ITERATOR: interruptible remote attestation through cuckoo filters

Sponziello, N.; Sateesan, A.; Rabbani, M.M.; Mentens, N.; Dragoni, N.; Dushku, E.

Citation

Sponziello, N., Sateesan, A., Rabbani, M. M., Mentens, N., Dragoni, N., & Dushku, E. (2025). ITERATOR: interruptible remote attestation through cuckoo filters. *Ieee Internet Of Things Journal*, *12*(24), 54746-54758. doi:10.1109/JIOT.2025.3621016

Version: Publisher's Version
License: Licensed under Article 25fa Copyright Act/Law (Amendment Taverne)
Downloaded from: https://hdl.handle.net/1887/4303516

Note: To cite this publication please use the final published version (if applicable).

ITERATOR: Interruptible Remote Attestation Through Cuckoo Filters

Nicoló Sponziello, Arish Sateesan, *Member, IEEE*, Md Masoom Rabbani, Nele Mentens, *Senior Member, IEEE*, Nicola Dragoni, and Edlira Dushku, *Member, IEEE*

Abstract—**Remote attestation (RA) is emerging as a promising security mechanism that establishes trust in Internet of Things (IoT) devices by detecting malware presence. Typically, RA consists of computing a hash over the device's memory and is executed as an *atomic* procedure to guarantee the reliability of the attestation evidence. However, in real-world situations, such as those involving real-time systems, energy-harvesting devices, or mission-critical operations, the IoT device may not be able to complete the attestation procedure due to various factors like task scheduling, limited battery life, or higher priority tasks. In such scenarios where flexibility, adaptability, and security are paramount, enabling *interruptibility* of RA is crucial. This article presents a novel approach called ITERATOR, which leverages hash-based storage to enable interruptible RA without any additional hardware requirements. Our proposal transforms the device attestation procedure from the traditional approach of memory hash computation to a lookup operation in a hash-based storage, namely, a Cuckoo filter. The ITERATOR protocol divides the device's memory into blocks associated with a Cuckoo filter bucket. This approach allows the device to perform RA in multiple rounds, ensuring secure interruptible attestation. We perform software simulations of ITERATOR, demonstrating its high effectiveness in detecting malware presence. Due to its interruptible design, ITERATOR cannot guarantee 100% detection in a single attestation round; however, repeated rounds make long-term evasion by malware highly unlikely. In particular, the experiments showed that the probability of evading the detection ranges between 37% and less than 1%, depending on the protocol configuration. Moreover, we validate ITERATOR's efficiency through two hardware proof-of-concept (PoC) implementations that rely on ESP32 and FPGA platforms. The FPGA implementation shows the high efficiency of the protocol, with 34.3 ns to attest a single memory block.**

Index Terms—**Cuckoo filter, Internet of Things (IoT) security, interruptible attestation, remote attestation (RA).**

Received 16 September 2025; accepted 9 October 2025. Date of publication 13 October 2025; date of current version 8 December 2025. *(Corresponding author: Edlira Dushku.)*

Nicoló Sponziello and Nicola Dragoni are with the Technical University of Denmark, 2800 Lyngby, Denmark (e-mail: nicolosponziello@gmail.com; ndra@dtu.dk).

Arish Sateesan is with the Department of Mobile Communications and Computing, RWTH Aachen University, 52062 Aachen, Germany (e-mail: arish.sateesan@rwth-aachen.de).

Md Masoom Rabbani is with the Chalmers University of Technology, 412 96 Gothenburg, Sweden, and also with the University of Gothenburg, 405 30 Gothenburg, Sweden (e-mail: mdmasoom.rabbani@chalmers.se).

Nele Mentens is with ES&S, COSIC, ESAT, KU Leuven, 3000 Leuven, Belgium, and also with LIACS, Leiden University, 2311 EZ Leiden, The Netherlands (e-mail: nele.mentens@kuleuven.be).

Edlira Dushku is with Aalborg University, 2450 Copenhagen, Denmark (e-mail: edu@es.aau.dk).

Digital Object Identifier 10.1109/JIOT.2025.3621016

I. Introduction

The Internet of Things (IoT) revolution has led to a significant increase in the number of IoT devices across various domains, such as transportation, healthcare, industrial systems, military applications, etc. As these devices become more interconnected and capable of controlling the environment, IoT systems are becoming a prime target for cyber attacks. Examples of such attacks include the Stuxnet [1], Mirai [2], and smart TV hacks [3], among others. Given the rapid expansion of the attack surface in IoT devices, there is a need for novel security measures to protect against such threats.

To detect the malware presence in IoT devices, remote attestation (RA) is gaining wide attention as a security mechanism that enables the integrity verification of a remote untrusted device. RA mechanism plays a fundamental role in ensuring trust in IoT systems [4]. Generally, RA starts with a trusted entity called *Verifier*, which sends an attestation request to an untrusted device called *Prover*. Upon receiving the attestation request, the Prover stops the regular device operation, performs attestation (typically a hash of the memory), and sends the attestation result to the Verifier.

Traditionally, RA has been implemented as an atomic procedure that cannot be interrupted once it has started in order to prevent attackers from forging the attestation result [5], [6]. Consequently, the device is required to stop the regular operation until the attestation has ended. This solution is generally applicable for many IoT devices that can provide (some degree of) tolerance for operation disruptions or have enough computational resources to complete attestation. However, such an atomic approach is impractical for IoT devices deployed in real-time systems, energy-harvesting IoT devices, or devices that perform high-priority tasks in mission-critical operations. In such cases, where the device's availability is important or the device follows an intermittent execution scheme, it is challenging to execute RA atomically [7], [8].

By removing the atomicity requirement to make the RA procedure interruptible, the protocol becomes susceptible to attacks, such as *mobile adversary* [9]. The mobile adversary aims to remain undetected in the system for as long as possible and can exhibit two main types of behaviors: *transient* or *migratory*. Transient malware aims to avoid detection by deleting itself, while migratory malware relocates itself in different parts of the memory.

2327-4662 © 2025 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

In the context of the aforementioned challenges, this article aims to design a novel interruptible RA protocol. The proposed solution is ITERATOR, which leverages a Cuckoo filter—a probabilistic data structure that represents sets of elements and resolves set membership queries. Thus, this article transforms the device attestation procedure from the traditional approach of memory hash computation to a lookup operation in the Cuckoo filter. The proposed protocol divides the attestation of device memory into multiple rounds. During each round, the Cuckoo filter checks each block, and if the query returns a positive answer, the block is considered genuine. This approach is restricted to IoT devices with static code and, due to its interruptible design, cannot guarantee a 100% detection rate, unlike conventional attestation. Nevertheless, while malware may evade a single round, it is unlikely to remain undetected across repeated attestations. The main contributions of this article are summarized as follows.

1) We design ITERATOR as a novel approach that solves the problem of interruptible attestation using hash-based storage. The proposed approach converts the memory hash calculation of traditional RA into a lookup operation in Cuckoo filters.
2) We design a lightweight interruptible attestation without requiring additional specific hardware components. In particular, ITERATOR employs self-attestation by dividing memory into blocks and verifying them over multiple rounds, allowing the process to be interrupted when necessary.
3) We provide a software simulation of ITERATOR, showing that the attacker invasion probability ranges from 37% to under 1%, depending on protocol configuration.
4) We validate ITERATOR through two proof-of-concept (PoC) implementations: 1) on an ESP32, representing a resource-constrained environment, and 2) on an FPGA, demonstrating high-performance potential. The FPGA implementation attests a single memory block in 34.3 ns.

II. Related Works

This section summarizes the state-of-the-art RA protocols in the IoT domain, focusing on their interruptibility aspects.

A. Hybrid RA

Hybrid RA is a popular approach in the literature, especially for IoT devices, due to its ability to work with minimal hardware requirements. Hybrid RA protocols are designed to ensure code and memory isolation by utilizing software/hardware co-design with minimal hardware support. Some of the most popular hybrid RA platforms in the literature are SMART [5], TrustLite [10], and TyTan [11]. SMART [5] derives its security from two main properties: *atomic execution* and *key secrecy*. To ensure security, the attestation routine cannot be interrupted, and system interrupts are disabled during attestation. This design prevents malware from modifying memory during the measurement computation, but it also means that regular device operations are blocked while attestation is running. Unlike SMART, the TrustLite [10] and TyTAN [11] architectures offer solutions that allow all system tasks, including attestation, to be interrupted. However, compared to SMART, they require additional hardware components. These interruptible approaches have limitations in detecting mobile adversaries who try to evade detection by relocating or deleting themselves. TrustLite does not address the detection of mobile adversaries at all. In TyTAN, the attestation can be interrupted by a higher-priority task, while the attested task is prevented from execution. However, this design does not provide protection against malware spreading across different tasks. Furthermore, when the system has only one task, the attestation becomes noninterruptible [13]. Different from the aforementioned works, Different from traditional on-demand schemes, ERASMUS [12] performs attestation through self-measurement, where the prover initiates the process at specific time intervals. This improves the detection of mobile malware and enables scheduling the attestation efficiently around more critical tasks. However, ERASMUS does not support interruptible attestation. To counter mobile threats, it uses irregular intervals to make attestation timing unpredictable.

B. Interruptibile Attestation

SMARM [6] proposes a lightweight, interruptible attestation system that can detect the presence of mobile malware with probabilistic guarantees. The system divides the prover's memory into smaller blocks, which can be attested atomically. SMARM uses a shuffled measurement technique to perform interruptible attestation. The order of attestation is kept secret and randomized, making it difficult for malware to predict which block will be measured next, thereby reducing the probability of evading detection. The system computes the hash of each memory block to perform attestation, but it does not consider transient malware behavior. In addition, hashing a memory block may require a substantial amount of time. Finally, the shuffled-measurement approach may be limited in detecting migratory malware, since the probability of relocating to an already attested block increases as the attestation progresses. To improve SMARM, the HAtt protocol [7] proposes a different approach for memory attestation. Specifically, instead of computing the hash, HAtt uses a physical unclonable function (PUF) and the seeds sent by the verifier to randomly select some bits from each word of each memory block. These selected bits are appended to a bit string to form the attestation result. However, HAtt requires an additional hardware component (i.e., PUF) and the attestation result in HAtt is relatively long. The RESERVE approach [8] involves attesting one or more memory blocks by performing a hash computation. During the attestation of a module, the protocol randomly selects a line of code and stores it securely to protect against memory modifications across interruptions. However, both HAtt and RESERVE only verify a small part of the module and, therefore, cannot detect malware that modifies other parts of the module that these schemes have not checked.

C. Discussion

Table I summarizes the aforementioned RA protocols, highlighting their interruptibility, adversary models, and hardware requirements. This article introduces ITERATOR as

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

TABLE I
SUMMARY OF THE APPROACHES

RA Protocol	Interruptible	Additional components	Transient malware	Migratory malware	Attestation approach
SMART [5]	✗	Hardware Bus	✓	✓	Hash
TrustLite [10]	✓	EA-MPU	✗	✗	Hash
TyTAN [11]	✓	EA-MPU	✗	◐	Hash
SMARM [6]	✓	-	✗	✓	Hash + Shuffle measurements
ERASMUS [12]	✗	-	✓	✓	Hash
RESERVE [8]	✓	checkpoints	◐	◐	Hash and random code line
HAtt [7]	✓	PUF	✗	◐	bit string
ITERATOR	✓	-	✓	✓	Cuckoo filter lookup

a lightweight approach based on Cuckoo filters. By using such a hash-based storage, ITERATOR shifts attestation from memory hash computation to a lookup operation. The goal is to design a protocol that can be securely interrupted and detect mobile adversaries without extra hardware.

III. BACKGROUND

A Bloom filter is an efficient probabilistic data structure that is used to resolve set membership queries [14]. It is composed of an array of m bits, indexed by a set of k independent noncryptographic hash functions. Each element in the set is mapped to a subset of k locations in the array by the hash functions. During the insertion of an element, the bit in these k locations is flipped to one. Similarly, to check whether an item belongs to the set of elements, the k locations in the array are identified and a positive response is returned only if all of them contain a one bit. A query can return a false positive but not a false negative.

The false positive rate can be controlled by changing the length of the bit array and the number of hash functions.

The overall query latency is proportional to the number of hash functions employed. Moreover, Bloom filters do not support deletion operations.

The Cuckoo filter is another probabilistic membership query data structure that aims to improve the performance of the Bloom filter [15]. It consists of an array of buckets indexed by independent hash functions, and each bucket consists of b entries. Each entry to the bucket is a fingerprint of the element to be added. To map an element to the corresponding bucket, the Cuckoo filter uses two hash functions: one to calculate its main bucket and a second one to find its alternate position if the first bucket is full. If neither bucket is empty, the filter executes a series of evictions by relocating a random element from the designated bucket to its alternate location.

Unlike Bloom filters, the Cuckoo filter can deny an entry if either the memory is full or the maximum number of evictions is reached before finding an empty bucket. Compared to Bloom filters, the Cuckoo filter is more space-efficient and supports deletion operations. In addition, a Cuckoo filter has a constant query latency, irrespective of the size of the filter. Cuckoo filters proved to be more efficient than Bloom filters when the desired false positive rate is below 3% [15].

A. Our Choice

We use the Cuckoo filter because it exhibits the best false positive rate for a given memory capacity compared to alternative data structures. In addition, it offers the flexibility of dynamic updates and deletions. It also has a constant query time of only two clock cycles. The false positive rate of the Cuckoo filter is determined by the number of buckets, fingerprint size, and the load factor, which measures how full the filter is. We can maintain the required false positive rate by adjusting the load factor and the fingerprint size, thereby ensuring high accuracy.

One could think of relying on alternative data structures such as Cuckoo hashing [16] or traditional hash tables [17], [18]. However, they require substantial memory due to the need to store keys. In addition, the susceptibility of traditional hash tables to collisions significantly increases the false positive rate.

B. Choice of Hash Functions for the Filters

In selecting a hash function for probabilistic data structures such as Cuckoo and Bloom filters, both implementation speed and collision probability are critical factors. It is important to distinguish that the hash function for a probabilistic filter serves as a high-speed indexing mechanism, not as a cryptographic primitive. The essential properties are therefore uniform distribution and strong avalanche properties to minimize the likelihood of collisions, rather than providing cryptographic security guarantees.

While cryptographic hashes like SHA-256 offer stronger collision resistance, their high latency and resource costs are unsuitable for this per-access indexing role. Instead, we employ lightweight noncryptographic hashes such as FNV, MurmurHash, and Xoodoo-NC, which are designed for efficiency in hardware while still offering robust distributional properties. Moreover, the collision probability and false positives in probabilistic data structures are managed structurally by maintaining a low filter load factor, i.e., the ratio of stored elements to available buckets. In our design, the load factor is kept sufficiently low, given the limited number of memory blocks to construct the Cuckoo filter in ITERATOR. This approach does not undermine the overall cryptographic security of the attestation scheme, which is guaranteed by the separate keyed MAC module, while the Cuckoo filter benefits from the efficiency of noncryptographic hashing.

IV. SYSTEM MODEL

In order to design an interruptible RA protocol, we consider a system model composed of the following three entities.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

Prover: The Prv is an untrusted low-end IoT device. The Prv's program memory is logically divided into *blocks* of predefined size. Each block represents the minimum amount of data that is processed at a time.

Network Operator: The OP is a trusted entity, responsible for the secure setup and deployment of the system.

Verifier: The Vrf is a trusted external entity that aims to check the attestation results produced by the Prv and detects whether the Prv has been compromised or not. Vrf may be different from *OP*. Aligned with RA schemes in the literature, the Vrf knows in advance the legitimate program binaries of the Prv. In our system model, the Vrf can be both a passive or an active entity. When the Prv performs a self-attestation, the Vrf is a passive entity, whose responsibility is to collect and verify the Prv's results. In addition, the Vrf can be an active entity by requesting the Prv to perform the memory attestation on demand.

V. Adversary Model and Security Requirements

In this section, we describe the adversary capabilities and the security requirements for the interruptible RA protocol.

A. Adversary Model

The main goal of the adversary is to compromise the IoT device and evade detection. In line with the adversary model described in [19], we consider the following potential actions an adversary might take against the device. Although mobile adversaries are typically considered out of scope in conventional RA protocols, we explicitly include them in our adversary model. This is because our protocol supports *interruptibility*, a feature that inherently increases susceptibility to mobile adversaries, as also addressed in [6], [7], and [8].

1) Software Adversary: A remote software adversary aims to inject malware into the Prv's memory and modify the memory's content in order to disrupt or alter the Prv's normal operations. In addition, the adversary can produce a malicious behavior in the Prv by swapping two memory blocks without modifying their content.

2) Communication Adversary: The adversary has full control of the communication channel, and can intercept, create, modify, and replay any messages that are sent between the Prv and the Vrf.

3) Mobile Adversary: A mobile adversary, in addition to its malicious operations, aims to prolong its presence in the Prv by exploiting interruptions in the attestation procedure. The mobile adversary can exhibit one of the following types of behaviors.

1) *Migratory Malware:* Migratory malware is a type of mobile adversary that can move itself in the memory to avoid detection by relocating itself in the memory area that has already been attested. This can be done while the attestation process is interrupted, and before it resumes.
2) *Transient Malware:* A transient malware is a type of mobile adversary that tries to avoid detection by deleting itself from the memory, with the purpose of re-infecting the device at a later time.

4) Assumptions: In line with other RA schemes [6], [9], [19], [20], physical adversaries and denial-of-service attacks are considered out of scope. Moreover, runtime attacks that compromise the device without modifying program memory are also excluded from the scope of this work. We further assume that hardware-protected memory regions remain secure and cannot be compromised by the adversary.

B. Security Requirements

In the context of the aforementioned system model and adversarial actions, ITERATOR should satisfy the following security requirements.

Freshness: The protocol should be able to identify a compromised Prv which reports an old legitimate state of a memory block to evade detection of an ongoing attack.

Tampering Detection: The protocol should provide reliable evidence to guarantee the integrity of the Prv's memory blocks. In this way, the protocol should detect any unauthorized modification of the Prv's memory, including the swapping of the blocks.

Mobile Adversary Detection: The protocol should detect the presence of any adversary that moves within Prv's memory (migratory behavior) or deletes itself to reinfect the Prv later (transitory behavior).

Integrity of the Results: The protocol should detect any modification to the attestation result that is stored in the Prv.

Communication Integrity and Authenticity: The protocol should ensure that the messages exchanged between Prv and Vrf are not modified or replayed by an adversary.

C. Device Requirements

In the design of the protocol, we assume that the underlying architecture of the prover has the following components, which are common in the state-of-the-art approaches [6], [8], [9].

Read-Only Memory: To prevent any unauthorized modification, it is crucial to ensure that the memory is secure and cannot be tampered with. This can be achieved by using a ROM or by setting specific memory protection unit (MPU) rules. ITERATOR will be stored in this secure memory.

Secure Storage: The secure storage protects the stored information from unauthorized access. It can be implemented via an MPU, a chip that allows defining access control rules for different memory areas. In ITERATOR, the secure storage contains the Cuckoo filter and the secret key used for the RA protocol, which can be accessed only by the attestation code.

Real-Time Clock: The real-time clock is a hardware component that keeps track of the time (monotonically increasing). Commonly, to the state-of-the-art, we assume the clock cannot be modified via software.

Cryptographically Secure Pseudorandom Number Generator (CSPRNG): It is a random number generator whose output sequence cannot be predicted. It is used for initiating the attestation so that the adversary cannot predict which number will be generated next.

VI. Our Proposal: ITERATOR

ITERATOR aims to design an interruptible RA protocol for lightweight IoT devices. Overall, ITERATOR consists of

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

TABLE II
NOTATION SUMMARY

Term	Description
Vrf	Verifier
Prv	Prover
OP	Network Operator
key	shared key between Vrf and Prv
n	total number of memory blocks
k	number of memory blocks to check randomly during an attestation round
T_{att}	attestation period
T	current time
t_{att}	random time interval added to the current time to decide next attestation time
T^{end}	time when attestation ended
p	attestation progress has reached to block p
T_{coll}	time interval when the Vrf collects the Prv's results
res	attestation result
Procedure	**Description**
$attest()$	Perform the ATTEST action
$check()$	Perform the CHECK action
$random_time_interval()$	Generates a random time interval
$filter_contains(index, block)$	Queries the filter
$store_res(res)$	Stores the attestation result
$mac \leftarrow MAC(m; key)$	outputs a MAC tag mac on input of m and key

three main phases: 1) initialization; 2) attestation; and 3) verification. We present the notation of ITERATOR in Table II, and in the following, we describe each of the ITERATOR's phases.

A. Initialization

Initialization is an offline phase executed by the OP before the attestation procedure. This phase aims to ensure the correct setup and secure deployment of the IoT device. During this phase, the OP first ensures that the IoT device satisfies the device requirements described in Section V-C and then installs the secure device application and attestation protocol. Since the attestation protocol will be interruptible, the OP defines the configuration parameters of the attestation protocol, e.g., the size of the memory blocks *BlockSize*, the time interval for performing attestation T_{att}, and the k random number of blocks that will be checked during each attestation round.

1) Cuckoo Filter Setup: In ITERATOR, the attestation protocol is associated with a Cuckoo filter that the OP constructs during the initialization phase. In particular, the construction involves inserting each legitimate block of memory (i.e., in a pre-defined block size) in the Cuckoo filter. Once completed, the filter (i.e., initialized with the benign state of the Prv's memory blocks) is deployed in the Prv's protected memory.

During the construction of the filter, each memory block is inserted in the data structure together with the index of the block, following the Cuckoo filter design described in Section III. This links the legitimate state of a memory block with its position in the memory array representation.

2) Key Setup: In addition, the OP generates a secret key and shares it with both the Prv and the Vrf. The key is securely stored in the Prv's protected memory in order to be securely accessed by the attestation code. Please note that the proposed protocol is agnostic to the underlying key management scheme, so the shared symmetric key *key* can be replaced with an asymmetric key-pair (pk, sk) without affecting the protocol details.

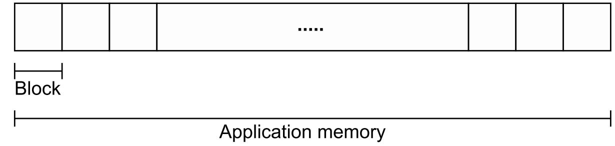

Fig. 1. Representation of the Prover's memory division in blocks.

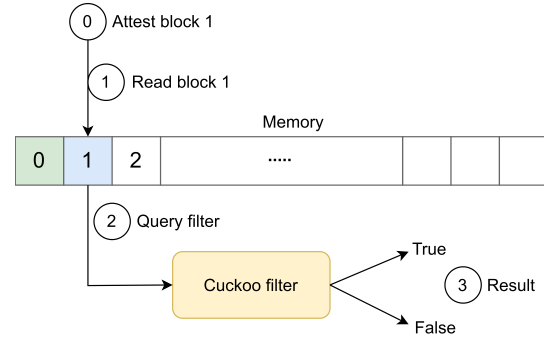

Fig. 2. Cuckoo filter to attest a memory block.

B. Attestation

To describe the attestation protocol, we consider an IoT device that can be interrupted to perform scheduled workloads with priority tasks. To achieve this, the ITERATOR protocol is self-initiated by the Prv and it can be interrupted.

1) Protocol Overview: ITERATOR is an interruptible RA protocol that splits the Prv's memory into multiple *blocks* (Fig. 1) and the attestation into different independent *rounds*. This division allows the protocol to be interrupted between rounds, lowering the impact of the attestation on the device's normal operations and availability. In this way, instead of measuring the entire memory, ITERATOR attests, in each round, a certain number of memory blocks. By employing a self-attestation scheme, ITERATOR enables fine-grained control of attestation rounds when adequate resources are available.

In order to check the integrity of the memory blocks, ITERATOR relies on a Cuckoo filter, which is used to represent the set of the legitimate memory blocks of the Prv. The Cuckoo filter construction is described in more detail in Section VI-A. During the attestation, the filter is queried to check whether a certain block is contained in the filter (Fig. 2). If the query returns a positive result, the block is considered genuine; otherwise, it is compromised.

Overall, each attestation round in the ITERATOR protocol consists of two actions: ATTEST and CHECK.

1) *ATTEST:* Checks the integrity of the next memory block in the memory array based on the value of the attestation progress variable p, where p ranges from 0 to the total number of memory blocks, $p \in [0, n]$. Such a variable p is stored in the secure storage of the Prv.
2) *CHECK:* Selects and re-attests randomly a predefined number k of blocks that have already been attested (in the interval from 0 to the current attestation progress), $k \in [0, p]$.

In this way, the ATTEST action makes the attestation progress forward, while the CHECK action verifies that the

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

previously attested blocks have not been modified since their last attestation.

2) Protocol Details: The attestation protocol begins at a randomly selected time that cannot be predicted by an adversary. Specifically, the ith attestation starts when the following condition is met:

$$T_i \geq T_{i-1} + t_{att}$$

where T_{i-1} is the timestamp of the end of the last attestation, and t_{att} is the random time interval of attestation. Specifically, to ensure the integrity of attestation timestamps and prevent manipulation, a random time interval t_{att} is added to the current time T, resulting in the actual attestation time. This approach adheres to the constraint $T+t_{att} \leq T_{att}$, ensuring that attestation occurs within the prescribed interval.

During each scheduled attestation task, an attestation round is initiated. The Prv disables the interrupts at the beginning and then performs the ATTEST and CHECK action. The ATTEST action retrieves the pth block from the memory and verifies whether it is present in the filter. The CHECK action proceeds by randomly selecting a number k of memory blocks from the interval $[0, p]$. After reading and verifying each of these blocks from the program memory, the attestation progress variable p is incremented by one. The pseudo-code of the ITERATOR protocol is presented in Algorithm 1. The attestation process is considered complete when one of the two conditions is met.

1) *Condition (1):* All memory blocks have been successfully attested, if the following equation holds:

$$p == \frac{MemorySize}{BlockSize}. \quad (1)$$

In this case, the attestation is considered successful since no compromised memory blocks have been found.

2) *Condition (2):* If the filter returns a negative result for any of the memory blocks, the attestation fails and the Prv is considered compromised.

The attestation result is represented by a Boolean value: true when the attestation is completed successfully (i.e., first condition), and false (i.e., second condition) otherwise. The result has the following format:

$$< T_i, res, MAC\,(T_i, res, key) >$$

where T_i is the timestamp of the start of the attestation, res is the Boolean attestation result and key is the secret key. The attestation result is signed with a message authentication code (MAC) created with the secret key. When the attestation ends, the attestation result is created and stored in the Prv's persistent storage. The attestation i is concluded.

C. Verification

At specific intervals, denoted by T_{coll}, the Vrf requests the Prv to send the collected attestation results. The Prv responds by sending all the stored attestation results. On receiving the collected results, the Vrf uses the shared key to verify the MAC signature and prove the integrity of the results. If there is a match between the calculated and the received MAC, then the result has not been tampered with and hence can be trusted by the Vrf. Finally, the Vrf updates the history of the results of the Prv.

Algorithm 1 ITERATOR Attestation Pseudo-Code

1: $k \leftarrow number\ random\ checks$
2: $T \leftarrow current\ time$
3: $p \leftarrow 0$ ▷ progress
4: $key \leftarrow secret\ key$
5: $n \leftarrow memory_size \div block_size$ ▷total number of blocks
6: $t_{att} \leftarrow random_time_interval()$
7: $T_{next} \leftarrow T + t_{att}$
8: **if** $T < T_{next}$ **then**
9:     **return**
10: **end if**
11: **if** $p == n$ **then**
12:     $res \leftarrow True$
13:     $mac \leftarrow MAC(T, res, key)$
14:     $res_{att} \leftarrow < T, res, mac >$
15:     $store_res(res_{att})$
16:     $p \leftarrow 0$
17:     $T_{next} \leftarrow T + t_{att}$
18:     **return**
19: **end if**
20: $block \leftarrow memory[p]$ ▷ATTEST action
21: $res \leftarrow filter_contains(p, block)$
22: **if** $res \neq True$ **then**
23:     $mac \leftarrow MAC(T, res, key)$
24:     $res_{att} \leftarrow < T, res, mac >$
25:     $store_res(res_{att})$
26:     $T_{next} \leftarrow T + t_{att}$
27:     $p \leftarrow 0$
28:     **return**
29: **end if**
30: $j \leftarrow 0$ ▷CHECK action
31: **while** $j < k$ **do**
32:     $rand_idx \leftarrow random(0, p)$
33:     $block \leftarrow memory[rand_idx]$
34:     $res \leftarrow filter_contains(rand_idx, block)$
35:     **if** $res \neq True$ **then**
36:         $mac \leftarrow MAC(T, res, key)$
37:         $res_{att} \leftarrow < T, res, mac >$
38:         $store_res(res_{att})$
39:         $T_{next} \leftarrow T + t_{att}$
40:         $p \leftarrow 0$
41:         **return**
42:     **end if**
43:     $j \leftarrow j + 1$
44: **end while**
45: $p \leftarrow p + 1$

VII. ITERATOR EXTENSIONS: ON-DEMAND AND OFFLOADING APPROACH

In ITERATOR's design, the Prv initiates the self-attestation process and the Vrf plays a passive role in collecting the reported results. However, in situations where the Vrf needs to perform a critical operation or a software update, it may be necessary to verify the trustworthiness of the Prv before proceeding. In such cases, an on-demand attestation approach would be necessary.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

A. On-Demand Approach

In Section VI, we introduced the self-attestation approach, which can be easily extended to provide on-demand attestation. The on-demand extension follows the same process as the self-measurement protocol, using the local Cuckoo filter in the Prv's memory. However, instead of starting at specific time intervals, the attestation is triggered by the Vrf, which sends a challenge (in the form of *Nonce*) to the Prv. This nonce is treated as the timestamp for the beginning of the attestation.

B. Offloading Approach

Offloading is a technique used in the RA context to securely send memory content to a third-party device, which performs integrity computation on behalf of the Prv and sends the attestation result to the Vrf [21]. In ITERATOR's context, offloading can be a promising approach when the Prv does not have enough secure memory to store the Cuckoo filter or when it lacks sufficient processing power to check whether the filter contains a particular element. This "contain" operation can be executed by the third party by offloading the Prv's memory blocks. This can be achieved by deploying the prebuilt Cuckoo filter to the trusted third party and modifying the attestation protocol deployed in the Prv. In each round, instead of querying the filter stored in the local memory, the Prv offloads the memory blocks it wants to attest by sending a message over the communication channel to the trusted third entity. Therefore, the message has the following format:

$$< b, i, T, MAC\,(b, i, T, key) >$$

where b is the memory block, i is the index of the memory block, T is the timestamp at the beginning of attestation, and key is the shared secret key.

The offloading approach for attestation of memory blocks involves the Prv querying "remotely" the filter stored in a trusted party via network requests. No computation is performed by the Prv for attestation. The secure offloading of memory blocks is ensured by the MAC attached to the message. However, this approach has a performance limitation due to the number of messages. Depending on the block size, this extension may not be convenient to implement when the communication overhead is significantly higher than performing the attestation on the Prv.

VIII. Protocol Simulation

We simulate the ITERATOR protocol in C# using the.NET 6 framework. The simulator uses the Cuckoo filter implementation provided by an open-source library [22]. We implement Prv's memory as an array of bytes. The Cuckoo filter is initialized at the beginning of the simulation by relying on the state of the Prv's memory blocks. The OP or Vrf iterates over the legitimate memory blocks and inserts each block into the Cuckoo filter with the corresponding block index. In this setup, we consider an adversary that can access the Prv's memory content. For the simulation, the malware content is generated as random bytes. Overall, the simulation comprises the following implementations of attacks.

1) Memory Injection Attack: The adversary selects a random block index and generates a memory block filled with random bytes representing the malware content. Then, it overwrites the selected block with the generated block in the Prv's memory.

2) Block Swap Attack: It is implemented as the injection of two memory blocks whose content is swapped. The adversary randomly selects two memory blocks and swaps their content.

3) Transient Adversary: The adversary injects a memory block into a random position in the Prv's memory and stores the block index along with its original content. Then, the adversary chooses a time in the future when the malware should be removed from the system. Later on, when the malware intends to leave the system, it restores the original content in the device memory and calculates a future time for the malware to re-infect the device. The time is determined by adding a random value to the current removal time. Once the malware returns, it randomly injects itself into another memory block. The malware may leave and return multiple times during the attestation.

4) Migratory Adversary: The attack starts by selecting a random memory block and storing the original content. Then, the malware injects its content into the selected block in the device's memory. After each attestation unit, the malware relocates itself to another block.

Simulation Details: The simulation relies on parameters such as: 1) the block size in bytes; 2) the number of blocks that are tested by the CHECK operation at each round; 3) the maximum time in which the adversary performs the attack is chosen randomly at runtime; and 4) the maximum time interval between two attestations. As the attestation start time is chosen randomly (to prevent the adversary from predicting when the next attestation starts), this value specifies the upper bound limit for choosing the next attestation time. The simulation is implemented as a loop that terminates when all the memory blocks have been tested, or one is detected as compromised. At each iteration, all the entities perform one action.

The simulation results show how often the protocol detected the malware presence. In the case of offloading and on-demand protocols, the simulation ends when the attestation is complete, regardless of whether malware is detected or not.

A. ITERATOR's Effectiveness Based on Simulations

In each run, the adversary performs an attack at a random time. The simulation's objective is to verify how many times each type of attack is detected, as well as how many times the protocol fails to do so. Specifically, there are two potential factors that might impact the effectiveness of the ITERATOR protocol on detecting the malware presence: 1) the k number of random blocks verified by the CHECK operation at each attestation round and 2) the block size of Prv's memory.

1) Number of Random Blocks in the Check Operation: In order to evaluate the protocol's effectiveness against all the implemented attacks (as described in Section VIII), namely, Memory injection, Block swap, Transient and Migratory, the simulator specifies a random attack option, which involves the attacker randomly selecting which attack to perform at each run. The simulation result breaks down the detection performance for each attack type.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

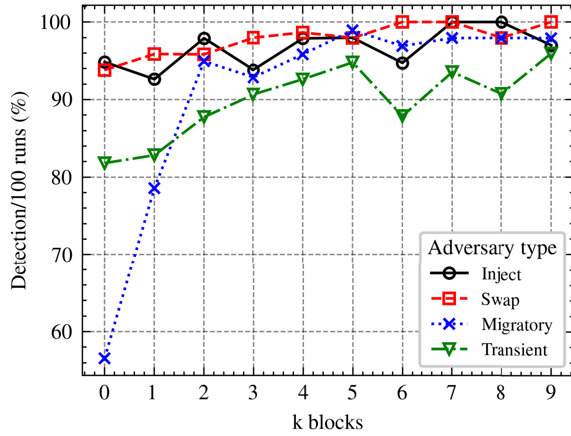

Fig. 3. On-demand detection performance. Each value is the average detection performance over 100 runs. k represents the number of blocks tested by the CHECK action.

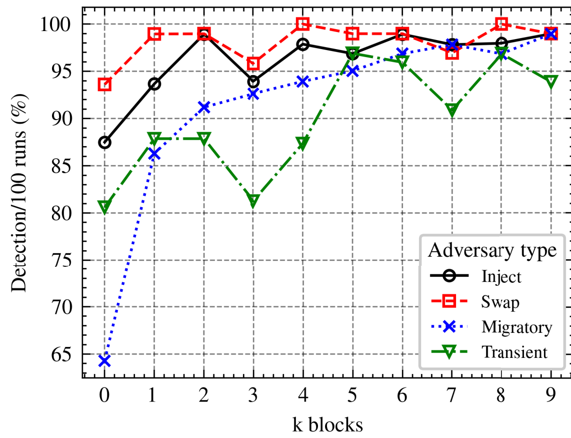

Fig. 4. Offloading detection performance. Each value is the average detection performance over 100 runs. k represents the number of blocks tested by the CHECK action.

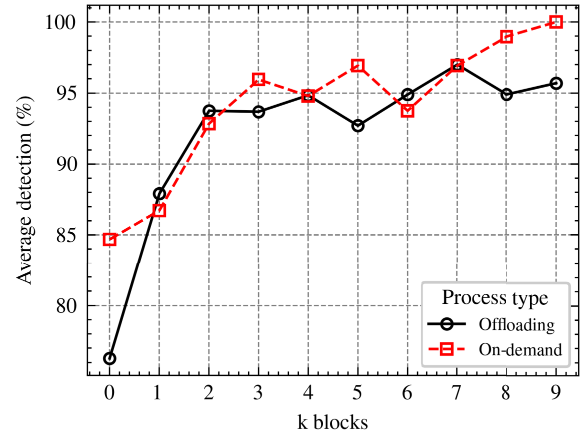

Fig. 5. On-demand and offloading average detection and block size.

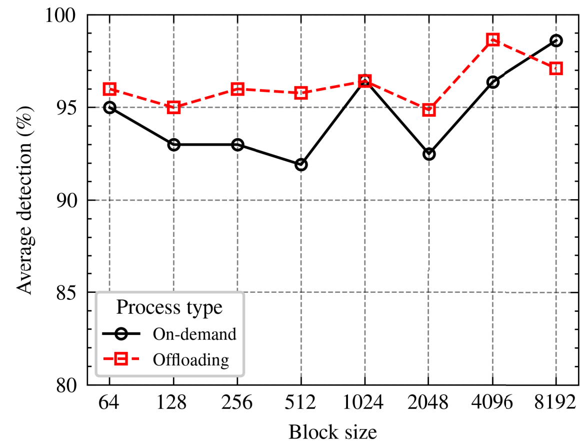

Fig. 6. On-demand and Offloading average detection and block size.

Figs. 3 and 4 show how the overall malware detection performance (for both on-demand and offloading approaches) changes concerning different values of k when considering a random choice of attack. Please note that the self-attestation and on-demand approaches of ITERATOR are the same, only with the difference that the device triggers self-attestation. As shown in the figures, the detection improves significantly when the number of blocks tested by the CHECK operation at each round increases. This is especially evident in the migratory and transient attacks. As they move in the memory during the attestation to avoid being detected, they are the most affected by the increased number of block checks. On the other hand, the detection of the block injection and the block swap attacks is not significantly affected by the increased number of checks. This is due to the fact that these types of attacks are static and do not perform any memory modification after the initial infection. The fluctuations in the detection accuracy are due to the fact that the attacks and the attestation happen at random times.

This overall performance of the protocol over multiple attack types is shown in Fig. 5. The figure presents how the overall malware detection performance changes in relation to different values of k when considering a random choice of attack. The value is calculated as the average of the detection performance for each attack type over 100 runs. As depicted in the figure, the detection improves significantly when the number of blocks tested by the CHECK operation at each round increases.

2) Block Size of the Prover's Memory: Besides the number of random blocks, another interesting characteristic to evaluate is how the block size influences the malware detection performance of the protocol. Intuitively, larger blocks reduce options for malware to relocate and improve detection. Smaller block sizes increase the number of blocks, giving more opportunities for malware to relocate and decrease detection.

However, the experiments demonstrate a different result. As shown in Fig. 6, the average detection performance is almost constant and does not change when the block size increases or decreases. Such results can be explained by the fact that when the total number of blocks increases, the total number of attestation rounds increases too, and in each round, the CHECK action performs multiple checks on the previously attested blocks. Therefore, as the total number of blocks increases, so does the number of checks performed on the

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

memory blocks. The same principle holds true when the total number of blocks decreases.

Assuming that the memory is divided into n blocks and the malware occupies one memory block, in each round, the CHECK action attests a number k of blocks with a probability of finding the compromised one of k/n. In addition, since the ATTEST action checks one block, the overall probability for malware detection is $(1+k)/n$. Therefore, the probability for the malware to evade detection in one round is $1-(1+k)/n$. Since the attestation requires n rounds to complete, the probability for the malware to evade detection by the ITERATOR protocol is

$$P = \left(1 - \frac{1+k}{n}\right)^n \approx e^{-(1+k)}. \quad (2)$$

This is similar to the probabilistic proof given in SMARM [6] for the migratory malware (referred to as *roving malware* in the same work) to escape detection, when $k = 0$ (since SMARM attests only one block per round without checking additional blocks)

$$P = \left(1 - \frac{1}{n}\right)^n \approx e^{-1} \approx 0.37. \quad (3)$$

ITERATOR reduces this probability exponentially with the number of additional checks k, as given in (2). As k increases, the evasion probability drops from 37% (for $k = 0$) to under 1% for sufficiently large k.

This mathematical proof gives comparable results to the ones obtained in the experiments in the case of the migratory malware. As shown in Figs. 3 and 4, the detection performance of ITERATOR improves and converges toward the theoretical values given in (2) as k increases. Note that Figs. 3 and 4 illustrate detection performance, whereas (2) represents the malware evasion probability. Therefore, the two values are complementary probabilities.

In general, the accuracy of malware detection increases as the number of blocks tested per round increases. This is true especially for migratory and transient malware attacks, since the attacker tries to avoid being detected during the attestation by moving in the memory of the device. However, the increase in the number of checks does not seem to greatly affect the detection of block injection and block swap attacks. This is due to the fact that these types of attacks are static and do not perform any memory modification after the initial infection. Therefore, if malware is injected into a block that hasn't been attested yet, it will certainly be detected once the affected block is attested. Otherwise, if the attack involves a block that has already been attested, then it will be detected only if that specific block is tested again by the CHECK action. This explains why the detection performance increases slightly as the value of n increases.

Moreover, since the probability of escaping detection depends only on the total number of memory blocks and the number of checks performed, it supports the experimental finding that the attestation performance is not affected by the block size.

TABLE III
HASH FUNCTION COMPARISON: FNV AND MURMURHASH 3

	FNV	MurmurHash3
Avg check CPU cycles	35741.71	16348.4
Avg check time (us)	224,3	102.34

TABLE IV
PERFORMANCE OF CHECKING A BLOCK AND MEMORY OCCUPATION OF THE FILTER FOR DIFFERENT BLOCK SIZE

	256 bytes	512 bytes	1024 bytes	2048 bytes	4096 bytes
Avg check CPU cycles	8860.35	16348.4	31713.24	62459.21	123918.26
Avg check time (us)	53.62	102.34	198.21	390.81	774.71
Blocks	4096	2048	1024	512	256
Cuckoo filter bytes	16384	8192	4096	2048	1024

IX. HARDWARE PoC ON ESP32

We implemented the hardware PoC of ITERATOR in an ESP32, Xtensa[1] dual-core 32-bit LX6 CPU with FreeRTOS. The device's memory consisted of 4-MB integrated flash memory, 448 kb of ROM, and 530 kb of SRAM. This hardware platform supported Bluetooth and WiFi technologies and was equipped with a cryptographic hardware accelerator for AES, SHA-2, RSA, ECC, and RNG.

To implement the Cuckoo filter, we adopted the open-source project [23] as one of the building blocks for implementing ITERATOR on the ESP32 platform. The device initializes the filter during boot by reading the program partition memory blocks. The application partition data is read using the *esp_partition_read* function, providing the pointer to the partition, the starting offset, and the number of bytes it should read. The progress of the attestation is stored in a variable, which indicates how many blocks of memory have been attested.

A. Protocol's Efficiency Based on Esp32

The performance of the protocol implementation has been tested on the ESP32 hardware platform. In particular, we measure the performance in terms of CPU cycles and run time of querying the filter for a memory block and completing one attestation round.

The most expensive operation in a Cuckoo filter is the hash calculation used to insert or query the elements. For this reason, we first evaluated two different noncryptographic hash functions: FNV [24] and MurmurHash 3 [25]. The evaluation results, as shown in Table III, demonstrate that MurmurHash 3 is substantially quicker than FNV. Therefore, we rely on MurmurHash 3 for the rest of our experiments.

Table IV shows the performance of checking whether a block is contained in the Cuckoo filter and the memory occupation change with the block size. The experimental results show a linear correlation between the time required to measure one block and its size. The smaller the block is, the quicker it is processed by the Cuckoo filter. Moreover, the total number of blocks increases when the block size decreases. In this case, the filter has to store more items, increasing its size and memory footprint. When the block size grows, the total

[1] Registered trademark.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

TABLE V
ROUND TIME AS THE NUMBER OF RANDOM BLOCK CHECKS CHANGES

	0	1	2	3	4	5	6	7	8	9
CPU cycles	58062	114559	171881	228854	285996	343166	400407	457567	514728	571799
time (us)	362	716	1074	1430	1787	2144	2502	2859	3216	3573

number of blocks is lower. Therefore, with big memory blocks, the overhead of performing a membership query on the filter is higher, but the memory footprint of the filter is lower.

Another interesting aspect to analyze is the time required to complete one attestation round in relation to different values for the number k of random block checks. If the number of randomly checked blocks increases, the time and the CPU cycles (required to complete one round) increase linearly. In addition, Table V shows the runtime of the Cuckoo filter approach to measure memory blocks in the context of RA. Even with many additional checks (e.g., $k = 9$), the total round time is around 3.5 ms. Typically, the runtime of state-of-the-art RA schemes varies from milliseconds to seconds [20]; thus, ITERATOR's runtime is comparable to state-of-the-art RA schemes.

1) Discussion: Overall, the evaluation shows the *flexibility* of the ITERATOR. By tuning its parameters, the protocol can be adapted to different situations. For example, a device that performs multiple time-critical operations might prefer a smaller block size or fewer random block checks to benefit from the lower attestation round time. Alternatively, a device that requires maximum detection performance at the cost of a higher attestation round time might want to increase the number of blocks checked by the CHECK operation.

B. Comparison With State-of-the-Art

The ITERATOR protocol can be compared to state-of-the-art protocols like SMARM [6] and HAtt [7], which also aim to address the problem of the interruptibility of the attestation procedure and the detection of a mobile adversary. These three schemes assume that the device memory is divided into blocks and that the adversary has to relocate at least once, while it can only infer the volume of memory that has not been measured yet.

Overall, these schemes differ from each other in the adversary model, the approach in enabling interruptible attestation, and the performance, as highlighted in Table VI. While having the same goal, ITERATOR has some fundamental differences with respect to SMARM and HAtt.

1) ITERATOR includes the transient malware type in its adversary model, while SMARM and HAtt are limited to the migratory one.
2) ITERATOR uses the Cuckoo filter and the CHECK action to ensure that the previously measured memory is still in its expected state. This is different from SMARM, which relies on shuffled measurements in a secret and fresh random order of the memory blocks. This prevents the malware from relocating to an already measured memory block. Moreover, SMARM uses the hash computation of a memory block as proof of its integrity, and HAtt provides a partial attestation, as it uses a PUF to randomly select bits from each word of the memory. ITERATOR uses a Cuckoo filter to ensure that each memory block is in its genuine state.
3) SMARM and HAtt show similar detection performance, with a probability for the malware to escape the detection of 37%. ITERATOR can achieve better detection performance, with the escaping probability ranging from 37% to less than 1%, depending on the number of blocks measured by the CHECK action.
4) The comparison on the timing and the resource usage of the three protocols is limited due to the different choices of hardware platforms and the availability of the protocol's source code. Still, the experiments show that ITERATOR has higher effectiveness on average compared to the state-of-the-art and a comparable efficiency, with only 34.3 ns to attest a singular memory block.

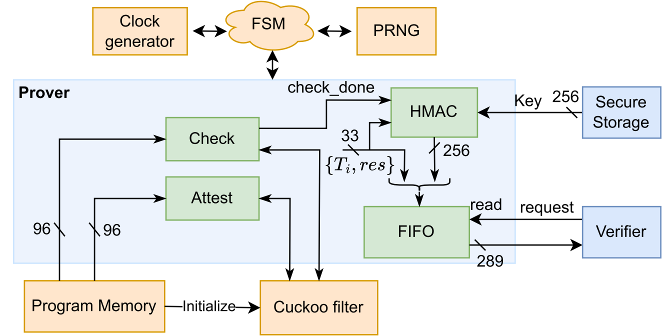

Fig. 7. Hardware architecture of the PoC on FPGA, detailing the core components for attestation and HMAC generation.

Please note that in the field of RA, reproducing experimental results from other articles is challenging due to reliance on customized hardware implementation and often unavailable source code.

X. HARDWARE PoC ON FPGA

The PoC on FPGA targets a lightweight implementation of the ITERATOR, optimized for low resource utilization and latency. To ensure proper message integrity and authenticity, the design employs a hash-based MAC (HMAC). This HMAC is built around a SHA-256 core, adapted from the work in [26]. The implementation targets the Xilinx Zynq Ultrascale+ ZCU104 MPSoC evaluation platform, featuring an XCZU7EV-FFVC1156-2-E FPGA, and all synthesis and analysis were carried out using Vivado 2022.2 design suite.

Fig. 7 shows the hardware architecture of the PoC on the FPGA. The processes within the architecture are controlled by a finite state machine (FSM). The clock generator, which is a simple 32-bit up counter controlled by the FSM, generates the real-time clock values. A pseudorandom number generator (PRNG) generates random time intervals for calculating t_{att}.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
COMPARISON OF ITERATOR WITH THE STATE-OF-THE-ART PROTOCOLS

	SMARM	HAtt	ITERATOR
Mobile adversary model	Only migratory	Limited Migratory	Migratory & Transient
Attestation scheme	Challenge-response	Challenge-response	Self-attestation
Attestation method	Hash	Bit string	Cuckoo filter
Strategy	Shuffle-measurements	PUF to select random bits	CHECK action
Attestation unit	Memory block	Memory block	Round
Evasion probability	37%	37%	37% to $< 1\%$

TABLE VII
FPGA RESOURCE UTILIZATION OF CUCKOO FILTER-BASED POC

Component	LUTs	Registers	Slices	BRAM
PoC	**2554**	**1524**	**482**	**29**
Cuckoo filter	867	226	174	4
Prover & FSM	1638	1249	244	3.5
HMAC	1559	1080	276	0
Utlization (%)	1.11%	0.33%	1.67%	9.29%

TABLE VIII
TIMING REQUIREMENTS OF CUCKOO FILTER-BASED POC

Component	Attest time per block		Attest time for one round		F_{clock} (MHz)
	[CC]	[ns]	[CC]	[ns]	
PoC	7	20.4	1470	4292.4	342
Cuckoo filter	5	14.6	-	-	-

CC - Clock cycles, F_{clock} - Maximum operating frequency

The Cuckoo filter in the PoC employs the Xoodoo-NC as the hash function [27], a noncryptographic version of the Xoodoo permutation. Xoodoo-NC takes a 96-bit input and produces a hash output in a single clock cycle thanks to its low logic depth. The output of Xoodoo-NC can be any multiple of 96 bits. The program memory is implemented as a single-port ROM, partitioned into 96-bit blocks to match the input width of the hash function.

Upon completion of the ATTEST and CHECK phases, a cryptographic proof is generated using the keyed-hash construction. A MAC is calculated over a message composed of the final timestamp (T_i) and the Boolean result (res). To provide authenticity, this message is concatenated with a secret key provisioned uniquely to the device. The HMAC unit processes this combined input, producing a 256-bit MAC. The complete $\langle T, res, mac\rangle$ proof is then written to a FIFO for asynchronous collection by the Verifier. The Vrf can collect the results from the Prv by sending a collection request, which acts as the *read enable* signal to the FIFO.

A. PoC Analysis: Resource Utilization

Table VII summarizes the FPGA resource utilization of the PoC. For the analysis, the design provisions 768 kb of program memory, corresponding to the utilization of 21.5 Block RAMs (36-kb BRAM). It requires only four BRAMs for the Cuckoo filter to store the content of the program memory. Notably, the SHA-256 module consumes the lion's share of the resources used by the PoC. The higher resource consumption of SHA-256 motivates the need for a lighter MAC engine, such as the Xoodoo cryptographic permutation [28], to minimize resource consumption and enhance performance. While the program memory allocated to the PoC is only 768 kb for the analysis, it is worth noting that increasing the program memory allocation has only a negligible effect on overall FPGA logic usage.

B. PoC Analysis: Timing

Table VIII summarizes the timing requirements of the implementation with regard to the attestation time. In this implementation, each attestation round performs the attestation of 210 blocks, equivalent to 2520 bytes, followed by the CHECK operation. The time required for the ATTEST/CHECK operation of a single block is seven clock cycles, which includes reading the memory block and the Cuckoo filter lookup. Attesting a single block requires only 34.3 ns, while completing a full round takes 7203 ns. In addition, the SHA-256-based HMAC unit requires 68 clock cycles to compute the MAC value. Since the MAC computation proceeds independently, substituting the SHA-256-based MAC with an alternative unit will not impact the attestation time, as there is sufficient slack to compute the MAC until the end of the subsequent attestation.

C. Enhancing the Performance on FPGA Using Bloom Filter

A Bloom filter is an attractive data structure for hardware implementation due to its simplicity and resource efficiency, compared to the Cuckoo filter. However, a standard Bloom filter's query latency scales linearly with the number of hash functions l, as each hash computation and memory access must occur sequentially. Hence, a single block attestation requires l hash computation cycles and l memory access cycles. These higher latency requirements of the Bloom filter prompt us to prefer Cuckoo filters, as they maintain a consistent query latency regardless of filter size. However, leveraging FPGA parallelism can optimize Bloom filter performance on hardware, such as the parallel Bloom filter (PBF) presented in [27]. PBF employs a single hash function, Xoodoo-NC, to generate required hash values, and the Bloom filter memory is partitioned into l sub-blocks. This enables the generation of required hash values in a single clock cycle, and all hash-indexed memory locations can be accessed in parallel. Consequently, the overall query latency is reduced to just two clock cycles, irrespective of the number of hash functions in use, making it a highly effective solution for this work.

We implemented the PoC on an FPGA by replacing the Cuckoo Filter with the Bloom filter while allocating the same amount of memory to the Bloom Filter. Tables IX and X provide the resource utilization and timing requirements, respectively. For Bloom filter-based PoC, the resource utilization in terms of lookup tables (LUTs) is reduced by $\sim$17%, and the attestation time for one round in terms of

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

TABLE IX
RESOURCE UTILIZATION OF BLOOM FILTER-BASED POC

Component	LUTs	Registers	Slices	BRAM
PoC	**2114**	**1441**	**437**	**29**
Bloom filter	351	142	72	4
Prover & FSM	1671	1250	282	3.5
HMAC	1465	1081	251	0
Utlization (%)	0.92%	0.31%	1.52%	9.29%

TABLE X
TIMING REQUIREMENTS OF BLOOM FILTER-BASED POC

Component	Attest time per block		Attest time for one round		F_{clock} (MHz)
	[CC]	[ns]	[CC]	[ns]	
PoC	4	15.6	840	3292.8	255
Bloom filter	2	7.8	-	-	-

CC - Clock cycles, F_{clock} - Maximum operating frequency

the number of clock cycles is reduced by ~43%, compared to the Cuckoo filter-based PoC. Nevertheless, the maximum operating frequency of the Bloom filter-based PoC is lower than its Cuckoo filter-based counterpart. This reduction is attributed to the increased routing delay due to multiple parallel memory blocks, resulting in a longer critical path. In summary, substituting the Bloom filter with the Cuckoo filter can improve the performance of our proposed scheme on an FPGA. Nonetheless, in other implementation platforms where parallelization is not feasible, the Cuckoo filter stands out as the optimal choice.

XI. SECURITY ANALYSIS

In this section, we present an informal discussion on how ITERATOR satisfies the security requirements in Section V-B.

A. Freshness of the Result

In order to detect a replay attack, the freshness of the self-initiated approach in ITERATOR is guaranteed by the presence of the timestamp T at the beginning of attestation. When an Adv_{replay} sends a previously recorded attestation result res_{old} which corresponds to a legitimate measurement of one or more memory blocks, the Vrf can verify the freshness of the result based on the expected incremental attestation time occurring within the time interval T_{att}. Moreover, in the on-demand approach, the freshness of the result is guaranteed by a randomized *Nonce* sent by the Vrf. Assuming the probability of transmitting a randomized *Nonce*, where $Nonce = Nonce_{old}$ is negligible, two distinct attestation results will not match. Consequently, Vrf can identify replay attacks.

B. Tampering Detection

To detect unauthorized memory modifications, the protocol uses a Cuckoo filter to attest each memory block. During the offline phase, the Cuckoo filter is securely constructed with the benign memory blocks by concatenating the block index with the block data. Such construction allows the Cuckoo filter to verify if a block belongs to it and prevent block swap attacks. The ATTEST action ensures that each block is attested at least once. In ITERATOR, the integrity of the Cuckoo filter is guaranteed by storing it in a secure, persistent memory and only accessible by the attestation code.

C. Mobile Adversary Detection

ITERATOR aims to detect a mobile adversary that exhibits the behavior of a: 1) migratory malware and 2) transient malware. Regarding the migratory malware, ITERATOR assumes that it performs its best strategy as presented in [6]: the malware relocates at least once at every attestation round. The detection of this type of malware is done with the ATTEST and the CHECK actions as follows.

1) If the malware relocates to the next memory block to attest, the ATTEST action will find it at the next round.
2) If the malware relocates to a previously attested memory block, the CHECK action will be able to detect it with probability k/p, where k is the number of checked random blocks and p is the attestation progress. To detect transient malware that self-deletes before attestation, ITERATOR randomly changes the attestation time. Moreover, self-attestation prevents malware from intercepting Vrf's attestation challenge.

D. Integrity of the Results

In ITERATOR, the attestation result *res* is stored in the secure storage of the Prv, which protects it from modification by a software adversary Adv_{soft}. In addition, *res* is also associated with a MAC to ensure integrity and authenticity. As only ITERATOR can access the shared secret key *key*, the attestation result *res*, is authenticated and cannot be tampered with by an Adv_{soft}.

E. Communication Integrity and Authenticity

Any changes of the communication data by a communication adversary Adv_{com} that aims to manipulate the attestation details, e.g., attestation time T or the attestation result *res*, will be detected. In ITERATOR, the communication data between the Prv and the Vrf are authenticated with a MAC shared secret key *key*. Given a secure MAC function, it will be infeasible for Adv_{com} to forge the data without knowing the *key*.

XII. LIMITATIONS

A. Last Memory Blocks Are Checked Less Than the First Ones

During an attestation round, the CHECK action randomly selects a predefined number of previously attested memory blocks to be re-checked. A malware aware of this pattern can maintain its presence for as long as possible by placing itself in the last memory block—the least checked. To address this limitation, the CHECK action can be extended by allowing it to choose the random blocks from the whole memory instead of limiting the choice to only the already attested ones.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.

B. Increased Latency

Considering the attestation's starting and ending time, the ITERATOR protocol takes longer than traditional atomic attestation approaches. This is because the attestation is interleaved with the device's normal operations instead of being executed in one go. This increased flexibility is a trade-off for the higher latency. In addition, since the attestation is done via self-measurement, the device can execute it when there is enough computation time available. This allows the execution of multiple rounds in a row, decreasing the overall attestation latency.

XIII. Conclusion and Future Works

In this article, we design, implement, and evaluate ITERATOR as a novel attestation protocol that can be securely interrupted while being able to detect mobile adversaries. In particular, the ITERATOR relies on a Cuckoo filter to check whether the entire content of a memory block matches an expected legitimate one. To ensure that a mobile adversary does not evade detection, each protocol round includes a check on the previously attested blocks. We evaluated the effectiveness and efficiency of ITERATOR through a software simulation and two hardware PoC implementations. Our future work includes implementing ITERATOR over a large IoT swarm that operates under intermittent connectivity.

References

[1] D. Kushner, "The real story of Stuxnet," *IEEE Spectr.*, vol. 50, no. 3, pp. 48–53, Mar. 2013. [Online]. Available: https://spectrum.ieee.org/the-real-story-of-stuxnet

[2] M. Antonakakis et al., "Understanding the mirai botnet," in *Proc. 26th USENIX Secur. Symp.*, Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf

[3] (2023). *Is Your Smart TV Spying on You?*. Accessed: May 29, 2025. [Online]. Available: https://www.kaspersky.com/resource-center/threats/is-your-smart-tv-spying-on-you

[4] E. Dushku and N. Dragoni, "Remote Attestation in IoT Devices," in *Encyclopedia of Cryptography, Security and Privacy*, S. Jajodia, P. Samarati, and M. Yung, Eds., Berlin, Germany: Springer, 2022, pp. 1–4, doi: 10.1007/978-3-030-71522-9_1782.

[5] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *Proc. 19th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2012, pp. 1–15.

[6] X. Carpent, N. Rattanavipanon, and G. Tsudik, "Remote attestation of IoT devices via SMARM: Shuffled measurements against roving malware," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Apr. 2018, pp. 9–16, doi: 10.1109/HST.2018.8383885.

[7] M. N. Aman et al., "HAtt: Hybrid remote attestation for the Internet of Things with high availability," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 7220–7233, Aug. 2020, doi: 10.1109/JIOT.2020.2983655.

[8] M. M. Rabbani, E. Dushku, J. Vliegen, A. Braeken, N. Dragoni, and N. Mentens, "RESERVE: Remote attestation of intermittent IoT devices," in *Proc. 19th ACM Conf. Embedded Networked Sensor Syst. (SenSys)*, 2021, pp. 578–580, doi: 10.1145/3485730.3493364.

[9] B. Kuang, A. Fu, W. Susilo, S. Yu, and Y. Gao, "A survey of remote attestation in Internet of Things: Attacks, countermeasures, and prospects," *Comput. Secur.*, vol. 112, Jan. 2022, Art. no. 102498, doi: 10.1016/j.cose.2021.102498.

[10] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014, pp. 1–14, doi: 10.1145/2592798.2592824.

[11] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6, doi: 10.1145/2744769.2744922.

[12] X. Carpent, G. Tsudik, and N. Rattanavipanon, "ERASMUS: Efficient remote attestation via self-measurement for unattended settings," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1191–1194, doi: 10.23919/DATE.2018.8342195.

[13] X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Invited: Reconciling remote attestation and safety-critical operation on simple IoT devices," in *Proc. 55th ACM/ESDA/IEEE Design Autom. Conf. (DAC)*, Jun. 2018, pp. 1–6, doi: 10.1109/DAC.2018.8465928.

[14] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 1, pp. 131–155, 1st Quart., 2012, doi: 10.1109/SURV.2011.031611.00024.

[15] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than Bloom," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2014, pp. 75–88, doi: 10.1145/2674005.2674994.

[16] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004, doi: 10.1016/j.jalgor.2003.12.002.

[17] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with 0 (1) worst case access time," *J. ACM*, vol. 31, no. 3, pp. 538–544, Jun. 1984, doi: 10.1145/828.1884.

[18] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Proc. VLDB*, 1980, pp. 212–223. [Online]. Available: https://dl.acm.org/doi/10.5555/1286887.1286911

[19] T. Abera et al., "Invited: Things, trouble, trust: On building trust in IoT systems," in *Proc. 53nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6, doi: 10.1145/2897937.2905020.

[20] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise, "Collective remote attestation at the Internet of Things scale: State-of-the-art and future challenges," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 4, pp. 2447–2461, 4th Quart., 2020, doi: 10.1109/COMST.2020.3008879.

[21] J. H. Ostergaard, E. Dushku, and N. Dragoni, "ERAMO: Effective remote attestation through memory offloading," in *Proc. IEEE Int. Conf. Cyber Secur. Resilience (CSR)*, Sep. 2021, pp. 73–80, doi: 10.1109/CSR51186.2021.9527978.

[22] Microsoft. *Mixer-Cuckoo*. Accessed: Sep. 16, 2025. [Online]. Available: https://github.com/microsoft/mixer-cuckoo

[23] J. Harris. *Libcuckoofilter*. Accessed: Sep. 16, 2025. [Online]. Available: https://github.com/jonahharris/libcuckoofilter

[24] *FNV-hash*. Accessed: Sep. 16, 2025. [Online]. Available: https://github.com/catb0t/fnv-hash

[25] PeterScott.*Murmur3*. Accessed: Sep. 16, 2025. [Online]. Available: https://github.com/PeterScott/murmur3

[26] E. Dushku, M. M. Rabbani, J. Vliegen, A. Braeken, and N. Mentens, "PROVE: Provable remote attestation for public verifiability," *J. Inf. Secur. Appl.*, vol. 75, Jun. 2023, Art. no. 103448, doi: 10.1016/j.jisa.2023.103448.

[27] A. Sateesan, J. Vliegen, J. Daemen, and N. Mentens, "Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications," *Microprocessors Microsyst.*, vol. 93, Sep. 2022, Art. no. 104619, doi: 10.1016/j.micpro.2022.104619.

[28] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, "The design of Xoodoo and Xoofff," *IACR Trans. Symmetric Cryptol.*, vol. 2018, no. 4, pp. 1–38, Dec. 2018, doi: 10.46586/tosc.v2018.i4.1-38.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on May 06,2026 at 11:30:29 UTC from IEEE Xplore. Restrictions apply.