] implements self-attestation based on a pure software architecture, it fails to provide timestamped attestation reports, and thus the legitimacy of operations cannot be guaranteed.

Unlike existing self-attestation schemes, FlashAttest eliminates hardware extension requirements for the prover device, making it suitable for low-end embedded IoT systems. In summary, Table I compares FlashAttest with existing self-attestation protocols and other state-of-the-art attestation methods.

B. Flash-based Security

Prior works on flash-based security leverage flash devices to provide hardware-based security features in various devices without requiring custom hardware. IceClave [32] provides a lightweight TEE for programs running inside the Solid-State Drive (SSD), aiming to mitigate the security threats faced by in-storage computing. IceClave extends the TrustZone available in a majority of ARM-based SSD controllers, enabling security isolation between the untrusted in-storage program and flash management functions. FlashGuard [33] is a ransomwaretolerant SSD that achieve firmware-level recovery from encryption ransomware. FlashGuard leverages out-of-place-writes intrinsically supported by modern SSDs and further modifies the garbage collection mechanism of the firmware to ensure data recovery without requiring explicit backups. In addition to using features of flash controller or modified firmware, another relevant category of work leverages the intrinsic properties of flash devices. For device authentication purposes, Time-Print [34] utilizes timing variation within Universal Serial Bus (USB) storage devices to generate the unique fingerprint. Flashmark [35] changes the physical properties of flash cells to imprint watermarks that can be read out through standard digital interfaces. Besides, the manufacturing variations of flash devices can also be used to produce true random numbers [36]. FlashAttest leverages the flash device to provide necessary security properties for self-attestation and only requires a slight modification to its firmware.

III. Preliminary

A. Typical Structure of a Flash Device

Flash memory is widely used in mobile devices due to its non-volatile property and light weight. There are two types of flash memory: NOR and NAND. NAND is commonly used for mass storage devices, such as TF cards and SSDs. A typical flash memory device contains a controller and one or more flash chips. The controller is responsible for efficiently managing the underlying storage medium through embedded firmware. The firmware generally contains various algorithms such as garbage collection, error correction code and wear-leveling to cope with the erase-write constraints. The Flash Translation Layer (FTL) is an important part of the firmware to conceal the unique characteristics of NAND flash memory by emulating the block device interface. The host's I/O requests are mapped by the FTL and assigned to the read or write operations of specific locations in the flash.

The flash chip is composed of a set of flash blocks (e.g., 128kB–512kB large), each of which consists of a set of flash pages (e.g., 512B–4kB large). Each page is composed of data sectors for user information and spare sectors for associated metadata, such as error correction codes. The spare sectors, reserved to boost data reliability and prolong flash storage durability, are not accessible to users. They act as backup storage areas where data can be migrated when the original sectors deteriorate in quality.

B. Self-attestation Protocols

Self-attestation allows the prover to initiate the attestation autonomously, rather than initiating it synchronously upon receiving the request from the verifier, thus will not suspend the normal operation of the prover device. Self-attestation protocols generally consist of a measurement phase and a collection phase. In the measurement phase, the prover measures its own memory status and records the results. In the collection phase, the verifier collects the measurement results to determine prover's status. Since measurement and collection are performed asynchronously, self-attestation can be easily extended to collective attestation through broadcast [24] or publish/subscribe paradigm [5].

Verifying the legitimacy of the operation is crucial in self-attestation to prevent attackers from evading detection by delaying or shielding the attestation. Legitimacy means that the prover performs the attestation as expected. The prover should provide timestamped report so that the verifier can verify the legitimacy by checking the freshness of the report. Exit schemes rely on hardware extensions of hybrid architectures to construct a read-only clock to provide timestamps.

TABLE I
FLASHATTEST VS. STATE-OF-THE-ART ATTESTATION METHODS FROM VARIOUS PERSPECTIVES (HW-MODIF: HARDWARE MODIFICATION).

Method	Mechanism	HW-Modif.	Legitimate Operations	Class-1 IoT Device	Verified Design
SeED [3]	Self-attestation	Yes ✗	✓	✓	✗
ERASMUS [4]	Self-attestation	Yes ✗	✓	✓	✗
SIMPLE [24]	Self-attestation	No ✓	✗	✓	✓
SARA [5]	Self-attestation	Yes ✗	✓	✓	✗
RATA [30]	On-demand	Yes ✗	-	✓	✓
IDA [29]	On-demand	Yes ✗	-	✓	✗
FlashAttest	Self-attestation	No ✓	✓	✓	✓

Unlike the existing self-attestation protocols, FlashAttest leverages the flash device with modified firmware to provide necessary security properties. The dedicated controller and isolated firmware space of the flash device provide the pre-requisites for implementing these security properties.

IV. System Overview and Design Goals

A. System Overview

We consider an IoT network system consisting of three entities: (1) A low-end embedded devices (*Prover*) that is being attested; (2) A trusted party (*Verifier*) that verifies the trustworthiness of the untrusted IoT device; (3) An external flash (*Flash device*) that is connected to the untrusted IoT device. In our system model, the flash device and the untrusted IoT device can exchange data via the bus and the untrusted IoT device can communicate with the remote verifier over a wireless network.

The proposed protocol is divided into two phases: measurement phase and collection phase, as sketched in Fig. 1. In the measurement phase, *Flash device* serves as a Flash Verification Module (FVM) to assist the *Prover* in performing self-attestation. Specifically, *Prover* initiates the attestation procedure based on a predetermined schedule and requests a current timestamp from *Flash device* (Step ①). *Flash device* verifies the request and returns the current timestamp (Step ②). The *Prover* computes a timestamped measurement and generate the attestation report (Step ③). In the collection phase, the *Verifier* sends requests to collect the reports to verify the status of the *Prover* (Step ④ - ⑤).

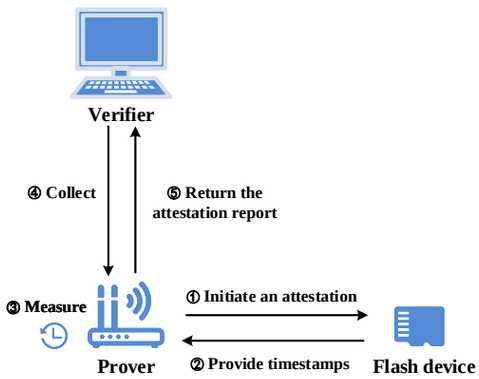


Fig. 1. System model.

B. Adversary Models and Scope

We target IoT devices that execute single-thread bare-metal applications, which are the most common in the IoT domain. This category of IoT devices has a single core, RAM, and ROM memories. Besides, we assume that the target device has a slot for external flash, which is also common in IoT devices for expanding storage capacity.

We consider the following adversaries envisaged in the context of self-attestation. Our classification is inline with the adversary model described in prior techniques [3], [5].

Communication Adversary. The communication adversary, $\mathcal{A}dv_{com}$, is a Dolev-Yao [37] adversary that controls all communication channels between the prover and the verifier. $\mathcal{A}dv_{com}$ is able to drop, modify, eavesdrop on and delay messages exchanged between the prover and the verifier, as well as inject its own messages. In our system, we consider the worst-case scenario and assume the channel between the prover and its external flash device to be an insecure channel.

Software Adversary. The software adversary, $\mathcal{A}dv_{sw}$, can manipulate any software deployed in the unprotected memory part of the device. This includes modifying any writable code, injecting malicious code, or learning any code and/or data from any memory address that is not explicitly protected. $\mathcal{A}dv_{sw}$ can also eavesdrop on or tamper with packets on any communication medium supported by the victim device.

Mobile Adversary. The mobile adversary, $\mathcal{A}dv_{mob}$ has the ability to erase its traces on the prover (e.g., removing itself from the prover) and restore the original software state. In this case, $\mathcal{A}dv_{mob}$ attempts to evade detection by the attestation protocol after the attack is completed.

Assumptions: For an adversary who may delay or refuse to send the attestation report, we assume that the verifier will detect a significantly delayed response or notice a missing interaction if it anticipates that a specific interaction should occur within a predetermined time interval.

For a mobile adversary $\mathcal{A}dv_{mob}$, we assume that the relationship between the shortest time required to infect the victim device and the interval between successive attestations satisfies the condition to achieve a negligible probability of malware evasion [3], [30].

As assumed in most prior works, the adversary's physical

access is excluded. In other words, we rule out all physical and hardware-focused attacks. For the target flash device, we assume that the modified firmware is correctly installed on the embedded device by a trusted party. We also assume that the modified firmware does not contain any memory corruption vulnerabilities. This means that the adversary cannot bypass any access policy enforced by the modified firmware.

C. Security Requirements Analysis

In this subsection, we describe the design goals required for FlashAttest to ensure security. In Section VI, we will formally prove these goals. For clarity, we denote the target device, prover, as *Prv*, and the verifier as *Vrf*. The flash device that is an external components of *Prv* is denoted as *FVM*.

Security Goals. As described in the adversary model, our security goals are stated against three kinds of attackers. Thus, the achievement of security goals is provided by both protocol design and system characteristics. In addition, although FlashAttest leverages the flash device to provide write-protected clock for self-attestation, it introduces new challenges. The difficulty comes from the fact that the channel between *Prv* and *FVM* is considered an insecure channel. Therefore, the security goal also needs to cover the security of communication data exchanged between *Prv* and *FVM*.

Goal 1. Confidentiality of secret variables.

Keys and other secret variables used in the protocol remain secret. Only trusted attestation code can acquire secret variables.

Goal 2. Immutability of the attestation code.

The attestation code should be immutable. Immutability of the attestation code prevents attackers from modifying and redirecting the access requests to the location that stores the benign copy of the memory (known as memory copy attacks).

Goal 3. Integrity and authenticity of timestamps.

In the measurement phase, the *Prv* must receive timestamps as they were generated by the authenticated *FVM*. And received timestamps cannot be modified by an attacker.

Goal 4. Freshness of timestamps.

In the measurement phase, a given timestamp can be received at most once, and only by the *Prv* generating the corresponding request. This goal mandates that the timestamp received by *Prv* can only be generated by the trusted *FVM*. It also mandates the reception of a given timestamp can only happen once.

Goal 5. Integrity and authenticity of attestation reports.

In the collection phase, the *Vrf* must receive attestation reports as they were generated by the authenticated *Prv*. And received attestation reports cannot be modified by an attacker.

Goal 6. Freshness of attestation reports.

In the collection phase, a given attestation report can be received at most once, and only by the *Vrf* generating the corresponding request. This goal mandates that the attestation report received by *Vrf* can only be generated by the trusted *Prv*. It also mandates the reception of a given attestation report can only happen once.

Goal 7. Execution Correctness of Protocol.

Apart from ensuring the design correctness of the system, the execution correctness of protocol must also be strictly followed. Specifically, the protocol should be executed in the exact order as outlined in the design.

TABLE II
NOTATION SUMMARY

Entities	**Description**
Prv	Prover
Vrf	Verifier
FVM	Flash-based verification module
$\mathcal{A}dv_{com}$	Communication adversary
$\mathcal{A}dv_{sw}$	Software adversary
$\mathcal{A}dv_{mob}$	Mobile adversary
Protocol parameters	**Description**
k_{att}	Secret key used in the measurement phase
k_{col}	Secret key used in the collection phase
mem_{prv}	Memory content of *Prv*
mem_b	Benign memory content of *Prv*
r_t	Attestation report generated at time *t*
θ	Internal state of PRNG
σ	Random output of PRNG
Time parameters	**Description**
T_{att}	Time of current attestation
T_{rec}	Timestamp obtained from *FVM*
t_{intvl}	Random time interval before the next attestation
t_{max}	Maximum time interval between two consecutive attestations
Procedures	**Description**
Enc(*k*, *m*)	Encrypts a message *m* with a key *k*
Dec(*k*, *m*)	Decrypts a message *m* with a key *k*
GetMem()	Gets the current memory content of *Prv*
GenNonce()	Generates a random challenge
GenReport(*r*, *t*)	Generates the attestation report at time *t* with attestation result *r*
Time()	Obtains current clock value from *FVM*'s clock
SetTimer(*t*)	Sets timer to expire after *t* period of time

V. PROTOCOL DESIGN

As we discussed above, FlashAttest is divided into offline measurement phase and online collection phase. In this section, we provide comprehensive details for each of the phases of the protocol.

A. Primitives and Notation

In the proposed protocol, we use several primitives to achieve aforementioned security properties. Let $\{0, 1\}^\lambda$ denote the set of all bit-strings of length λ, where $\lambda \in \mathbb{N}$. The notation of entities, parameters and procedures used in the description of FlashAttest is summarized in Table II.

Message Authentication Code. We use Message Authentication Code (MAC) to calculate the digest of the *Prv*'s memory to facilitate the verification of message correctness and authenticity. A secure MAC can be described as an algorithm $Mac(k, m) \rightarrow \{0, 1\}^\lambda$ that generates a MAC digest of input m with key k.

Pseudo-random Number Generator. We use the Pseudo-Random Number Generator (PRNG) to implement the randomness involved in the protocol. A secure PRNG can be described as an algorithm $PRNG(\theta_{i-1}) \rightarrow \{\theta_i, \sigma_i\}$ that generates internal state $\theta_i \in \{0, 1\}^\lambda$ and random output $\sigma_i \in \{0, 1\}^\lambda$ on input of previous state $\theta_{i-1} \in \{0, 1\}^\lambda$.

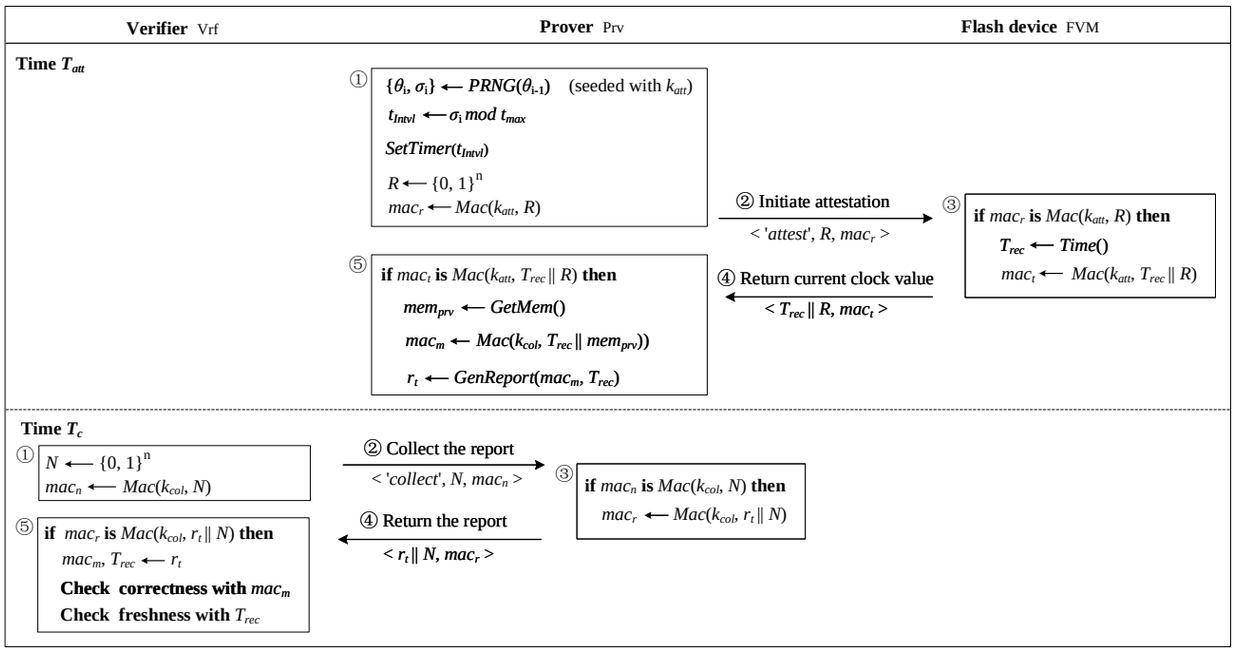


Fig. 2. FlashAttest attestation protocol. *Prv* initializes an measurement at time T_{att} via requesting *FVM* for a timestamp. *Vrf* requests to collect the measurement results from *Prv* at time T_c, which is performed asynchronously with respect to the measurement phase.

B. Measurement Phase

Fig. 2 depicts the protocol process of FlashAttest. The measurement phase utilizes *FVM* to assist *Prv* to achieve attestation based on unpredictable schedule. We assume that T_{att} is the trigger time of current attestation. At time T_{att}, *Prv* generates the random time interval t_{intvl} that triggers the next attestation (Step 1). We use PRNG seeded with attestation key k_{att} to generate a random number and truncate it with the maximum time interval t_{max} to gain t_{intvl}. Then *Prv* initiates the attestation protocol by sending a random challenge *R* and its MAC to *FVM* (Step 2). Upon receiving the request, *FVM* authenticates it and obtains the current clock value T_{rec} from its clock (Step 3). *FVM* then returns the clock value T_{rec} and its MAC to *Prv* (Step 4). Upon receiving the respond, *Prv* authenticates it and performs self-attestation combined with T_{rec}. Then, *Prv* generates an attestation report based on its memory measurement and timestamps it with T_{rec} to ensure its freshness (Step 5).

C. Collection Phase

The collection phase covers the procedure of collecting reports by the *Vrf*. We assume that T_c is the time when a collection occurs. At time T_c, *Vrf* generates a random challenge *N* and its MAC (Step 1) and sends them to *Prv* to collect the report (Step 2). Upon receiving the request, *Prv* authenticates it (Step 3) and returns the attestation report and its MAC to *Vrf* (Step 4). After authenticating the response, *Vrf* can parse the attestation report to obtain the current state of *Prv* (Step 5). To verify the correctness of the measurement, *Vrf* can recalculate the expected MAC based on k_{col} using *Prv*'s

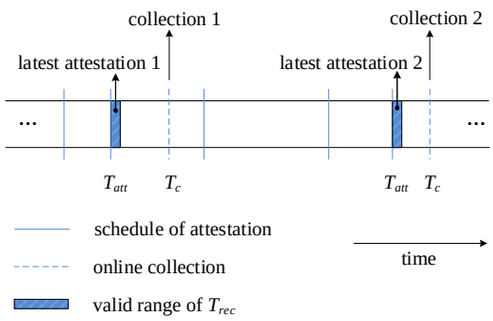


Fig. 3. Timeline of FlashAttest.

benign memory copy and compare it with the received one. The freshness of the report can be verified by checking based on T_{rec} and T_c, as shown in Fig. 3. We assume that *Vrf* knows the schedule of attestation, and its clock is synchronized with *FVM*'s and *Prv*'s. Then *Vrf* claims that the attestation result meets the freshness requirement when the following three aspects are met: (1) the response to the collection request is received within the expected time; (2) the received attestation report is the latest one; (3) the generation time of the report is in accordance with the schedule (i.e., T_{rec} falls within the legitimate time range required by freshness). Considering the communication overhead and other factors, the actual valid range of T_{rec} is evaluated by *Vrf*.

If the correctness of the measurement and the freshness of the attestation report are checked successfully, *Vrf* concludes that *Prv* is in a trustworthy state. Otherwise, *Prv* is concluded to be compromised.

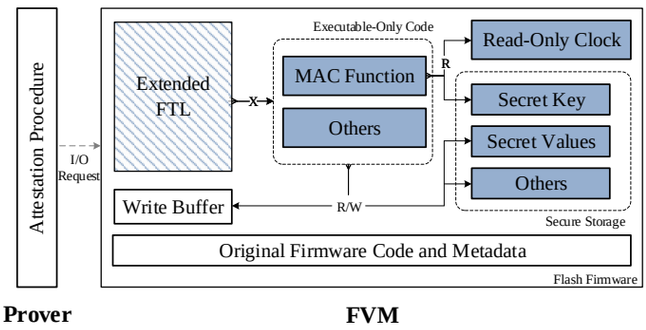


Fig. 4. The systems architecture for the flash-based verification module (FVM) in FlashAttest.

D. Flash-based Verification Module Design

According to the protocol, *FVM* provides authenticated timestamps to ensure the freshness of the attestation results. To implement these security features, we design *FVM* on a common commercial flash device. The internal components of *FVM* with the employed access policy are visualized in Fig. 4.

Compared with the original firmware, *FVM* adds three other parts. The *executable-only code* region is used to store the MAC function and other protocol-related code. The *secure storage* memory is responsible for storing secret attestation keys and other secret variables. Lastly, the *read-only clock* is responsible to provide timestamps used in the attestation. These components are design to adhere following access policy:

- The *executable-only code* region is only executable and only invoked by extended FTL.
- The *secure storage* memory can only be accessed by code reside in *executable-only code* region.
- The read-only clock and the secret attestation key are read-accessed only by the protocol-related code and are not modifiable by any software.

To implement our design, we reverse-engineer and analyze the original flash firmware, injecting custom code to expand its functionality. In the following, we will describe the design and implementation details of each component.

Custom I/O requests. According to the characteristics of flash memory devices, the controller can only passively manipulate the firmware code to respond to the host's I/O requests. To implement the proposed protocol, we split the interaction procedure between *Prv* and *FVM* into several equivalent I/O request-response pairs. These I/O requests are designed as reading or writing specific sectors of the flash chip to avoid conflicting with regular data read/write requests. Spare sectors are ideal for this purpose, as they are additional sectors that are not included in the flash's advertised capacity. They serve as a backup for when a sector becomes corrupted. Typically, a user application does not initiate I/O requests to access these sectors under normal circumstances.

In Fig. 2, the second steps of the measurement phase are equivalent to an I/O request of writing spare sector S1. The attestation request will be written into *FVM* as the target data

Algorithm 1: Attestation code on *Prv* side.

Input: AR: address range to be attested
s_1: address of spare sector S1
θ_{i-1}: random number generated in the last attestation
Output: $Report$: attestation report

1 **Function** Attestation**:**
2 DisableIRQ();
3 k_{att}, k_{col} = *LoadKey*();
4 /* set the time for the next attestation */
5 θ_i = *PRNG*(θ_{i-1});
6 t_{intvl} = $\theta_i \mod t_{max}$;
7 *SetTimer*(t_{intvl});
8 /* initialize the attestation */
9 *R* = *GenNonce*();
10 mac_r = *Mac*(k_{att}, R);
11 $Write\_sector(s_1, R\|mac\_r)$;
12 /* perform measurement and generate the report*/
13 $T_{rec}, mac_t = Read\_sector(s_1)$;
14 **if** (mac_t == *Mac*($k_{att}, T_{rec}\|R$)) **then**
15 mem_{prv} = *GetMem*(AR_{begin}, AR_{end});
16 mac_m = *Mac*($k_{col}, T_{rec}\|mem_{prv}$);
17 $r_t = GenReport(mac_m, T_{rec})$;
18 RestoreIRQ();
19 **return**

of the I/O request. This write request will also trigger the authentication of the attestation request on *FVM* side. The fourth step of the measurement phase is equivalent to an I/O request of reading spare sector S1, which reads the current clock value of *FVM* and its MAC back to the *Prv* side. The data involved in the protocol is exchanged through the write buffer, following the execution logic of the original firmware. As a result, the continuous execution of the above two custom I/O requests will constitute the complete protocol flow.

Algorithm 1 presents the details for the attestation procedure on *Prv* side. This algorithm will be invoked when the timer for scheduling attestation expires. As can be seen, the procedure begins by generating the random time interval t_{intvl} that trigger the next attestation (line 5-7). Then the procedure continues by generating a random challenge and sending it to *FVM* through a write request (line 9-11). Next, the attestation procedure will obtain the current timestamp T_{rec} from *FVM* through a read request (line 13). If the authentication of the response is successful, the MAC of *Prv*'s memory will be computed with the timestamp and the timestamped attestation report will be generated (line 14-17). To ensure the non-interruptibility of the protocol, interrupts are disabled during the attestation.

Extended FTL and custom code. Generally, FTL of flash firmware is responsible for parsing the incoming I/O requests from the host and perform the corresponding operations. To enforce proper operation of custom I/O requests, FTL needs to correctly identify them and execute the corresponding custom code. To achieve this, it is necessary to extend the functionality of the original FTL and add custom code in the firmware. The

details of extended FTL and custom code within FVM are presented in Algorithm 2.

Initially, FTL is in the infinite loop to continuously monitor and incoming I/O requests. When receiving an I/O request, FTL will determine which operation to perform based on the operation type (e.g., read or write) and operand (e.g., the address of the target sector). To handle custom I/O requests of reading or writing spare sectors S1, additional conditional branches are added to original FTL to handle requests of reading S1 (line 4) and writing S1 (line 9) respectively. Writing and reading spare sector S1 will trigger the call of custom firmware functions `Authentication` (line 10) and `GetTimestamp` (line 5) respectively. In the extended FTL, standard read and write requests will trigger the original read operation (line 7) or write operation (line 12), ensuring that the original firmware functionality remains unaffected.

We also design custom functions `Authentication` (line 14-22) and `GetTimestamp` (line 23-27) to handle the specific operations of custom I/O requests. The former is used for authentication of the request, while the latter is used for timestamp generation. When `Authentication` is invoked, it will first obtain the request from the write buffer and authenticate it with the attestation key (line 14-18). If the request has been authenticated, a timestamp T_{rec} will be obtained from *FVM* and the flag $auth\_flag$ will be set to 1 (line 19-21). When `GetTimestamp` is invoked, it will copy the timestamp T_{rec} to the write buffer if $auth\_flag$ has been set to 1 (line 23-25). Lastly, the control will be transferred back to the extended FTL. Note that only custom functions within *executable-only* code region can access secret key, read-only clock and other secret values, strictly adhering the access policy visualized in Fig. 4.

Firmware modification. To extend the functionality of original FTL, we apply hook technology to modify the flash firmware. Upon receiving an I/O request, FTL traverses the command index table to parse the operation type and passes the control to the corresponding handler. If a read/write request is received, the Logical Block Address (LBA) of the target sector will also be included as the operand. With that in mind, FTL extension can be achieved in four steps. First, we analyze the flash firmware and locate the command index table in the FTL code. Second, we identify the command number of read/write request based on the communication protocol and further locate the call sites of the corresponding handlers. Third, we design the hook functions that contain additional judgment conditions to detect access to S1 via its LBA. These hook functions will jump to the entry points of the custom code if the received LBA is S1, otherwise they will transfer control flow to the entry points of the original read/write handlers. Fourth, the original entry points at the call sites of the read/write handlers are overwritten to point to the hook functions.

We utilize the free space in the firmware to accommodate hook functions, custom code, and secret variables, thereby ensuring that the existing firmware code remains unaffected. According to the above access policy, the custom code belongs to the *executable-only code* region, and can only be invoked by the extended FTL (i.e., the hook functions). The keys and other secret variables belong to the *secure storage* region, and can only be accessed by the custom code in the *executable-only code* region. We check each memory access (read/write) and control-transfer (jump/call) instruction of the firmware to guarantee that no violation of the access policy occurs in the existing code. As the firmware code is free from all runtime vulnerabilities (see Section IV-B), FVM is resistant to all code injection attacks and remote memory corruption. Therefore, the memory isolation defined by the access policy on FVM will be strictly enforced at runtime. The memory isolation property further guarantees the integrity and confidentiality of secret values (e.g., secret keys) stored in the FVM firmware.

Additionally, firmware modification is compatible with the original firmware functionality and has little impact on flash read/write performance. In terms of functionality, since the spare sector address will not be accessed during the normal data reading/writing process, the initiation of custom requests does not conflict with the original data read/write operations. In terms of performance, modified firmware does not introduce significant overhead to the read/write operations of the flash device compared to the original firmware [38].

Algorithm 2: Extended FTL and custom code in *FVM*.

Input: LBA: logical block address of the received I/O request

```
1  init: auth_flag = 0, WB; /* WB: write buffer of the flash*/
2  switch operation do
3      case read do
4          if (LBA == s_1) then
5              GetTimestamp();
6          else
7              OriginalRead(LBA); /* raw read function of the firmware */
8      case write do
9          if (LBA == s_1) then
10             Authentication();
11         else
12             OriginalWrite(LBA); /* raw write function of the firmware */
13 end
14 Function Authentication:
15     MemMov(WB, &message);
16     mac_r, R = Split(message);
17     k_att = LoadKey();
18     if (mac_r == Mac(k_att, R)) then
19         T_rec = Time();
20         mac_t = Mac(k_att, T_rec||R);
21         auth_flag = 1;
22 return
23 Function GetTimestamp:
24     if (auth_flag == 1) then
25         MemMov(&T_rec||mac_t, WB);
26         auth_flag = 0;
27 return
```

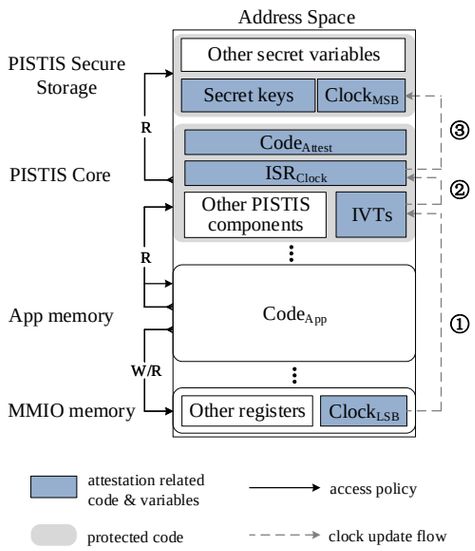


Fig. 5. PISTIS-enabled memory map of *Prv* side in FlashAttest. Access control is enforced at system startup by a secure boot mechanism.

E. PISTIS-based Prv Side Design

The *Prv* side is bulit on top of a recent software-based security architecture on low-end embedded devices: PISTIS [25]. PISTIS provides basic security features such as memory isolation and secure code update for low end embedded devices through purely software-enabled access control. To eliminate the need for hardware support, PISTIS leverages load-time verification of binaries and selective software instrumentation to guarantee the legitimacy of the deployed binary image. FlashAttest leverages some of PISTIS's security features to complement security requirements on *Prv* side summarized in Section V. In practice, this translates to deploying a new Trusted Application (TA) to PISTIS, as can be seen in Fig. 5.

The implementation of FlashAttest on *Prv* side mainly consists of trusted attestation code $Code_{Attest}$ and timer interrupt service routine ISR_{Clock}, which are deployed in the PISTIS Core memory part. $Code_{Attest}$ attests untrusted memory with key-based MAC and ISR_{Clock} implements the irregular triggering of attestation. They are write protected by PISTIS's access policy to achieve immutability. According to the aforementioned security requirements, secret attestation key k_{att} must be kept confidential and non-malleable. This is achieved by store k_{att} in PISTIS Secure Storage region so that it is readable only by $Code_{Attest}$.

A counter is required for *Prv* side to trigger the measurement when it expires after t_{intvl} period of time. We implement it as a software counter, similar to the approach in [39]. The counter consists of higher-order bits $Clock_{MSB}$ and lower-order bits $Clock_{LSB}$. As shown in Fig. 5, $Clock_{LSB}$ is set through relevant counter register to issue an timer interrupt when it wraps around (Step ①). The interrupt is handled by the trusted and integrity-protected timer interrupt service routine ISR_{Clock} depending on the Interrupt Vector Tables (IVTs) (Step ②). ISR_{Clock} maintains $Clock_{MSB}$ such that the time interval t_{intvl} can be formed by $Clock_{MSB}$+$Clock_{LSB}$. $Clock_{MSB}$ is both read- and write-protected so that it can only be accessed by ISR_{Clock}. Hence, $\mathcal{A}dv_{mob}$ who potentially resides in untrusted memory can not know the exact duration of t_{intvl}.

VI. Security Analysis

In this section, we describe the formal criteria for potential attack scenarios and discuss how FlashAttest is capable of protecting the system against them.

Attack Model. Recall that, we assume that the attacker can drop, modify, eavesdrop on and delay any message in insecure channels. It can also inject its own messages ($\mathcal{A}dv_{com}$). Furthermore, the attacker may have full control over all software and memory, and can modify *Prv*'s software ($\mathcal{A}dv_{sw}$). It can also hide all its malicious traces by restoring *Prv* to its original benign state ($\mathcal{A}dv_{mob}$). Specifically in FlashAttest, $\mathcal{A}dv_{sw}$ and $\mathcal{A}dv_{mob}$ mainly interacts with *Prv* and *FVM* in the measurement phase, while $\mathcal{A}dv_{com}$ mainly launches attacks through the communication channel. More concretely, the attacker can launch the following attacks (T):

T1) Software Attacks: In this type of attack, $\mathcal{A}dv_{sw}$ has full control over unprotected memory region of *Prv*. The attacker attempts to alter attestation code or access keys with the goal of outputting a correct response. More concretely, the attacker has several options to conduct such an attack.

T1.1.) The attacker may try to extract k_{att} and k_{col} from *Prv* or *FVM*. Then the attacker can use them to forge the attestation response.

T1.2.) The attacker may try to modify the attestation routine and redirect the measurement process to obtain a benign response based on the copy of the original memory (memory copy attacks).

T2) Communication Attacks: In this type of attack, $\mathcal{A}dv_{com}$ could eavesdrop, delay or forge messages in insecure channels. Recall that we consider the channel between *Vrf* and *Prv* and the channel between *Prv* and *FVM* as insecure channels, so the attacker has several ways to launch such an attack.

T2.1.) When *Prv* issues an request to obtain a recent clock value from *FVM*, the attacker may drop the response and replace it with his own message.

T2.2.) When *Prv* issues an request to obtain a recent clock value from *FVM*, the attacker may modify the clock value in the message.

T2.3.) When *Prv* issues an request to obtain a recent clock value from *FVM*, the attacker may record the response and replay it later.

T2.4.) When *Vrf* issues an request to obtain the attestation report from *Prv*, the attacker may drop the response and replace it with his own message.

T2.5.) When *Vrf* issues an request to obtain the attestation report from *Prv*, the attacker may modify the report in the message.

T2.6.) When *Vrf* issues an request to obtain the attestation report from *Prv*, the attacker may record the response and replay it later.

T3) Transient Attacks: Alternatively, instead of altering critical code during the attestation to obtain the ultimate

benign response, $\mathcal{A}dv_{mob}$ could try to erase its malicious traces right before the initiating of attestation. In addition to $\mathcal{A}dv_{sw}$ capabilities, the attacker needs to know the exact attestation time interval. This can be achieved via several methods:

T3.1.) The attacker may attempt to obtain the attestation time interval t_{intvl} stored on *Prv* and disinfect the malicious memory before the next attestation is triggered.

T3.2.) An advanced method is to delay the triggering of attestation until $\mathcal{A}dv_{mob}$ completes the attack and left *Prv*. This can be achieved by modifying the timing value of the relevant register, or by disabling the timer interrupt on *Prv* to prevent the control flow from passing to the attestation routine.

The security guarantees of FlashAttest are supported by both the architecture design (e.g., the PISTIS design and the *FVM* design) and protocol design. As the design correctness of architecture has been verified, we mainly verify the correctness of the protocol design. The verified security properties or assumptions are outline as following security assertions (A):

Security Assertions. Our security argument is based on the following assertions:

A1) Prv correctly implements and strictly adheres to its specifications. This implies that the system hardware specification such as CPU, memory, and all connections are correctly implemented, and that PISTIS is correctly deployed on *Prv* and its access policy strictly adheres to the design [25].

A2) FVM correctly implements and strictly adheres to its specifications. This implies that security features of the extended firmware and the access policy for each part strictly adheres to the design. Specifically, the formal verification of FVM's design correctness is given in our previous work [38].

A3) Attestation code and critical variables (e.g., attestation key and clock variables) in *Prv* and *FVM* are stored in the specified memory area, strictly adhering to the design. Specifically, the attestation code on *Prv* side is bug-free and does not contain any memory corruption vulnerabilities.

Modelling in Tamarin. In order to formally verify the design correctness of protocol, we evaluated it using the Tamarin prover [9]. Evaluation is performed by modeling the protocol in Tamarin, specifying and proving a set of lemmas based on security properties. Security in FlashAttest can be achieved based on the validity of the following security properties (P):

P1) Correctness Trace: Proving the executability of a protocol is crucial to ensure that it can be carried out as intended and other security properties do not vacuously hold.

Definition 1 (Correctness Trace (CT)). There must exist an execution path of the protocol such that each protocol step executes as designed. Additionally, the measurement process can be executed multiple times before a collect phase.

P2) Authentication Properties: The strength of authentication is typically evaluated using the hierarchy specified by Lowe [40]. Our definitions are attestation-specific instantiations of Lowe's standard authentication properties. Specifically, we refer to one-way authentication [41] to better model our protocol.

In the attest phase of our protocol, authentication of *FVM* ensures *Prv*, knows that the current clock value it receives matches the result that *FVM* sent. We therefore require *Prv* to have One-way Injective Agreement with *FVM*, resulting in the following definition:

Definition 2 (FVM Authentication (FA)). A remote attestation protocol satisfies FA iff, whenever an honest prover *Prv* completes an attestation procedure apparently with *FVM*, then *FVM* acting as responder not necessarily with *Prv*, and *Prv* and *FVM* agreed on the data for this run, and for each run of *Prv* there was a unique run by *FVM*.

In the collect phase of our protocol, authentication of *Prv* ensures *Vrf*, knows that the attestation report it receives matches the result that *FVM* sent. Similarly, we require *Vrf* to have One-way Injective Agreement with *Prv*.

Definition 3 (Prv Authentication (PA)). A remote attestation protocol satisfies PA iff, whenever an honest verifier *Vrf* completes a collection procedure apparently with *Prv*, then *Prv* acting as responder not necessarily with *Vrf*, and *Vrf* and *Prv* agreed on the data for this run, and for each run of *Vrf* there was a unique run by *Prv*.

P3) Integrity of Data: In a remote attestation protocol, it is essential to ensure that the integrity of the attestation data is preserved and cannot be altered by the adversary. We prove the integrity of the data using One-way Non-Injective Agreement.

In the attest phase of our protocol, the clock value received by *Prv* should agree with the clock value that was produced by *FVM*. This can be described as the following definition:

Definition 4 (Integrity of Clock Value (ICV)). A remote attestation protocol satisfies ICV iff, whenever an honest prover *Prv* completes an attestation procedure apparently with *FVM*, then *FVM* acting as responder not necessarily with *Prv*, and *Prv* and *FVM* agreed on the same clock value for this run.

In the collect phase of our protocol, the attestation report received by *Vrf* should agree with the report that sent by *Prv*. This can be described as the following definition:

Definition 5 (Integrity of Attestation Report (IAR)). A remote attestation protocol satisfies IAR iff, whenever an honest *Vrf* completes a collection procedure apparently with *Prv*, then *Prv* acting as responder not necessarily with *Vrf*, and *Vrf* and *Prv* agreed on the same attestation report for this run.

P4) Freshness of Data: Freshness guarantee can prevent an adversary from reusing a valid attestation response. We refer to One-way Recent Injective Agreement [41] for freshness modeling.

In the attest phase, the freshness of clock value received by *Prv* can be described as the following definition:

Definition 6 (Freshness of Clock Value (FCV)). A remote attestation protocol satisfies FCV iff, whenever an honest prover *Prv* completes an attestation procedure apparently with *FVM*, then *FVM* acting as responder not necessarily with *Prv*, and *Prv* and *FVM* agreed on the same clock value for this run, and the run by *FVM* happened after the first message by *Prv*, and for each run of *Prv* there was a unique run by *FVM*.

In the collect phase, the freshness of attestation report received by *Vrf* can be described as the following definition:

Definition 7 (Freshness of Attestation Report (FAR)). A remote attestation protocol satisfies FAR iff, whenever an honest *Vrf* completes an collection procedure apparently with *Prv*, then *Prv* acting as responder not necessarily with *Vrf*,

TABLE III
SUMMARY OF VERIFICATION RESULTS FOR OUR PROTOCOL. 'VERIFIED' INDICATES THAT THE DEFINITION IS PROVED AUTOMATICALLY BY TAMARIN, WHEREAS '-' INDICATES THAT THE GOAL IS PROVED IN PREVIOUS WORK

Security Goals	Definitions & Assertions	Result
Goal 1	*A1&A2*	-
Goal 2	*A1*	-
Goal 3	*P2-FA*	verified (14 steps)
	P3-ICV	verified (10 steps)
Goal 4	*P4-FCV*	verified (14 steps)
Goal 5	*P2-PA*	verified (21 steps)
	P3-IAR	verified (13 steps)
Goal 6	*P4-FAR*	verified (21 steps)
Goal 7	*P1-CT*	verified (64 steps)

TABLE IV
SECURITY PROPERTIES FOR MITIGATION PREDEFINED THREAT MODEL

Adversary	Attack method	Security properties for mitigation
$\mathcal{A}dv_{sw}$	*T1.1*	*A1&A2&A3*
	T1.2	*A1&A3*
$\mathcal{A}dv_{com}$	*T2.1*	*P2-FA*
	T2.2	*P3-ICV*
	T2.3	*P4-FCV*
	T2.4	*P2-PA*
	T2.5	*P3-IAR*
	T2.6	*P4-FAR*
$\mathcal{A}dv_{mob}$	*T3.1*	*A1&A3*
	T3.2	*A1&A2&A3&P3-ICV*

and *Vrf* and *Prv* agreed on the same attestation report for this run, and the run by *Prv* happened after the first message by *Vrf*, and for each run of *Vrf* there was a unique run by *Prv*.

Analysis. We captured the above definitions given in as first-order logic lemmas in Tamarin. In the attest phase, we have three lemmas to capture the definitions *FA*, *ICV*, and *FCV*. Then, we have two lemmas for *CT*. In the collect phase, we have three lemmas to capture the definitions *PA*, *IAR*, and *FAR*. Then, we have two lemmas for *CT*. The verification results are shown in Table III. The first and second columns show the correspondence between security goals and definitions. Goals 1 and 2 are proved in previous work, whereas the rest were verified using the autoprove mode of Tamarin.

Table IV shows the security properties required to mitigate the above attack methods. *T1.1* is protected based on assertion *A1*, *A2* and *A3*. The confidentiality of secret variables is protected by PISTIS's and *FVM*'s access policy so that the adversary cannot directly access the memory region where they are stored. *T1.2* is protected based on assertion *A1* and *A3*. Attestation code is stored in PISTIS Core memory part and protected by its restrictive access control policy. The adversary in app memory cannot modify it.

T2.1, *T2.2* and *T2.3* are protected based on *P2-FA*, *P3-ICV* and *P4-FCV*, respectively. Forgery, modification or replay of messages during the measurement phase will be mitigated by corresponding properties.

T2.4, *T2.5* and *T2.6* are protected based on *P2-PA*, *P3-IAR* and *P4-FAR*, respectively. Similarly, forgery, modification or replay of messages during the collection phase will be mitigated by corresponding properties.

T3.1 is protected based on assertion A1 and A3. The onfidentiality of t_{intvl}'s higher-order bits $Clock_{MSB}$ is protected by PISTIS's access policy so that the adversary cannot determine the exact time of attestation.

T3.2 is protected based on assertion A1, A3 and definition *P3-ICV*. If the adversary delays or modifies the triggering time of attestation, the timestamp obtained from *FVM* will exceed the valid time range. Since the integrity of the timestamp is protected both during storage and while transmitted over insecure channels, it cannot be modified by the adversary. When the adversary leaves *Prv*, the attestation report with abnormal timestamped will be ultimately collected by *Vrf*. Hence, the past presence of the adversary will be detected.

VII. EVALUATION

In this section, we evaluate FlashAttest's overhead and performance. We first introduce the experimental settings and discuss the implementation details. Then we present our evaluation results and analysis.

A. Experimentation Settings

***Prv* Implementation.** The *Prv* side of FlashAttest is implemented based on MSP430 architecture from Texas Instrument [42]. MSP430 architecture is widely employed in IoT domains for its low power consumption and low cost. Specifically, we choose MSP430F5529 Micro Controller Unit (MCU), which features 8kB of RAM, 132kB of FLASH, and a CPU speed that can reach up to 8 megahertz when utilizing internal oscillators.

***FVM* Implementation.** *FVM* is implemented on a PNY's Elite TF card with an independent 8-bit flash controller, and 64 GB of Multi Level Cell (MLC) NAND flash. The controller is ASolid's AS2703EN [43], a tiny system in Intel chipsets.

Software Deployment. Software deployment of FlashAttest is two-fold: deployment of *Prv* side for MSP430F5529 MCU and modified flash firmware that constructs *FVM*. We deploy PISTIS on *Prv* side and then implement the FlashAttest service as a TA in its TCM. We used HMAC-DRBG and HMAC-SHA256 from the HACL library [44] to implement PRNG and compute the digest of the application memory, respectively. The relevant implementation is modular and composed of 1336 lines of C code. *FVM* is implemented based on the modified TF card firmware along with 2296 lines of additional assembly that provides additional functionality. Among them, 948 lines of assembly provide the cryptographic primitives for FlashAttest, and the rest implement extended FTL and general functions. Since the same MAC algorithm needs to be implemented on both the *Prv* side and the *FVM* for message authentication, we choose LightMAC [45], considering the limited performance of the flash controller. LightMAC is a lightweight MAC mode of operation, offering compact authentication primitive for resource-constrained devices without sacrificing security. We instantiate LightMAC on *Prv* side and *FVM* respectively with XTEA [46], a lightweight block cipher.

TABLE V
RUNTIME OF THE MEASUREMENT PHASE IN FLASHATTEST

Operations	Time
1. Computing MAC of 32-byte nonce	11.079 ms
2. Sending 32-byte nonce and 8-byte MAC to *FVM*	4.231 ms
3&4. Recieving 4-byte timestamp and 8-byte MAC from *FVM*	20.960 ms
5. Computing hash of 1kB memory	243.931 ms
Total	280.201 ms

The *Prv* version comprises 253 lines of C code, while the *FVM* version comprises 788 lines of assembly.

Experimental Procedure. We carry out evaluation on above prototype system. Our evaluation metrics include: memory footprint, communication overhead, attestation time, and power consumption. Section VII-B reports on memory footprint. Section VII-C discusses communication overhead. Section VII-D evaluates runtime performance. Section VII-E provides energy consumption assessment. Section VII-F compares FlashAttest's overhead with several related remote attestation schemes. Lastly, the discussion on FlashAttest can be found in Section VII-G.

B. Memory Footprint

In the case of *Prv*, FlashAttest needs to store two 32-byte secret keys and the higher-order 2 bytes of 4-byte t_{intvl} (denoted by $Clock_{MSB}$) in flash memory which should be only accessible by attestation code. Also, *Prv* stores a 32-byte internal state θ of PRNG and a 4-byte protocol constant t_{max}. Additionally, the encrypted attestation report occupied another 8 bytes of flash memory. The attestation code persistently occupies 1377 bytes of the PISTIS Core region, resulting in the increase of the size of the entire PISTIS software ($\sim$19.5kB) by 6.95%. For the target MCU with around 132kB of flash memory, this constitutes about 1.02% memory overhead of the entire system.

FVM needs to store a 32-byte secret key, as well as 4-byte clock values T_{rec}. Also, *FVM* requires another 4 bytes to maintain a 32-bit clock. The attestation-related custom code is implemented as patch code and applied to the original flash firmware. The size of the attestation code is 1321 bytes, amounting to 2.26% of the available free firmware space ($\sim$58.9kB) on the target flash.

C. Communication Overhead

The communication overhead in FlashAttest primarily involves overheads between *Prv* and *FVM* via the bus during the measurement phase, as well as overheads between *Vrf* and *Prv* via the wireless network during the collection phase.

In the measurement phase, *Prv* is required to send a 32-byte random challenge along with its 8-byte MAC to *FVM*. Then *Prv* is required to receive a 4-byte timestamp and its 8-byte MAC from *FVM*. Additionally, the protocol involves two I/O operations initiated by *Prv*, each of which will send a 6-byte command packet to *FVM*, resulting in 12 bytes of communication overhead for each run. In the collection phase, the communication overhead is an encrypted 8-byte attestation report along with 32-byte MAC received by *Vrf*.

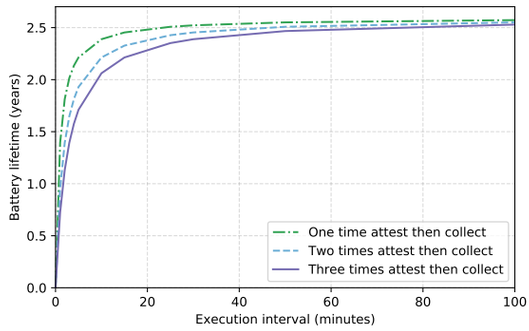


Fig. 6. Power consumption of different execution rates of FLashAttest.

D. Attestation Time

We then evaluate the runtime breakdown of the measurement phase for FlashAttest. In the experiment, the MSP430 MCU of our implementation works at a clock frequency of 8 MHz. Table V shows the detailed execution times for each operation during the measurement phase. We record the time to attest a 1kB program memory.

As shown in the first row of the table, computing the MAC of a 32-byte random challenge took approximately 3.384 ms. Sending the random challenge and its 8-byte MAC to *FVM* took approximately 4.231 ms, as shown in the second row of the table. Essentially, this step mainly involves data transmission between *Prv*'s transmit/receive buffer and *FVM* through the bus, with little computation involved. The third row of the table shows the total time from when *Prv* issued the read request to when it received the response. It is composed of communication overhead and authentication time on *FVM* side. In fact, when *Prv* issues the read operation to obtain the timestamp, it is blocked until *FVM* finishes authenticating the request and returns the timestamp. The fourth row of the table shows the time cost of step 5 in the measurement phase. The time consumption of computing the hash of *Prv*'s memory is linearly dependent on memory size, taking approximately 243.931 ms for attesting a 1kB program memory.

E. Power Consumption

Considering that IoT devices are often used in areas where access to power lines is unfeasible, we have also measured the impact factor on the power consumption. Our prototype consumes 28.1mA when performing checksum calculation, and 132μA in the low power consumption mode. Note that our prototype consumes more working current in the idle mode due to the connection of *FVM*. Figure 6 shows the impact on battery lifetime when considering various execution intervals. The prototype is powered by a 3.0V/3Ah battery. The results show that the estimated battery lifetime would last for no less than 2.5 years when performing measurement three times and then collecting reports once every 100 minutes.

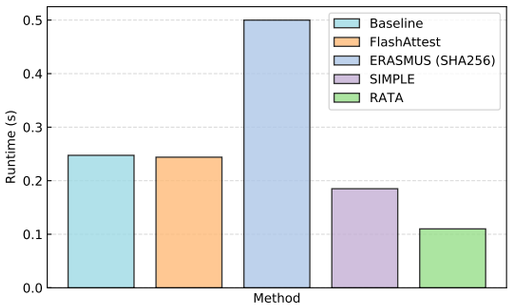
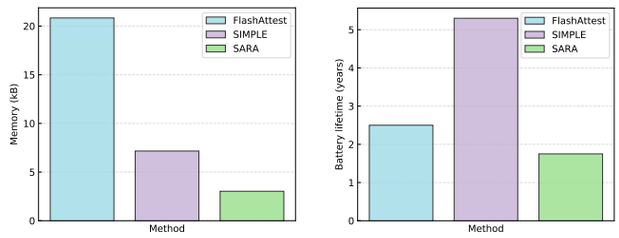


Fig. 7. Runtime (attestation per 1kB) for FlashAttest and other state-of-the-art attestation techniques. The baseline attestation time is measured from PISTIS's original remote attestation service.



(a) Memory consumption of different self-attestation schemes (b) Power consumption of different self-attestation schemes

Fig. 8. Comparison between self-attestation schemes in memory consumption and power consumption

F. Comparison with Other Related Schemes

To provide a more detailed comparison, we compare our experimental results with several related attestation schemes: SIMPLE [24]: a software-based self-attestation method based on a formally-verified memory isolation architecture [26]. SARA [5]: a self-attestation scheme for secure asynchronous attestation. ERASMUS [4]: a hybrid self-attestation implemented based on SMART [8]. RATA [30]: a formally-verified hybrid attestation protocol. Among these technologies, SIMPLE [24], SARA [5], and ERASMUS [4] have similar design goals to FlashAttest, all of which achieve self-attestation. RATA [30] is also a related approach to FlashAttest, even though it does not achieve self-attestation. In addition to the similarity of protocol design to FlashAttest, its security properties are also enforced by verified implementation.

Fig. 7 compares the attestation time of FlashAttest with aforementioned techniques. To make the comparison easier, we present the results of all techniques for performing attestation on 1kB of program memory at a CPU speed of 8 MHz. Specifically, we compared FlashAttest with PISTIS's original remote attestation service, SIMPLE [24], ERASMUS [4] and RATA [30].

As shown in the figure, FlashAttest achieves similar attestation time compared to baseline design. This is primarily because both FlashAttest and PISTIS's original remote attestation service are implemented based on HMAC-SHA256 from the HACL library [44]. Similar to prior work, checksum computation loop spends the majority of the runtime in the whole attestation procedure. FlashAttest achieves around 2x faster attestation time compared to ERASMUS [4] while eliminating the need for hardware modifications. FlashAttest achieves a relatively similar runtime overhead result compared to SIMPLE [24], because they are both software-based. Compared to RATA [30], FlashAttest achieves an affordable runtime overhead result. The overhead is slightly higher due to the application of the software-based memory isolation architecture (as opposed to its hardware/software co-design in RATA).

Fig. 8 compares FlashAttest to SIMPLE [24] and SARA [5] in terms of memory consumption and power consumption. In the memory consumption comparison, we include the memory overhead of the software isolation architecture in FlashAttest and SIMPLE. The results show that FlashAttest requires more storage due to the deployment of PISTIS core functions, compared to both SIMPLE and SARA. In the power consumption comparison, we unify the execution interval of the attestation in FlashAttest and SIMPLE to 100 minutes. The estimated lifetime of a 3.0V/3Ah battery used in different schemes indicates that FlashAttest consumes less energy than SARA and more than SIMPLE.

Summary. We finish this section by summarizing the comparisons among different schemes. Overall, FlashAttest achieves similar runtime overhead compared to exiting self-attestation schemes while adding security properties for ensuring legitimate operations (compared to SIMPLE [24]) and achieving strong security guarantees supported by a formally verified design.

FlashAttest introduces more memory overhead, but we think this is a reasonable trade-off for eliminating the need for hardware modifications. Further, FlashAttest consumes less energy than SARA [5], which confirms that FlashAttest is an appropriate attestation protocol for low-end IoT devices.

While traditional software-based schemes such as SWATT [18] and VIPER [15] do not depend on any verified hypervisor or hardware properties, they cannot meet the security properties required for self-attestation. Existing self-attestation rely on classic hybrid architectures such as SMART [8] and TrustLite [47] to provide a hardware-protected clock. Compared to these schemes, obtaining a clock value in FlashAttest is more time-consuming due to the communication delay between the host and the flash device. Nevertheless, FlashAttest obtains hardware security properties while preserving the advantage of not requiring hardware modification or extension to already deployed IoT devices in traditional software-based schemes. Furthermore, the significantly (around 2x) faster in attestation time compared to ERASMUS [4] also indicates that FlashAttest is more efficient than SMART [8].

G. Discussion

Attack surface analysis. We further discuss possible attack surfaces of FlashAttest in actual deployment. Since the TCM code and the FVM firmware are free from all runtime vulnerabilities, only the vulnerabilities in the untrusted application region of the prover device can be exploited to launch code injection or memory corruption attacks. To compromise the

prover device, an attacker must find a vulnerability that can overwrite or delete the TCM (i.e., protected memory region). According to the protection strategy of PISTIS, any untrusted software to be run on the prover device should be deployed and verified through the verification module of TCM. The software is considered unsafe and rejected outright if it contains instructions that attempt to compromise the protected memory region. Therefore, the running software may contain bugs, none of them will violate the access policies enforced by FlashAttest. Another available attack surface is the insecure channels of FlashAttest. An attacker can not only control the communication channel between the verifier and the prover, but can also manipulate the messages exchanged between the prover and the flash device in some way. The robustness of FlashAttest in this case has been formally verified using the Tamarin prover.

Additionally, the TOCTOU vulnerability can also be exploited by an attacker. We simulate TOCTOU attack process on FlashAttest by using a lemma in Tamarin:

```
lemma toctou: "∃ verifier prover #i #j .
  Commit_ColRes(verifier, prover, onee)@ #i &
  Untrusted(prover) @ #j &
  #j < #i &
  ¬(∃ A #k. Key_Reveal(A) @ #k)"
```

The lemma shows that even if the `verifier` has obtained a benign attestation result (denoted by `onee`) at time #i (label Commit_ColRes), there is a possibility that the `prover` was compromised (label Untrusted) at the previous time #j. We run this lemma together with our protocol model in Tamarin and the results show that this lemma has been verified, indicating that the TOCTOU problem exists in FlashAttest.

One potential solution is to monitor any changes to the program memory, even between consecutive measurement instances. This can be achieved by integrating FlashAttest with memory shadowing techniques. Specifically, shadow memory is allocated for the application memory region to record the access status of each byte in the corresponding application memory. Whenever any location within the application memory region is written, the corresponding shadow memory is marked with a specific value to indicate that the region has been modified. Consequently, the verifier can determine whether the application memory region has been modified since the last successful attestation request by examining the markers in the shadow memory. The PISTIS-enabled toolchain can be leveraged to replace the original write access function with a virtualized version that synchronously updates the shadow memory state corresponding to the application memory, thus enabling modification monitoring of application memory between two consecutive attestations.

Extensibility. Our design is implemented atop of commercial flash device, suitable for a variety of current flash devices. By now, we have implemented firmware modifications and function expansion on three categories of flash memory devices, including TF cards, USB flash drives and SSDs [48]. This shows that FlashAttest has good extensibility and applicability in actual deployment. Besides, implementing the proposed scheme on flash memory with a higher clock frequency can achieve more efficient MAC computation and authentication.

In addition, our design has good portability and can be applied to various IoT platforms. The target IoT platform can obtain the security features provided by FVM as long as it can normally communicate with the flash device through the bus. The minimal security property required for the target platform is to guarantee the immutability of the attestation code. Most IoT platforms can meet this security requirement: either by the microcontroller's security extensions or by deploying software-based memory isolation architectures. For example, FlashAttest can be ported to the IoT platform with an ARM Cortex-M microcontroller in three steps. First, adjust the attestation code for interactions with FVM to align with the communication protocol between the target IoT platform and the flash device. This is to ensure that custom I/O requests function correctly because communication protocols may vary between different platforms and flash devices. Second, deploy the attestation code in the secure world offered by the processor's TrustZone extension. Third, adjust the firmware code of FVM in accordance with the communication protocol to ensure that it can correctly parse custom I/O requests.

Similar to prior work, the majority of the runtime in FlashAttest is spent in the hash computation loop. Therefore, porting FlashAttest to other IoT platforms of different clock frequencies may cause its performance to change.

Scaling to collective attestation. FlashAttest is possible to scale to a collective attestation scheme that verifies the software state of a group of IoT devices. Clock synchronization can be challenging because the verifier needs to verify each prover's attestation reports, which are timestamped by its own *FVM*. Logical clock synchronization [5] is a promising direction to address the clock synchronization problem, which does not require precise timestamping of messages. In FlashAttest, each flash module can maintain a vector clock that is updated during a self-attestation execution.

VIII. Conclusion

In this paper, we proposed a novel attestation protocol called FlashAttest, which achieve self-attestation while eliminating the requirement for hardware modifications. The key insight was to introduce an additional flash-based verification module to provide required security features. We further presented the details of the protocol and summarized the security goals of the system. We showed the design of the flash-based verification module and how its security features could be used to complete protocol interaction.

We formally analyzed the security of our protocol using Tamrin prover and implement a prototype for effectiveness evaluation. It takes approximately 243.931 ms for FlashAttest to attest a 1kB program memory. Compared to the state-of-the-art, experimental results demonstrate that our approach achieves affordable runtime overhead, reasonable energy consumption, and good extensibility while eliminating the need for hardware support.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (no. U2336201).

REFERENCES

[1] Statista. (2024) Internet of things—number of connected devices worldwide 2022–2033. [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[2] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," Tech. Rep., 2014.

[3] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "Seed: secure non-interactive attestation for embedded devices," in *Proceedings of the 10th ACM conference on security and privacy in wireless and mobile networks*, 2017, pp. 64–74.

[4] 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)*. IEEE, 2018, pp. 1191–1194.

[5] 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.

[6] 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, pp. 99–110.

[7] 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*, 2014, pp. 1–14.

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

[9] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, ser. CAV 2013. Berlin, Heidelberg: Springer-Verlag, 2013, p. 696–701.

[10] W. Arthur, D. Challener, and K. Goldman, *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015.

[11] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487726.2488368

[12] A. ARM, "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[13] 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*, 2016, pp. 743–754.

[14] H. Tan, W. Hu, and S. Jha, "A remote attestation protocol with trusted platform modules (tpms) in wireless sensor networks." *Security and Communication Networks*, vol. 8, no. 13, pp. 2171–2188, 2015.

[15] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 3–16.

[16] 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*, ser. WiSe '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 85–94. [Online]. Available: https://doi.org/10.1145/1161289.1161306

[17] 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*, 2005, pp. 1–16.

[18] 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.* IEEE, 2004, pp. 272–282.

[19] 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*, 2009, pp. 400–409.

[20] U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *USENIX Security Symposium*, vol. 8, no. 3, 2004.

[21] J. Cao, T. Zhu, R. Ma, Z. Guo, Y. Zhang, and H. Li, "A software-based remote attestation scheme for internet of things devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1422–1434, 2022.

[22] S. Surminski, C. Niesler, F. Brasser, L. Davi, and A.-R. Sadeghi, "Realswatt: Remote software-based attestation for embedded devices under realtime constraints," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2890–2905.

[23] S. Kumar, P. Eugster, and S. Santini, "Software-based remote network attestation," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 5, pp. 2920–2933, 2021.

[24] 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)*. IEEE, 2020, pp. 247–258.

[25] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo, "{PISTIS}: Trusted computing architecture for low-end embedded systems," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3843–3860.

[26] 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.

[27] 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*, 2015, pp. 1–6.

[28] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with {CUstomizable} and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1073–1090.

[29] F. Arkannezhad, J. Feng, and N. Sehatbakhsh, "Ida: Hybrid attestation with support for interrupts and toctou," in *Network and Distributed System Security (NDSS) Symposium*, 2024.

[30] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the toctou problem in remote attestation," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2921–2936.

[31] 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)*, 2019, pp. 1429–1446.

[32] L. Kang, Y. Xue, W. Jia, X. Wang, J. Kim, C. Youn, M. J. Kang, H. J. Lim, B. Jacob, and J. Huang, "Iceclave: A trusted execution environment for in-storage computing," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 199–211.

[33] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, "Flashguard: Leveraging intrinsic flash properties to defend against encryption ransomware," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2231–2244.

[34] P. Cronin, X. Gao, H. Wang, and C. Cotton, "Time-print: Authenticating usb flash drives with novel timing fingerprints," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1002–1017.

[35] P. Poudel, B. Ray, and A. Milenkovic, "Flashmark: Watermarking of nor flash memories for counterfeit detection," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[36] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, "Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 33–47.

[37] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on information theory*, vol. 29, no. 2, pp. 198–208, 1983.

[38] Z. Zhang, J. Xue, T. Chen, Y. Zhao, and W. Meng, "Flash controller-based secure execution environment for protecting code confidentiality," *Journal of Systems Architecture*, p. 103172, 2024.

[39] F. Brasser, K. B. Rasmussen, A.-R. Sadeghi, and G. Tsudik, "Remote attestation for low-end embedded devices: the prover's perspective," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[40] G. Lowe, "A hierarchy of authentication specifications," in *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*, ser. CSFW '97. USA: IEEE Computer Society, 1997, p. 31.

[41] J. Wilson, M. Asplund, N. Johansson, and F. Boeira, "Provably secure communication protocols for remote attestation," in *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ser. ARES '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3664476.3664485

[42] J. H. Davies, *MSP430 microcontroller basics*. Elsevier, 2008.

[43] Asolid. (2024) Introduced sd 3.0 controller as2703en. [Online]. Available: https://asolid-tek.com/en/introduced-sd-3-0-controller-as2703en/

[44] 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*, 2017, pp. 1789–1806.

[45] A. Luykx, B. Preneel, E. Tischhauser, and K. Yasuda, "A mac mode for lightweight block ciphers," in *Fast Software Encryption: 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers 23*. Springer, 2016, pp. 43–59.

[46] R. M. Needham and D. J. Wheeler, "Tea extensions," *Report (Cambridge University, Cambridge, UK, 1997)*, 1997.

[47] 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*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592824

[48] Z. Zhang, J. Xue, T. Baker, T. Chen, Y.-a. Tan, and Y. Li, "Cover: Enhancing virtualization obfuscation through dynamic scheduling using flash controller-based secure module," *Computers & Security*, vol. 146, p. 104038, 2024.