2019 IEEE European Symposium on Security and Privacy (EuroS&P)

A Practical Attestation Protocol for Autonomous Embedded Systems

Florian Kohnhäuser, Niklas Büscher, Stefan Katzenbeisser
Security Engineering, TU Darmstadt, {kohnhaeuser,buescher,katzenbeisser}@seceng.informatik.tu-darmstadt.de

***Abstract*—With the recent advent of the Internet of Things (IoT), embedded devices increasingly operate collaboratively in autonomous networks. A key technique to guard the secure and safe operation of connected embedded devices is remote attestation. It allows a third party, the verifier, to ensure the integrity of a remote device, the prover. Unfortunately, existing attestation protocols are impractical when applied in autonomous networks of embedded systems due to their limited scalability, performance, robustness, and security guarantees.**

In this work, we propose PASTA, a novel attestation protocol that is particularly suited for autonomous embedded systems. PASTA is the first that (i) enables many low-end prover devices to attest their integrity towards many potentially untrustworthy low-end verifier devices, (ii) is fully decentralized, thus, able to withstand network disruptions and arbitrary device outages, and (iii) is in addition to software attacks capable of detecting physical attacks in a much more robust way than any existing protocol. We implemented our protocol, conducted measurements, and simulated large networks. The results show that PASTA is practical on low-end embedded devices, scales to large networks with millions of devices, and improves robustness by multiple orders of magnitude compared with the best existing protocols.

***Index Terms*—remote attestation, collective attestation, embedded systems, mesh networks, autonomous systems**

I. Introduction

In recent years, networked embedded devices have become an integral part of our daily lives. In the course of this evolution, which is commonly referred to as the Internet of Things (IoT), devices are increasingly deployed in distributed and autonomous networks. In these networks, devices operate collaboratively with minimal or no supervision. Specific application scenarios include industrial control [10], smart cities [25], building automation [38], logistics [24], or environmental monitoring [18]. In many of these scenarios, security is a vital objective, as devices process privacy-sensitive data or perform safety-critical operations. Unfortunately, recent attacks and independent security evaluations have shown that embedded devices frequently lack security and can often be compromised and misused with little effort [13], [31]. Therefore, much research effort has been put into the detection of compromised devices. This goal is achieved through so-called remote attestation protocols, which enable a third party, the verifier, to ensure the integrity of a remote device, the prover. However, existing attestation protocols are unable to fulfill the requirements on scalability, performance, robustness, and security of autonomous embedded systems, as described in the following and summarized in Table I.

Scalability & Performance: Autonomous embedded systems may comprise thousands of collaborating low-end embedded devices. For instance, a network may contain many actuator devices that control a safety-critical process based on measurements received from many sensor devices. In this case, all actuators need to verify that all sensors from which they receive data are uncompromised. Hence, sensors must be provers and actuators must be verifiers (and potentially provers as well). Applying traditional single-device attestation protocols in this case results in a huge overhead, because they only enable a single verifier to ensure the integrity of a single prover. Collective attestation protocols address this issue to some extent, as they distribute the computational and communication burden across all provers in the network [2], [4], [8], [21], [22], [28], [29]. This allows a scalable and efficient attestation of the entire network. Yet, collective attestation protocols are typically centralized, meaning that only a single entity, which is usually the network operator, is capable of verifying the attestation result. In fact, only one collective attestation protocol considers multiple verifiers [2]. However, it poses a high computational burden on provers and even more on verifiers, which makes it unsuitable in cases where provers and especially verifiers perform time-critical operations or have limited computational power, e.g., as they are low-end embedded devices[1].

Robustness: Autonomous systems typically need to operate undisturbed without manual intervention for long times, during which the system may have to overcome network and device disruptions. Therefore, an attestation protocol for autonomous systems must be robust and sustain its security service in case of failures. However, the dependence of existing protocols on a centralized [4], [8], [21], [22], [28], [29] or external verifier [2] that initiates, controls, and verifies the attestation, constitutes a single point of failure for the attestation protocol, which impairs robustness.

Security: Devices in autonomous systems are often physically accessible, deployed in large areas, and left unsupervised for long times. These circumstances allow an adversary to physically approach and manipulate devices much easier than typical computer systems. Most attestation protocols only protect against software attacks. Hence, their security can be undermined by an adversary who is able to physically tamper with devices. Recently, two protocols [21], [28] propose to combine collective attestation with absence detection [12]

[1]For example, the authors report that a 48 MHz embedded prover device requires more than 2.2 seconds to generate its attestation report and a server (Amazon EC2 t2.micro instance) requires more than 1 second to verify an attestation report containing 1000 prover devices [2].

DOI 10.1109/EuroSP.2019.00028

to detect software and physical attacks. Absence detection builds on the assumption that devices provide physical tamper-resistance that requires an adversary to take a targeted device offline for a noticeable amount of time, e.g., to decapsulate the device [5], [42]. Therefore, all prover devices that are offline for a time longer than a certain threshold are regarded as physically compromised. Unfortunately, existing protocols are prone to network and device outages, whereupon healthy, but temporarily unreachable, provers are mistakenly regarded as physically compromised. Thus, the additionally limited robustness of protocols that detect physical attacks make them unsuitable for autonomous systems.

Contribution. In this work, we propose PASTA, a practical attestation protocol for autonomous networks of embedded systems. In PASTA, prover devices periodically collaborate to generate so-called tokens. Each token attests the integrity of all provers that participated in its generation. During token generation, provers mutually ensure their integrity and then make use of a Schnorr-based multisignature scheme to compute an aggregated signature, which is stored in the token and attests the provers' integrity. The aggregated signature is of constant size and can be efficiently generated, which ensures scalability and performance. Furthermore, it is publicly and efficiently verifiable, which enables even untrustworthy low-end embedded devices to rapidly verify the integrity of all provers. For instance, in our implementation, one million low-end embedded prover devices can attest their integrity by generating a token within 0.5 s. The generated token is as small as 68 bytes and can be verified by an untrustworthy low-end embedded device in less than 21 ms.

To increase the availability of the attestation result in case of network and device outages, tokens are distributed to all devices in the network, instead of being stored centralized. Since tokens are protected by their contained signature, any device can store and forward tokens, which facilitates their quick distribution. Moreover, PASTA is fully decentralized, because all provers equally ensure the freshness of tokens by periodically initiating a new token generation. Thus, in case of arbitrary network or device outages, all remaining operative and connected provers still sustain the token generation and attest their integrity towards all network devices.

In addition to software attacks, PASTA is also able to detect physical attacks. For this purpose, PASTA relies on the common assumption that physical attacks require an adversary to take a targeted device offline for at least the time period δ_a (e.g., 10 min) [11], [12], [21], [28]. Hence, any prover whose last participation in a token generation is more than or equal to δ_a time ago is considered to be physically compromised. In contrast to the most robust existing solution [28], PASTA calculates the absence time of each prover individually. This doubles the timespan in which provers are required to participate in the protocol. Thus, compared with [28], PASTA can increase robustness against disruptions while providing the same security level or halve δ_a to provide stronger security guarantees with the same level of robustness.

	Scalability		*Performance*		*Robustness*		*Security*
	Scalable Prover Attestation	**Publicly Verifiable Report**	**Efficient Report Generation**	**Efficient Report Verification**	**Decentralized Attestation**	**Robust to Net. Disruptions**	**Detection of Physical Attacks**
SEDA [4]	●	○	●	●	○	◐	○
SANA [2]	●	●	◐	○	○	◐	○
DARPA [21]	●	○	●	●	○	○	●
SeED [22]	●	○	●	●	○	◐	○
SCAPI [28]	●	○	●	●	○	○	●
LISA [8]	●	○	●	●	○	◐	○
SALAD [29]	◐	○	●	●	○	●	○
This work	●	●	●	●	●	●	●

TABLE I: Overview of collective attestation protocols and features important for the attestation of autonomous embedded systems.

In general, physical attacks require not only much time, but also expensive equipment and laborious handwork of highly skilled personnel. Therefore, we argue that a physical adversary is only able to successfully tamper with a limited number of devices simultaneously. Based on this assumption and the property of tokens to be publicly verifiable, PASTA is able to reunite groups of prover devices that have been separated for longer than δ_a time, meaning that any prover of one group has not generated a token with any prover of the other group within δ_a time and thus violated the original absence assumption. To reunite groups, provers of different groups exchange their generated tokens that (i) testify the permanent presence of all provers of the particular group during the separation, and (ii) could not have been forged by an adversary under the aforementioned assumption due the large size of the group. This way, PASTA is the first protocol that can detect physical attacks and recover from long-lasting network splits that segmented a network in separated groups.

In short, we propose PASTA, the first attestation protocol that has the following properties:

- It provides a scalable and efficient attestation with many provers and many verifiers. In fact, thousands of low-end embedded prover devices are able to attest their integrity in a small token that can be verified by any network device, including low-end embedded and untrustworthy devices.
- It provides a high level of robustness against device and network disruptions. This is because (i) it is fully decentralized, thus, able to overcome arbitrary device outages, and (ii) it distributes the attestation result to all devices, whereby any device can be a data mule and increase availability.
- It can detect physical attacks while providing better security guarantees and higher robustness against disruptions than any existing protocol. This is because (i) it is able to recover from long-lasting network disruptions where the network was segmented in multiple separated parts, and (ii) it doubles the timespan in which provers are required to participate in the protocol.

Finally, we discuss the security of our protocol and show its scalability in large networks and robustness in dynamic as well as disruptive network topologies.

II. Related Work

Single-Device Attestation. Remote attestation is a technique with which a trusted party, the *verifier*, is able to check the integrity of a remote device, the *prover*. During attestation, the verifier typically challenges the prover and receives an attestation result, which indicates whether the prover is in a trustworthy system state (or not). To ensure a secure attestation, provers rely on a root of trust that is either *software-based* [32], *hardware-based* [30], or *hybrid* [27], i.e., based on a software/hardware co-design. Since software-based schemes provide questionable security guarantees [9] and hardware-based schemes are often too complex or expensive, hybrid schemes are regarded most suitable for embedded devices [16].

Collective Attestation. For the purpose of verifying interconnected groups of embedded devices, *collective attestation* protocols have been proposed. These protocols are able to efficiently verify the entire network by distributing the attestation burden across all devices in the network. In SEDA [4], a spanning tree overlay is initially arranged in the network, with the verifier being the root of the tree. During protocol execution, devices attest their neighboring devices, i.e., devices within direct communication range, and propagate the attestation result hop-by-hop along the spanning tree to the verifier. In each hop, the attestation result is aggregated, which significantly increases *efficiency* and *scalability*. Subsequently proposed collective protocols build on the initial approach of SEDA. LISAα and LISAs [8] provide a quality metric for collective attestation protocols and a practical performance evaluation. SeED [22] introduces a non-interactive attestation approach that mitigates Denial of Service (DoS) attacks and increases efficiency. SANA [2] presents a cryptographic scheme that enables aggregating devices to be untrustworthy and the attestation result to be *publicly verifiable* by multiple verifiers. However, the cryptographic scheme relies on pairing-based cryptography, which is computationally expensive, such that SANA imposes much higher computational costs compared to other protocols. SALAD [29] achieves *robustness* against network dynamics and disruptions by distributing the attestation result to all devices, instead of routing it along a fragile tree topology. Thus, SALAD can be applied when devices are mobile and lack continuous connectivity. Nevertheless, SALAD lacks an efficient attestation result aggregation and therefore provides only limited scalability.

DARPA [21] and SCAPI [28] are, in addition to software attacks, further able to detect *physical attacks*. To detect hardware attacks, DARPA and SCAPI build on the assumption that a physical adversary needs to take a device offline for a continuous amount of time δ_a to successfully tamper with it [11], [12]. In DARPA, each device emits an authenticated heartbeat every δ_a time that is logged by all other devices in the network. During attestation, the verifier receives the heartbeat log of all devices and considers devices that missed sending a heartbeat in at least one period as physically compromised. In SCAPI, devices share a session key that is periodically regenerated. Obtaining the newest session key requires devices to authenticate with the previous key. Physically attacked devices miss at least one session key, whereupon they are excluded from the network. SCAPI enhances DARPA's scalability and robustness by reducing the number of exchanged messages from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, and relying on a unidirectional 1-to-n link instead of a bidirectional n-to-n link. Nevertheless, SCAPI is still error-prone, as (i) each device must participate in the session key regeneration every $\delta_a/2$ time, (ii) devices that are unequipped with secure hardware cannot participate in the protocol and accelerate its execution, and (iii) the protocol is unable to recover from network splits, such that groups of devices that are separated from one another for longer than one session period consider each other as physically compromised.

Table I summarizes important properties for the attestation of autonomous embedded systems and indicates which collective attestation protocols provide these properties.

Secure Aggregation. In order to reduce the communication and storage overhead in sensor networks, so-called secure in-network aggregation schemes have been proposed. They combine data reported by multiple devices into a small aggregate in a way that the authenticity of the reported data is preserved. Unfortunately, in-network aggregation schemes can only detect manipulations retrospectively after additional communication rounds [44], [47], with a bounded probability [37], [47], or when less than a few devices are corrupt [20].

In contrast, cryptographic aggregation schemes provide rigorous security guarantees. They enable Message Authentication Codes (MACs) or digital signatures from multiple parties to be combined in a compact aggregate. Using the aggregate, a verifier can ensure that the data is authentic and originates from all parties. MAC aggregation schemes [26] require a verifier to hold the (secret) MAC key of all parties. This makes them inapplicable when the aggregate should be verifiable by multiple potentially untrustworthy verifiers. By contrast, using aggregate signature schemes [7], [33], a verifier must only hold the public keys of all parties. Multisignature schemes [15], [23], [35] are a special case of aggregated signatures, where all parties must sign the same data. Both aggregate and multisignature schemes are commonly based on RSA [23], discrete logarithms [6], [34], [35], pairings [7], [33], and lattices [15]. In this work, we apply Schnorr multisignatures [6], [34], which are one of the simplest, most-well understood, and efficient multisignature schemes [43]. On the downside, Schnorr multisignatures are generated in an interactive protocol, can only be aggregated at the time of signing, and are larger than some pairing-based signatures.

III. Background

In this section, we describe our system model (§ III-A), security assumptions (§ III-B), adversary model (§ III-C), and depict the signature scheme required by our protocol (§ III-D).

A. System Model

We consider a network with heterogeneous interconnected embedded devices. The network topology is a mesh, which can be static or dynamic. In dynamic networks, connections

between devices can frequently change in unpredictable ways. Yet, our protocol is designed for networks where all functional and uncompromised devices are reachable from time to time.

Each device $\mathcal{D}_i$ participating in our attestation protocol is considered to be a verifier $\mathcal{V}_i$, and can additionally be a prover $\mathcal{P}_i$. This implies that any prover at the same time also acts as a verifier. Prover devices typically perform privacy-, security-, or safety-critical tasks, which is why their system state must be verifiable for all network devices. Provers implement the necessary security requirements (§ III-B) and perpetually run the attestation protocol. The protocol allows any device in the network to check whether the software of each prover is in a trustworthy state and its hardware is untampered. Provers that passes both checks are called *healthy*, else *compromised*. As opposed to provers, devices that are only verifiers do not need to implement secure hardware and are not required to periodically execute the protocol. Thus, any device, including entities that are potentially untrustworthy or only occasionally connect to the network, e.g., an external network operator, can act as a verifier. For simplicity, we assume that all devices are deployed and maintained by a trusted network operator $\mathcal{O}$.

B. Security Assumptions

Common for collective attestation protocols that consider physical attacks [21], [28], we assume that each prover device $\mathcal{P}_i$ provides minimal secure hardware features for remote attestation [17] and a write-protected real-time clock (RTC). The secure hardware features can be implemented by Read-Only Memory (ROM) that stores the protocol code and a simple Memory Protection Unit (MPU) that allows only protocol code to access secrets like cryptographic keys. Both, secure hardware features and a write-protectable RTC are already available in many commodity embedded devices [41], [45]. We further assume that the clocks of all healthy devices are loosely synchronized. The execution space, where the stated security assumptions are fulfilled, is henceforth referred to as Trusted Execution Environment (TEE).

Additionally, we rely on the common assumption that any successful attack to bypass the TEE requires disabling the target device for a non-negligible amount of time [11], [12], [21], [28]. We acknowledge that non-invasive physical attacks, e.g., cache or power side-channels, may enable an adversary to bypass secure hardware during the operation of a device. However, this work focuses on the detection of invasive and semi-invasive attacks, which is why we expect that provers implement the various proposed mechanisms to prevent non-invasive attacks [39]. By contrast, invasive and semi-invasive attacks cannot be fully prevented by technical measures on the device itself. Performing these attacks an adversary directly accesses the internal components of the target device, which requires at least the decapsulation of the device and potentially also the bypass of hardware tamper resistance [39]. Therefore, invasive and semi-invasive attacks require an adversary to take a device offline and analyze it with specialized laboratory equipment for hours up to weeks [5], [42]. Depending on the hardware of provers and the capabilities of an adversary, we further expect the number of physical attacks that the adversary can accomplish simultaneously to be bound, e.g., due to limited resources in terms of equipment or personnel.

Finally, we expect all cryptographic schemes and the implementation of our proposed protocol to be secure.

C. Adversary Model

We assume an adversary $\mathcal{A}dv_{sw}$, who can eavesdrop, modify, delete, or insert any message between all entities in the network (Dolev-Yao model). $\mathcal{A}dv_{sw}$ is able to perform software attacks on all devices, whereupon $\mathcal{A}dv_{sw}$ has full control over their execution state, besides for code that is executed in the TEE. Additionally $\mathcal{A}dv_{sw}$ can read from or write to any memory outside of the TEE.

Furthermore, we consider a stronger adversary $\mathcal{A}dv_{hw}$, who has the same capabilities as $\mathcal{A}dv_{sw}$ but can additionally physically attack devices in the network. After successfully tampering with a device, $\mathcal{A}dv_{hw}$ has full control over its clock as well as code and data in its TEE. However, as explained in the security assumptions (§ III-B), we assume that $\mathcal{A}dv_{hw}$ is limited in the following ways:

(1) $\mathcal{A}dv_{hw}$ must take a prover $\mathcal{P}_i$ offline for at least the uninterrupted *attack time* δ_a to physically compromise $\mathcal{P}_i$.
(2) $\mathcal{A}dv_{hw}$ is unable to physically tamper with more than the *concurrency factor* β provers simultaneously.

The parameters δ_a and β can be adjusted to the hardware protection of all provers and the required security level. If it is assumed that $\mathcal{A}dv_{hw}$ can compromise all provers concurrently, i.e., β is equal to the number of all provers, physical attacks are still detected by our protocol, albeit less robustly (§ V-C).

We refer to an attestation protocol as secure, if $\mathcal{A}dv_{sw}$ and $\mathcal{A}dv_{hw}$ are unable to fake a healthy system state for a prover that is at the time of its attestation in a compromised software and/or hardware state.

D. Schnorr Multisignatures

Our proposed protocol makes use of Schnorr multisignatures to reduce the signature size and the computational costs to verify multiple signatures. Schnorr signatures [40] rely on a cyclic group $\mathbb{G}$ of prime order q, a generator g of $\mathbb{G}$, and a hash function $\mathcal{H}$. A signature key pair (x, X) is generated by choosing a random secret key $x \in \mathbb{Z}_q$ and computing the corresponding public key $X = g^x$. In the Schnorr multisignature setting [6], there are n signers, who each possess an individual secret key $x_1, \ldots, x_n$ and a corresponding public key $X_1 = g^{x_1}, \ldots, X_n = g^{x_n}$. In order to collectively sign a common message m, signers perform the following steps:

1) $R_i \leftarrow$ GenCommit(): $r_i \leftarrow \mathbb{Z}_q$; $R_i = g^{r_i}$.
   Description: Each signer i picks a random secret $r_i \in \mathbb{Z}_q$ and computes a commitment $R_i = g^{r_i}$.
2) $R \leftarrow$ AggCommit($R_1, R_2, ..., R_n$): $R = \prod_{i=1}^{n} R_i$.
   Description: The R_i of each signer i is aggregated into a final commitment R by computing $R = \prod_{i=1}^{n} R_i$.
3) $s_i \leftarrow$ GenSig(R, m): $c = \mathcal{H}(R|m)$; $s_i = r_i + cx_i$.

Description: Each signer i computes the common challenge $c = \mathcal{H}(R|m)$, which is based on the aggregated commitment R and the message m to be signed. Afterwards each signer i generates its partial signature $s_i = r_i + cx_i$.

4) $s \leftarrow \text{AggSig}(s_1, s_2, ..., s_n)$: $s = \sum_{i=1}^{n} s_i$.
Description: The signature s_i of each signer i is aggregated in a final signature s by computing $s = \sum_{i=1}^{n} s_i$. The final aggregated signature consists of the tuple (R, s).

To verify the aggregated signature (R, s) with the public keys of all signers, a verifier performs the following steps:

1) $X \leftarrow \text{AggKey}(X_1, X_2, ..., X_n)$: $X = \prod_{i=1}^{n} X_i$.
Description: The verifier aggregates the public keys of all participating signers by computing $X = \prod_{i=1}^{n} X_i$.

2) *valid*/*invalid* $\leftarrow$ Verify(R, s): $g^s \stackrel{?}{=} RX^c$.
Description: The verifier regards the aggregated signature as valid if $g^s = RX^c$, and in the other case as invalid.

Note that a corrupt signer could set its public key to $X_1 = g^{x_1}(\prod_{i=2}^{n} X_i)^{-1}$ and then forge valid signatures on behalf of all n signers. However, this so-called rogue-key attack [6], [34] is irrelevant in our attestation protocol, as all keys are predeployed (§ IV-A) and only uncompromised devices are able to participate in the signing process (§ IV-B).

IV. Attestation Protocol

In this section, we describe our attestation protocol PASTA, which consists of three different phases. In the *deployment phase*, each device is set up once by the network operator. Afterwards, prover devices in the network continually execute the *token generation phase*, in which provers repeatedly generate tokens. Each token attests the software and hardware integrity of all provers that participated in its generation, at the generation time. Simultaneous to the token generation phase, all devices, i.e., provers and verifiers, perform the *token exchange phase*. In this phase, devices distribute, verify, and validate the generated tokens from the provers. Devices eventually use their verified and validated tokens to determine the integrity of all provers in the network. In the following, we describe the deployment (§ IV-A), token generation (§ IV-B), and token exchange phase (§ IV-C). Finally, we explain the token validation (§ IV-D), which is part of the token exchange. In Appendix A and B, we demonstrate the security of PASTA.

A. Deployment Phase

Initially, devices in the network are deployed and set up by the network operator $\mathcal{O}$. Each device $\mathcal{D}_i$ is equipped with a unique identifier i and the public signature key X_O of $\mathcal{O}$. Devices that are provers additionally store a token signature key pair $(\mathrm{x}_i, \mathrm{X}_i)$. A prover $\mathcal{P}_i$ uses its private token key x_i to generate tokens that prove its integrity. To verify tokens from $\mathcal{P}_i$, a device $\mathcal{D}_j$ requires the public token key X_i of $\mathcal{P}_i$. Therefore, all devices in the network store the public keys of all prover devices. In addition, any two devices $\mathcal{D}_i$ and $\mathcal{D}_j$ hold a unique symmetric channel key ck_{ij}, which they use to authenticate any communication between them. To save storage, $\mathcal{O}$ may equip devices with a further key pair and a certificate over the public key. This enables devices to establish a channel key on demand and exchange public token keys ad-hoc. Provers store and execute all protocol data and code inside their TEE. Thus, an adversary must perform physical attacks to obtain secrets or tamper with the protocol execution.

Moreover, all devices maintain a token set TS, which initially contains the initialization token T_0. Each token T consists of a timestamp $T.ts$, a list of device identifiers $T.ids$, an aggregated signature $T.sig$, and a Boolean flag $T.valid$. $T.ts$ records the time when the token was generated, $T.ids$ stores the identifiers of all provers that participated in the token generation, $T.sig$ contains the aggregated signature from all participating provers computed over $T.ts$ and $T.ids$, and $T.valid$ indicates whether the device that stores T has successfully validated T ($T.valid = true$) or not. A device $\mathcal{D}_i$ considers T as valid, if $\mathcal{D}_i$ has ensured that all provers listed in $T.ids$ were at time $T.ts$ never offline for longer than the attack time δ_a, hence, were physically healthy at $T.ts$. The validity flag is not protected by the signature $T.sig$, because $T.valid$ is volatile and set by each device itself. Since all provers are assumed to be healthy at protocol start, the initialization token T_0 stores the start time of the protocol in $T_0.ts$, the identifier of all provers in $T_0.ids$, the aggregated signature over $T_0.ts$ and $T_0.ids$ from all provers in $T_0.sig$, and *true* in $T_0.valid$.

For convenience, we assume the initial deployment of all devices in the network. Nonetheless, $\mathcal{O}$ can enroll a new device $\mathcal{D}_i$ at any later stage. To this end, $\mathcal{D}_i$ is initialized as described, but additionally equipped with a certificate that allows other devices to establish required keys with $\mathcal{D}_i$ on demand. If $\mathcal{D}_i$ is a prover device, it also holds a special token that contains its deployment time and the identifier and signature of $\mathcal{O}$ and $\mathcal{D}_i$. Table II summarizes relevant definitions.

B. Token Generation Phase

Overview. After deployment, prover devices periodically generate tokens that attest their integrity at the token generation time. To this end, provers recurrently execute the token generation protocol, which consists of two communication rounds.

In the first round, a virtual spanning tree is arranged in the network, whose root is a particular initiator prover device that starts the protocol. The tree allows data from the initiator to be efficiently broadcasted to all provers and data from all provers to be efficiently propagated back to the initiator. At first, all provers receive a token generation request from the initiator and then respond with their Schnorr commitments and their device identifiers. Both the identifiers and commitments are collected and aggregated in each hop, and then forwarded to the next hop along the tree towards the initiator. Eventually, the initiator receives the identity of all participating provers and their commitments. In the second round, the initiator broadcasts the aggregated commitments, current time, and prover identifiers, which forms the attestation challenge. Subsequently, provers generate a partial signature over the time and prover identifiers, which is also aggregated and propagated along the tree to the initiator. After collecting

Acronym	*Usage*
$\mathcal{D}_i/\mathcal{V}_i/\mathcal{P}_i$	device / verifier / prover with identifier i
$\mathcal{O}$	trusted network operator
$\mathrm{X}_{\mathcal{O}}$	public signature verification key of $\mathcal{O}$
x_i / X_i	private / public token key of prover $\mathcal{P}_i$
ck_{ij}	channel key between $\mathcal{D}_i$ and $\mathcal{D}_j$
T	token; T contains: $T.ts, T.ids, T.sig, T.valid$
δ_a	time $\mathcal{A}dv_{hw}$ must take $\mathcal{D}_i$ offline during attack
β	number of devs $\mathcal{A}dv_{hw}$ can compromise in δ_a
δ_{gen}	time between periodic token generations
Time()	returns current time
VerifySW()	returns if local software state is trustworthy
IsHealthy($\mathcal{P}_i$)	returns if $\mathcal{P}_i$ is healthy ($\neq$ compromised)
ReqNewToken(δ)	returns if token needs to be regenerated

TABLE II: Notation.

all signatures, the initiator assembles the final token T. T contains the time in $T.ts$, the prover identifiers in $T.ids$, and the aggregated signature over $T.ts$ and $T.ids$ in $T.sig$.

During token generation, provers communicate authenticated using their channel key *ck*, which prevents a network adversary from manipulating the protocol execution. To also protect against compromise of the software and hardware, provers only participate in the token generation if they have ensured their own software integrity and the physical integrity of their neighboring provers in the tree. To verify their own software integrity, provers implement a function VerifySW(), which measures the local state and returns *true* if the software is in a trustworthy state[2]. We abstract from implementation details of VerifySW() to support a wide range of attestation mechanisms [1], [14], [36], [48]. Recall that any protocol code is executed inside the TEE, such that $\mathcal{A}dv_{sw}$ is unable to bypass VerifySW(). Furthermore, provers implement a function IsHealthy($\mathcal{P}_j$), which determines the integrity of provers using tokens that were generated in previous runs of the token generation protocol and were then propagated, verified, and validated in the token exchange phase. IsHealthy($\mathcal{P}_j$) returns *true* if $\mathcal{P}_j$ has proven to be healthy within the last δ_a time and *false* otherwise. Since $\mathcal{A}dv_{hw}$ requires at least δ_a time to perform physical attacks, provers that pass IsHealthy($\mathcal{P}_j$) are physically healthy at the present time. Details of IsHealthy($\mathcal{P}_j$) are given in the next subsection (§ IV-C).

In the following, we explain both rounds of the token generation protocol in detail.

Round 1: Initialization. To ensure that prover devices periodically engage in the token generation, each prover monitors the freshness of its own tokens. A prover device initiates a token generation, whenever it notices that the newest token T in whose generation it participated is older than a specific generation interval δ_{gen}, as indicated by the function ReqNewToken(δ_{gen}), shown in Algorithm 1.

[2]Depending on the implementation, the software state may actually be measured prior to the invocation of VerifySW(). For instance, a Trusted Platform Module (TPM) computes hash values over software binaries before they are loaded [36], so that VerifySW() only compares already performed measurements from the TPM with known good reference values.

Algorithm 1 : $\mathcal{P}_i$ determines if token generation is required.

```
1: procedure ReqNewToken(δ)
2:     for T ∈ TS do
3:         if i ∈ T.ids and T.ts > Time() − δ then
4:             return false
5:     return true
```

The generation interval δ_{gen} must be a fraction of δ_a ($\delta_{gen} < \delta_a$) to ensure that each prover participates at least once in a token generation within δ_a time. A prover that fulfills this requirement has never been offline for longer than δ_a time and is therefore considered to be physically healthy by all network devices. A small δ_{gen} enhances resilience to network disruptions, as tokens are generated more frequently, but on the downside increases communication and computational effort.

As illustrated in Figure 1, the token generation protocol starts with a prover $\mathcal{P}_i$ that checks whether it requires a new token and is in a trustworthy software state, using ReqNewToken(δ_{gen}) and VerifySW(). If both checks pass, $\mathcal{P}_i$ will initiate a token generation in the network. To this end, $\mathcal{P}_i$ stores its identifier i in ids_i and the current time in ts, which then both form the token initiation message msg_{init}. Furthermore, $\mathcal{P}_i$ generates its Schnorr commitment R_i over a random secret (§ III-D). Next, any neighboring prover $\mathcal{P}_j$ that is in a physically healthy state, determined by $\mathcal{P}_i$ using IsHealthy($\mathcal{P}_j$), is sent msg_{init}. All messages are authenticated by the sender and verified by the receiver using the respective channel key. Invalid messages are dropped and not processed. For clarity, we omitted message authentication in Figure 1.

A prover device $\mathcal{P}_j$ that receives a message msg_{init} initially ensures that the token generation was initiated within the last δ_a time, as Time() $< ts + \delta_a$. This time check is executed whenever provers receive a message in the token generation protocol to prevent a network adversary from replaying recorded messages from previous runs of the protocol. Next, $\mathcal{P}_j$ checks whether it would soon require a token generation, indicated by a call to the function ReqNewToken() with a specific join interval δ_{join} that is smaller than the generation interval δ_{gen} ($\delta_{join} < \delta_{gen}$). Using a smaller interval to join a token generation than to initiate one saves overhead, as it prevents the costly initiation of multiple token generations in close succession. Afterwards, $\mathcal{P}_j$ checks whether the prover from which it received msg_{init} is physically healthy and itself is in a trustworthy software state. If both checks also pass, $\mathcal{P}_j$ joins the token generation session. To this end, $\mathcal{P}_j$ stores its own identifier j and the received identifiers ids_i in its own ids_j and generates its Schnorr commitment R_j. Additionally, $\mathcal{P}_j$ invites further healthy neighboring provers to join the current token generation session by propagating msg_{init}. Note that provers only ensure the integrity of their neighboring provers, but not of all provers that participate in the token generation. This is because provers can rely on the trustworthiness of healthy neighbors and be confident that they in turn ensure the physical integrity of their neighbors. Thus, a chain of trust is established that prevents any physically compromised prover from participating in the token generation.

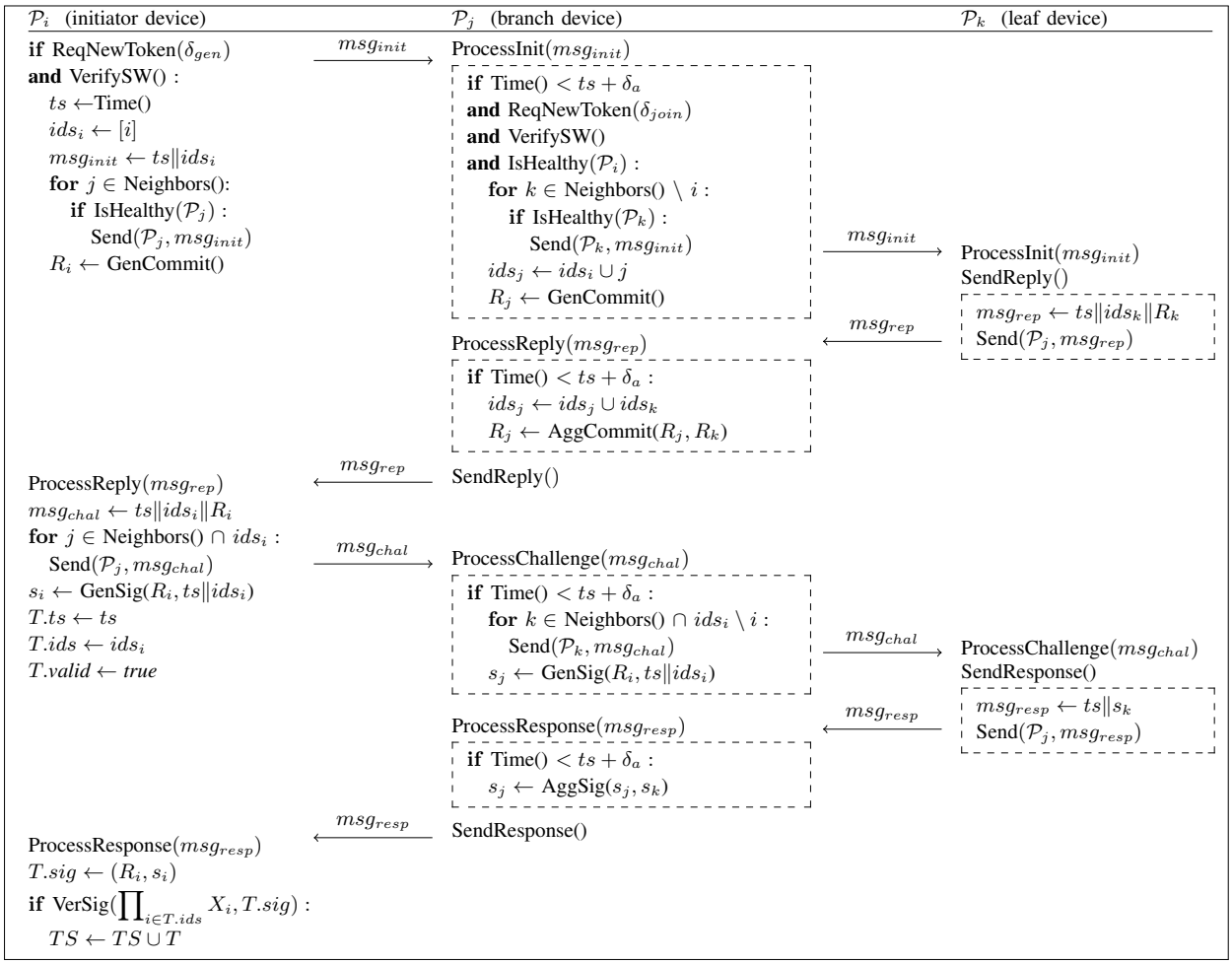


Fig. 1: Illustration of the token generation protocol. Prover $\mathcal{P}_i$ initiates the token generation, whereupon a spanning tree is constructed. $\mathcal{P}_j$ represents a branch device, which is connected to $\mathcal{P}_i$ and $\mathcal{P}_k$. $\mathcal{P}_k$ is a leaf device, as it is only connected to $\mathcal{P}_j$. When receiving messages, provers unpack their content, e.g., ts and ids_i from msg_{init}. For clarity, we omitted the authentication of messages. In practice, all exchanged messages between any two provers $\mathcal{P}_i$ and $\mathcal{P}_j$ are authenticated by the sender and verified by the receiver using the channel key ck_{ij}.

Prover devices that receive a msg_{init} from $\mathcal{P}_j$ likewise perform the same steps as $\mathcal{P}_j$. This way, beginning with the initiator device $\mathcal{P}_i$, a spanning tree is arranged in the network, where parent provers invite their children to join the token generation. Eventually, msg_{init} is received by leaf devices that have no children, because all their neighbors are already participating in the token generation, are compromised, or recently generated a token. Receiving msg_{init}, each leaf device $\mathcal{P}_k$ answers its parent with a msg_{rep} message, which contains the token generation timestamp ts, the prover identifier ids_k, and the Schnorr commitment R_k. A prover $\mathcal{P}_j$ that receives a msg_{rep} from a child prover $\mathcal{P}_k$ merges the received device identifiers ids_k with its stored identifiers in ids_j and aggregates the received Schnorr commitment R_k with its stored commitment in R_j (§ III-D). After processing the msg_{rep} from all child provers, provers create their own msg_{rep} and send it to their own parent devices, which in turn perform the same steps. Finally, the initiator prover $\mathcal{P}_i$ receives and aggregates the identifiers and Schnorr commitments of all provers that are participating in the token generation session.

Round 2: Finalization. After the first round, the only data that the initiator prover $\mathcal{P}_i$ is missing to generate the final token T, is the aggregated signature s of all participating provers, computed over $T.ts$ and $T.ids$. To collect s, $\mathcal{P}_i$ prepares a message msg_{chal}, which stores ts, ids_i, and R_i. Next, msg_{chal} is propagated in the network from device to device, using the tree topology from the first round. With the content of msg_{chal}, each participating prover $\mathcal{P}_j$ then computes its partial signature s_j (§ III-D). Afterwards, partial signatures are forwarded along the tree topology back to the initiator $\mathcal{P}_i$ and are aggregated in each hop, like commitments in the first round. $\mathcal{P}_i$ eventually receives and aggregates the partial signatures of all participating provers in s_i. Finally, $\mathcal{P}_i$ builds the token T, which stores ts in $T.ts$, ids_i in $T.ids$, *true* in

$T.valid$, and the tuple (R_i, s_i) in $T.sig$. In case $T.sig$ is valid, $\mathcal{P}_i$ adds T to its token set TS. Thus, T attests that all provers listed in $T.ids$ have been healthy at $T.ts$. Finally, $\mathcal{P}_i$ uses the token exchange phase (§ IV-C) to distribute T in the network.

Remarks. For clarity, we described the token generation in a simplified version. In practice, timers and error messages must prevent the protocol from hanging if messages are lost or replayed, verification checks fail, or provers are invited to join the same token generation multiple times. Furthermore, to support network dynamics during token generation, an ad-hoc routing protocol must enable each device to reach its (initial) neighboring devices in the virtual tree topology. Additionally, we recommend that provers store different values for δ_{gen}. For instance, a dedicated leader prover may use a lower δ_{gen} than other provers. This prevents multiple provers from initiating a token generation simultaneously, which would result in the unnecessary generation of multiple tokens. Instead, the dedicated leader prover would always initiate the token generation and other provers would only take over, if the leader is unavailable.

C. Token Exchange Phase

Token Exchange. After executing the token generation protocol, the initiator device holds a new token that testifies the integrity of one or multiple provers at the token generation time. To make this information available to all network devices, generated tokens are propagated and stored by all devices. For this purpose, devices within direct communication range continuously synchronize their token set TS. A synchronization is initiated when devices connect to each other or a connected neighboring device just generated or received a new token. For efficiency, devices only exchange tokens that the receiver is missing. To determine the missing tokens, devices initially compare the number and checksums of their stored tokens.

Any device, including untrustworthy devices, can participate in the exchange of tokens and act as a data mule. This is feasible and secure, as the integrity and authenticity of each token T is protected by its contained aggregated signature $T.sig$. Devices verify the signatures of all received tokens. Corrupt tokens, which fail the verification, are discarded and are not added to the token set TS. Note that the signature verification only protects against $\mathcal{A}dv_{sw}$, but not $\mathcal{A}dv_{hw}$, who is able to forge token signatures of provers after physically tampering with them. Attacks from $\mathcal{A}dv_{hw}$ are prevented by the token validity flags, which are not transferred during token exchange. Instead, each device sets $T.valid$ to *false* for any token T that is received and added to TS. This indicates that the device storing T has not (yet) ensured that the provers listed in $T.ids$ have never been offline for longer than δ_a time. Because provers listed in invalid tokens may potentially be in a compromised state, invalid tokens are disregarded when determining the integrity of provers with IsHealthy(). To determine and set the validity of stored tokens, each device performs a so-called token validation after it has synchronized all tokens with its neighboring devices, which is described in detail in the next subsection (§ IV-D). The token validation checks whether $\mathcal{A}dv_{hw}$ has been unable to physically tamper with all provers listed in $T.ids$ based on the limitations of $\mathcal{A}dv_{hw}$ (§ III-C), the timestamp in $T.ts$, the current time, and the chain of trust derived from already validated stored tokens. Any token T that passes these checks is marked as valid, with $T.valid$ set to *true*. Tokens that could not be validated are still kept, i.e., not discarded from TS, since tokens received in the future may prove their validity.

In case a receiver $\mathcal{D}_i$ of tokens determines with IsHealthy($\mathcal{P}_j$) that the sender $\mathcal{P}_j$ is a healthy prover, $\mathcal{D}_i$ can take advantage of $\mathcal{P}_j$'s trustworthiness. To this end, transmitted tokens are protected by a MAC using the channel key ck_{ij} of involved devices $\mathcal{D}_i$ and $\mathcal{P}_j$. Thus, $\mathcal{D}_i$ can rely on the authenticity of the channel as well as the correct behavior of $\mathcal{P}_j$. This enables $\mathcal{D}_i$ to omit verifying the signature of received tokens as well as transferring the token validity flags from $\mathcal{P}_j$, which both saves computational resources.

Devices routinely discard tokens whose timestamp is so old that they neither play a role in determining the healthiness of provers ($T.ts \leq$ Time() $- \delta_a$) nor in establishing validity in other tokens ($T.ts \leq$ Time() $- p/\beta \cdot \delta_a$), with β being the concurrency factor and p the total number of provers.

Determining the Integrity of Provers. Devices that have synchronized and validated their token set TS can afterwards determine the integrity of provers. To this end, devices execute the function IsHealthy($\mathcal{P}_i$), which is shown in Algorithm 2.

Algorithm 2 : A device determines the integrity of a prover.

```
1: procedure IsHealthy(P_i)
2:     for T ∈ TS do
3:         if i ∈ T.ids and Time() < T.ts + δ_a and T.valid then
4:             return true
5:     return false
```

Devices that execute IsHealthy($\mathcal{P}_i$) check whether $\mathcal{P}_i$ participated in the generation of a token T whose validity has been ensured and which was generated within the last δ_a time. A prover $\mathcal{P}_i$ that passes these checks has proven to be in a trustworthy software state between the time $T.ts$ and now. In addition, $\mathcal{P}_i$ is physically healthy at the present time, as T testifies the physical integrity of $\mathcal{P}_i$ at $T.ts$ and successful physical attacks require at least the attack time δ_a.

According to the adversary model (§ III-C), our attestation protocol is secure, if $\mathcal{A}dv_{sw}$ and $\mathcal{A}dv_{hw}$ are unable to fake a healthy system state for a prover $\mathcal{P}_a$ that is at the time of its own attestation in a compromised state. To fake a healthy state for $\mathcal{P}_a$, $\mathcal{A}dv_{hw}$, hence, also the weaker $\mathcal{A}dv_{sw}$, would need trick a device $\mathcal{D}_i$ into storing a token T that passes IsHealthy($\mathcal{P}_a$). To this end, $\mathcal{A}dv_{hw}$ must forge T in a way that it contains a valid signature for $\mathcal{P}_a$, since $\mathcal{D}_i$ will only accept T during token exchange, if T passes the signature verification. To forge a valid token signature for $\mathcal{P}_a$, $\mathcal{A}dv_{hw}$ must possess the private token key x_a of $\mathcal{P}_a$. By performing network attacks or compromising the software on $\mathcal{P}_a$, $\mathcal{A}dv_{hw}$ is unable to get access to x_a. This is because all secrets and protocol code are stored and execute inside the TEE of provers and never leave

the TEE. In addition, the TEE is immutable by compromised software (§ III-B) and the token generation protocol code enforces that provers with a compromised software quit the protocol execution. Hence, a software-compromised prover $\mathcal{P}_a$ is unable to access its private token key x_a. However, $\mathcal{Adv}_{hw}$ may also physically compromise $\mathcal{P}_a$ to gain access to x_a, which enables $\mathcal{Adv}_{hw}$ to forge valid token signatures on behalf of $\mathcal{P}_a$. Yet, because $\mathcal{Adv}_{hw}$ must take $\mathcal{P}_a$ offline for δ_a time during the physical attack, $\mathcal{P}_a$ will not generate a token for more than δ_a time. Thus, after the physical attack is completed, all devices will only store outdated tokens from $\mathcal{P}_a$ and regard $\mathcal{P}_a$ as compromised when executing IsHealthy($\mathcal{P}_a$). As a result, $\mathcal{P}_a$ is from then on neither able to participate in the token generation with other healthy provers, nor able to issue a token that will pass the token validation on healthy devices, as described in the following subsection (§ IV-D).

Note that devices cannot distinguish between an unreachable and a compromised prover. This is because $\mathcal{Adv}_{hw}$ is assumed to have full control over the network and, thus, can prevent the generation and exchange of tokens in the network.

D. Token Validation

Overview. The token validation is the essential part of PASTA to detect physical attacks. Recall that a stored validated token T testifies that all provers listed in $T.ids$ were physically healthy at $T.ts$. By contrast, the authenticity of stored invalid tokens has not been ensured (yet), meaning that each invalid token could have been forged by $\mathcal{Adv}_{hw}$ and list physically compromised provers in $T.ids$ and a bogus timestamp in $T.ts$. During token validation, a device $\mathcal{D}_i$ updates the validity flags of all stored invalid tokens, whereby any invalid token for which $\mathcal{D}_i$ can rule out that it is forged by $\mathcal{Adv}_{hw}$ is marked as valid. More specifically, $\mathcal{D}_i$ uses its already validated tokens to build a chain of trust between all physically healthy provers, which are provers that were never offline for longer than the attack time δ_a. Starting with the initialization token pre-deployed as valid, $\mathcal{D}_i$ iteratively uses the information from already validated tokens to validate further invalid tokens. In this process, $\mathcal{D}_i$ makes use of the time and simultaneously limitations of $\mathcal{Adv}_{hw}$, which are stated in (1) and (2) in § III-C. In case $\mathcal{D}_i$ attempts to validate a forged token T_a, $\mathcal{D}_i$ will be unable to establish a chain of trust for provers listed in T_a, since they have been offline for at least δ_a time during the physical attack. Consequently, $\mathcal{D}_i$ will not mark T_a as valid.

The token validation consists of two steps. In each step, tokens are validated taking assumption (1), respectively (2), on $\mathcal{Adv}_{hw}$ into account. Below, we describe both steps in detail.

(1) Validation of Time Assumption. In the first step, already validated tokens are used to determine the set of healthy provers. With the set of healthy provers, invalid tokens are then validated. The process is illustrated in Algorithm 3 and explained in the following.

2-5: Initially, a device $\mathcal{D}_i$ determines and stores the identifiers of all provers that $\mathcal{D}_i$ considers to be healthy in $hdevs$. For this purpose, $\mathcal{D}_i$ uses the function IsHealthy(), which we described in the previous subsection (IV-C).

6-9: Next, $\mathcal{D}_i$ makes use of $hdevs$ to validate tokens from its token set TS. In detail, the validity flag $T.valid$ of any token T that lists a healthy prover, which is the case if $T.ids \cap hdevs \neq \emptyset$, is set to *true*. This is because healthy provers only engage in the token generation with other healthy provers. Since $hdevs$ testifies that at least one prover in $T.ids$ is healthy, all provers in $T.ids$ must be healthy. Thus, T cannot have been forged by $\mathcal{Adv}_{hw}$ and is set valid. Next, $\mathcal{D}_i$ starts the procedure all over again. This is necessary, as T may now testify the integrity of additional provers that are currently not contained in $hdevs$. With the extended $hdevs$, already processed tokens that were not validated in the current iteration of the procedure may be validated in the next iteration.

Algorithm 3 : $\mathcal{D}_i$ validates its stored tokens in step one.

```
1: procedure ValidateTime()
2:     hdevs ← ∅                                  ▷ provers considered healthy
3:     for i = 1, 2, ..., n do
4:         if IsHealthy(P_i) then
5:             hdevs ← hdevs ∪ i
6:     for T ∈ TS do
7:         if (T.valid = false) and (T.ids ∩ hdevs ≠ ∅) then
8:             T.valid ← true
9:             go to 3
```

For clarity, Figure 2a shows an example for the validation of the time assumption.

(2) Validation of Simultaneity Assumption. In the second step, devices additionally make use of the assumption that $\mathcal{Adv}_{hw}$ is unable to compromise more than β provers per δ_a time concurrently. Based on this assumption, validated tokens that are outdated, i.e., older than δ_a, can be used to validate remaining invalid tokens from the first token validation step. This is possible, as outdated validated tokens may testify the healthiness of so many provers at a past time that $\mathcal{Adv}_{hw}$ would be unable to have physically compromised all of them at the present time. Therefore, newer (yet) invalid tokens generated by one of those healthy provers must be valid.

More specifically, each validated token T_v testifies the healthiness of all provers listed in T_v at time $T_v.ts$. At the present time, $\mathcal{Adv}_{hw}$ can at the maximum have physically compromised $maxcdevs = \lfloor(\text{Time}() - T_v.ts)/\delta_a\rfloor \cdot \beta$ provers of T_v. If an invalid token T_i shares more than $maxcdevs$ provers with T_v, i.e., $|T_i.ids \cap T_v.ids| > maxcdevs$, then T_i must be valid. This is because to forge T_i, $\mathcal{Adv}_{hw}$ has to compromise all provers listed in T_i, which is, however, a contradiction to the fact that at least one prover in $T_i.ids \cap T_v.ids$ must be healthy. A larger intersection $|T_i.ids \cap T_v.ids|$ enables T_v to be older and still be used for validation, which increases robustness against device and network disruptions. To maximize this intersection, T_i and T_v can be extended by further tokens from TS that share the same device identifiers with either T_i or T_v. This way, two device identifier sets $idevs$ and $vdevs$ of invalid ($idevs$) and valid ($vdevs$) tokens can be used, which results in the extended intersection $idevs \cap vdevs$. The process is illustrated in Algorithm 4 and explained in the following.

2-5: A device $\mathcal{D}_i$ iteratively selects an invalid token T_i from TS and attempts to validate T_i in the following steps. Initially, the index i of T_i is recorded in $itoks$ and provers listed in T_i are stored in $idevs$.

6-12: Next, $itoks$ is iteratively extended by further tokens to be validated, and $idevs$ is enlarged by their listed prover identifiers. In this process, $itoks$ is constructed in a way that a single valid token in $itoks$ is capable of testifying the validity of all tokens in $itoks$. To this end, TS is examined for tokens that (i) are not contained in $itoks$ and are more recent than $T_i.ts$, and (ii) share more provers with $idevs$ than $\mathcal{A}dv_{hw}$ can compromise between $T_i.ts$ and now. A token T_k fulfills (i) if $k \notin itoks$ and $T_k.ts > T_i.ts$. To fulfill (ii), $|T_k.ids \cap idevs|$ must be larger than $maxcdevs = \lfloor(\text{Time}() - T_i.ts)/\delta_a\rfloor \cdot \beta$. If both checks pass, $itoks$ is extended by k and $idevs$ by $T_k.ids$. Afterwards, this step (6-12) is repeated.
The checks (i) and (ii) ensure that if all tokens in $itoks$ are valid, T_k must be valid, and vice versa. This is because between now and the time $T_i.ts$, being according to (i) the oldest of all tokens $itoks$ and T_k, $\mathcal{A}dv_{hw}$ can only compromise $maxcdevs$ provers. Since more than $maxcdevs$ provers are according to (ii) contained in $idevs \cap T_k.ids$, $\mathcal{A}dv_{hw}$ would have had too little time to tamper with all provers $idevs \cap T_k.ids$, which $\mathcal{A}dv_{hw}$ requires to forge T_k or all $itoks$ tokens. By ensuring that each token T_k added to $itoks$ fulfills (i) and (ii), $itoks$ obtains the property that a single valid token in $itoks$ is able to testify the validity of all tokens in $itoks$. This property is important for the following step (13-22).

13-22: Next, $\mathcal{D}_i$ determines which provers $vdevs \subseteq idevs$ were listed in already validated tokens, hence, were healthy at a past time t. If the amount of $vdevs$ is greater than $maxcdevs = \lfloor(\text{Time}() - t)/\delta_a\rfloor \cdot \beta$, at least one prover in $idevs$ must be healthy at the current time. Thus, at least one token from $itoks$, namely the token listing the healthy prover, must be valid. This implies that all tokens in $itoks$ must be valid due to the property of $itoks$.
In detail, validated tokens in TS are examined in descending order with regards to their timestamp. For a token T_v that attests the integrity of provers in $idevs$ at an earlier time $T_v.ts$, $maxcdevs$ amounts to $\lfloor(\text{Time}() - T_v.ts)/\delta_a\rfloor \cdot \beta$. For the first found token T_v that attests the integrity of provers in $idevs$, $vdevs$ amounts to the intersection between $idevs$ and $T_v.ids$: $vdevs = idevs \cap T_v.ids$. However, because provers that are healthy at $T_v.ts$ are also healthy at $T_{v'}.ts < T_v.ts$, $vdevs$ inherits all provers from previous iterations for any subsequently found token $T_{v'}$ that attests the integrity of provers in $idevs$: $vdevs = vdevs \cup (T_{v'}.ids \cap idevs)$. If it is eventually ensured that $idevs$ are healthy, as $|vdevs| > maxcdevs$, all tokens from $itoks$ are set valid. Since the newly validated tokens may be able to change the validity of stored invalid tokens, both steps of the token validation are executed again.

Algorithm 4 : $\mathcal{D}_i$ validates its stored tokens in step two.

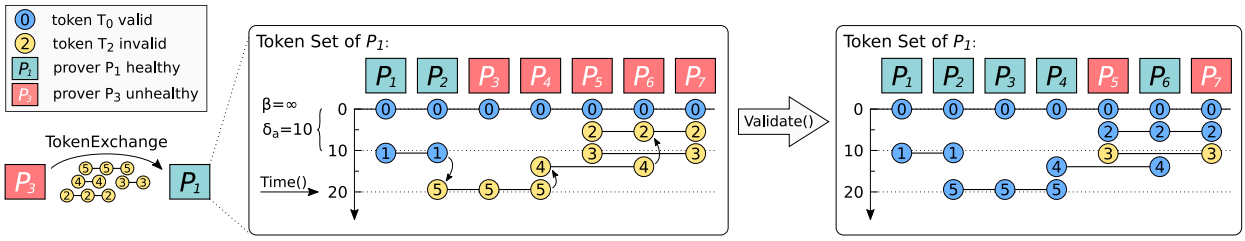
```
1:  procedure ValidateSimultaneity()
2:      for T_i ∈ TS do
3:          if T_i.valid = false then                ▷ attempt to validate T_i
4:              itoks ← {i}
5:              idevs ← T_i.ids
6:              for T_k ∈ TS do
7:                  if (k ∉ itoks) and (T_k.ts > T_i.ts) then
8:                      maxcdevs ← ⌊(Time() − T_i.ts)/δ_a⌋ · β
9:                      if |T_k.ids ∩ idevs| > maxcdevs then
10:                         itoks ← itoks ∪ k
11:                         idevs ← idevs ∪ T_k.ids
12:                         go to 6
13:             vdevs ← ∅
14:             for T_v ∈ reverse(sort(TS)) do    ▷ highest T.ts first
15:                 if (T_v.valid) and (T_v.ids ∩ idevs ≠ ∅) then
16:                     maxcdevs ← ⌊(Time() − T_v.ts)/δ_a⌋ · β
17:                     vdevs ← vdevs ∪ (T_v.ids ∩ idevs)
18:                     if |vdevs| > maxcdevs then    ▷ itoks valid
19:                         for k ∈ itoks do
20:                             TS.T_k.valid = true
21:                         ValidateTime()
22:                         go to 2
```

For clarity, Figure 2b shows an example for the validation of the simultaneity assumption.

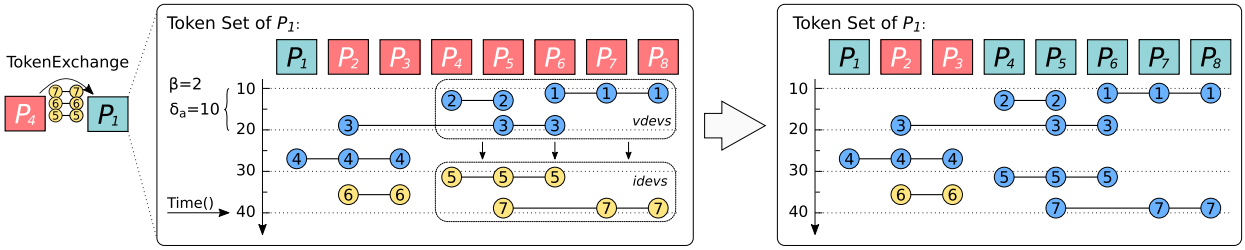
Further Solutions to Validate Tokens. The described token validation works well for devices that regularly exchange and validate tokens. However, some devices may only occasionally connect to the network to check the integrity of provers, such as, for instance, a verifier device from the network operator. Depending on their absence time and the parameters δ_a and β, such devices may have issues establishing the chain of trust in healthy provers and, therefore, may falsely regard healthy provers as compromised. In fact, for a proper token validation, each device needs to reconnect to the network at least every $\delta_{ex} < \lceil p/\beta \rceil \cdot \delta_a$ time, with p being the total number of healthy provers in the network. For instance, in a network with 1000 provers, an attack time δ_a of 10 min, and an attack concurrency factor β of 5, δ_{ex} amounts to 33.3 hours. In case a device is absent from the network for longer than δ_{ex} time, we propose two alternative solutions to validate tokens.

The first solution is to fall back to the approach of existing attestation protocols that detect physical attacks [21], [28]. They assume that $\mathcal{A}dv_{hw}$ is unable to physically tamper with a specific amount λ of provers in the network. Based on this assumption, a secure attestation of $p - \lambda + 1$ provers can be guaranteed. To this end, a device $\mathcal{D}_i$ determines all provers $pdevs$ that are listed in tokens with a timestamp more recent than δ_a: $T.ts > \text{Time}() - \delta_a$. If $|pdevs|$ is greater than $p - \lambda$, all $pdevs$ provers must be healthy. Subsequently, $\mathcal{D}_i$ marks all stored tokens that list a prover from $pdevs$ as valid.

The second solution is applicable in case provers provide some form of physical tamper evidence. In this case, the integrity of $hdevs$ provers is ensured by physically inspecting the particular provers. Next, all tokens that list a prover from $hdevs$ are set valid. Finally, all other stored tokens are validated by executing our proposed token validation.



(a) Prover $\mathcal{P}_1$ receives the tokens T_2, T_3, T_4, and T_5 from prover $\mathcal{P}_3$ at time 20. Before the token validation, $\mathcal{P}_1$ considers $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$, $\mathcal{P}_6$, and $\mathcal{P}_7$ to be compromised, since $\mathcal{P}_1$ does not store a validated and recent (i.e., less than Time() $-\ \delta_a$ time old) token that lists $\mathcal{P}_3$, $\mathcal{P}_4$, $\mathcal{P}_5$, $\mathcal{P}_6$, or $\mathcal{P}_7$. During token validation, $\mathcal{P}_1$ iteratively establishes a chain of trust based on already validated tokens, as indicated by the arrows. Initially, token T_1 testifies the integrity of $\mathcal{P}_1$ and $\mathcal{P}_2$ at time $T_1.ts \approx 11$. Because $\mathcal{A}dv_{hw}$ would had too little time to tamper with $\mathcal{P}_1$ or $\mathcal{P}_2$ between $T_1.ts$ and now, both provers must be physically healthy at the present moment. Since $\mathcal{P}_2$ must be healthy and is listed in T_5, T_5 cannot be forged by $\mathcal{A}dv_{hw}$, hence, is set valid. Next, the newly validated T_5 testifies the integrity of $\mathcal{P}_4$, which is why token T_4 is set valid. Finally, T_4 testifies that $\mathcal{P}_6$ is healthy, so that T_2 is set valid. Token T_3 cannot be validated, since $\mathcal{P}_1$ does not store a validated and recent token that testifies the integrity of provers listed in T_3, i.e, $\mathcal{P}_5$ and $\mathcal{P}_7$, at the present time. Thus, T_3 may be forged by $\mathcal{A}dv_{hw}$, hence, remains invalid. After token validation, $\mathcal{P}_1$ regards $\mathcal{P}_2$, $\mathcal{P}_3$, $\mathcal{P}_4$, and $\mathcal{P}_6$ as healthy, due to the newly received and validated tokens.



(b) Prover $\mathcal{P}_1$ receives the tokens T_5, T_6, and T_7 from prover $\mathcal{P}_4$ at time 40. The concurrency factor β is 2, which enables $\mathcal{P}_1$ to validate tokens based on the simultaneity assumption. As indicated by the arrows, $\mathcal{P}_1$ uses the outdated ($T.ts <$ Time() $-\ \delta_a = 30$) but already validated tokens T_1, T_2, and T_3 to validate the newly received tokens T_5 and T_7. In detail, T_1, T_2, and T_3 testify that $vdevs = \{\mathcal{P}_4, \mathcal{P}_5, \mathcal{P}_6, \mathcal{P}_7, \mathcal{P}_8\}$ were physically healthy at (at least) $T_1.ts \approx 11$, which is the oldest timestamp in T_1, T_2, and T_3. Between $T_1.ts$ and now, $\mathcal{A}dv_{hw}$ could at maximum have physically compromised $maxcdevs = \lfloor(\text{Time}() - T_1.ts)/\delta_a\rfloor \cdot \beta = 4$ provers of $vdevs$. Because $vdevs$ contains 5 provers, at least one prover of $vdevs$ must be healthy at the present time. This directly implies that T_5 or T_7 (or both) must be valid. For instance, assuming that $\mathcal{P}_4$ is healthy, T_5 must be valid. However, since T_5 and T_7 both list $\mathcal{P}_5$, the validity of T_5 further implies the validity of T_7 and vice versa. This is because assuming that either T_5 or T_7 is valid, $\mathcal{A}dv_{hw}$ would have too little time to physically tamper with $\mathcal{P}_5$ after $T_5 \approx 31$ or $T_7 \approx 39$, which is among others necessary to forge T_5 or T_7. Thus, both T_5 and T_7 must be genuine and are marked as valid. By contrast, $\mathcal{P}_1$ is unable to validate token T_6, generated by $\mathcal{P}_2$ and $\mathcal{P}_3$. This is because between $T_4.ts \approx 27$, at which $\mathcal{P}_2$ and $\mathcal{P}_3$ were healthy, and now, $\mathcal{A}dv_{hw}$ would have had enough time to physically compromise $\mathcal{P}_2$ and $\mathcal{P}_3$ and forge token T_6.

Fig. 2: Illustration of the token validation from the perspective of a prover $\mathcal{P}_1$. Both scenarios (a) and (b) represent highly disrupted networks, where provers only rarely had a connection to each other. Thus, provers were unable to periodically perform the token generation as a group. In (a), the concurrency factor β is set to ∞, which means that $\mathcal{A}dv_{hw}$ can physically compromise all provers concurrently. In (b), β is 2.

V. Evaluation

In this section, we first describe our implementation and show measurements conducted on low-end embedded devices (§ V-A). Next, we evaluate the scalability of PASTA by presenting simulations of static networks (§ V-B). Finally, we provide simulation results of PASTA in highly dynamic and disruptive networks to assess its robustness (§ V-C).

A. Implementation Setup & Measurements

Implementation. As a target platform for our implementation, we employed four ESP32-PICO-KIT V4 development boards. The core of the boards constitute a 7x7x0.94mm^2 ESP32-PICO-D4 system-in-package module, which features Wi-Fi and Bluetooth functionalities, 4 MB flash memory, and a 240 MHz dual-core 32-bit microprocessor. Due to its ultra-small size and low-energy consumption, the ESP32-PICO-D4 is well suited for space-limited or battery-operated applications, e.g., as wearable, medical, or sensor devices. Furthermore, it provides the Secure Boot feature and thus the minimal hardware properties required for remote attestation [16], [41].

We used SHA-256 as a cryptographic hash function and a Keyed-Hash Message Authentication Code (HMAC) based on SHA-256 for message authentication. To implement both, we made use of the mbed TLS code [3]. Our implementation of the Schnorr multisignature scheme is based on the Bitcoin cryptographic code [19], which offers a library for elliptic curve operations on curve secp256k1.

Runtime Measurements. We found out that the runtime of PASTA is dominated by the cryptographic algorithms and network performance. Table III depicts runtime measurements of the employed cryptographic algorithms. To attest its integrity, a prover device hashes its firmware and generates a token using the Schnorr multisignature scheme, which consumes around 40.1 ms of runtime in total with a firmware size of 50 kB. The runtime required to verify the integrity of provers listed in a received token depends on various factors. If the token is obtained from an untrustworthy device and the receiver does not store the aggregated public key of all k provers listed in the token, hence, must compute the key ad-hoc, the computation consumes circa $20.95 + 0.11k$ ms. If the receiver uses a stored

Algorithm	Function	Runtime
Schnorr MuSign	GenCommit($r \in \mathbb{Z}_q, R = g^r$)	21.284 ms
	AggCommit($R = R_1 \cdot R_2$)	0.109 ms
	GenChallenge($c = \mathcal{H}(R\|m)$)	3.819 ms
	GenSig($s = r + c \cdot x$)	2.449 ms
	AggSig($s = s_1 + s_2$)	0.006 ms
Schnorr MuVerify	AggKey($X = X_1 \cdot X_2$)	0.109 ms
	Verify($g^s \stackrel{?}{=} RX^c$)	20.896 ms
HMAC-SHA-256	HMAC(16 Bytes)	0.042 ms
	HMAC(1024 Bytes)	0.301 ms
SHA-256	$\mathcal{H}$(51200 Bytes)	13.171 ms

TABLE III: Cryptographic runtime measurements on the ESP32.

aggregated key, the runtime amounts to 20.95 ms. If the token is received from a healthy (i.e., trustworthy) prover, verifying the integrity requires less than 0.06 ms, as only the authenticity of the received message, but not the token signature, needs to be verified. For communication between devices, we used the Wi-Fi communication capabilities of the ESP32-PICO-D4. Unfortunately, our measurements were significantly below the stated theoretical throughput of 150 MBit/s, as we measured an average throughput of 12.51 MBit/s on the application layer using TCP and an average round trip time of 4.63 ms.

Memory Consumption. Each device stores its id (4 B), signature key (32 B), public key of $\mathcal{O}$ (64 B), channel key of each prover ($16k$ B), and public key of each prover ($64k$ B). Each stored token consumes between 69 B and $69 + k/8$ B (worst-case). Assuming a network with 10000 provers, our scheme consumes at most 781.4 kB + $|token|$·1.28 kB. The memory consumption can be reduced by establishing channel keys and public keys on demand, and storing aggregated public keys.

Conclusion. We showed that PASTA imposes a low computational complexity and memory consumption on each device. This makes PASTA practical, even on low-end embedded devices. Compared with SANA [2], the only protocol that allows a scalable attestation of many provers towards untrustworthy devices (but is centralized and only detects software attacks), PASTA requires at least one order of magnitude less computational overhead to generate the attestation result and two orders of magnitude less to verify it.

B. Static Network Simulations

Simulation Setup. To evaluate PASTA in large networks, we performed network simulations with the OMNeT++ [46] event simulator. We implemented PASTA on the application layer and used delays based on our runtime and network measurements. On lower network layers, we applied a simplified communication model that enables devices within direct communication range unimpaired half-duplex communication, as long as no other device within range transmits data.

Token Generation. To examine the scalability of PASTA, we conducted simulations in static networks. In these networks, each device is connected to the overall network topology and

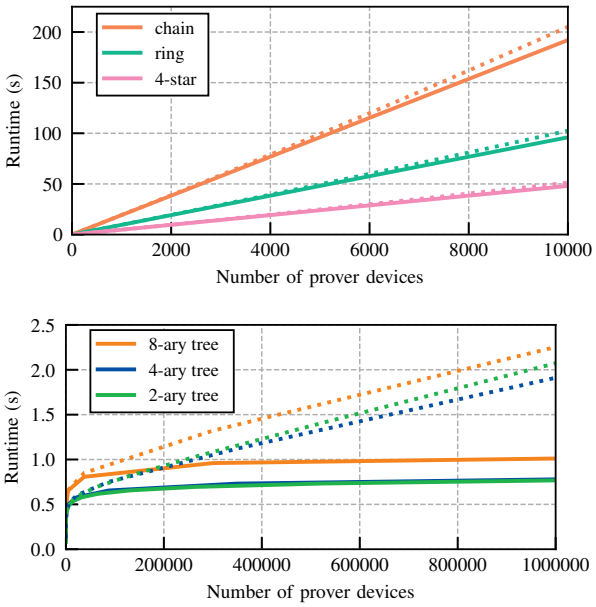
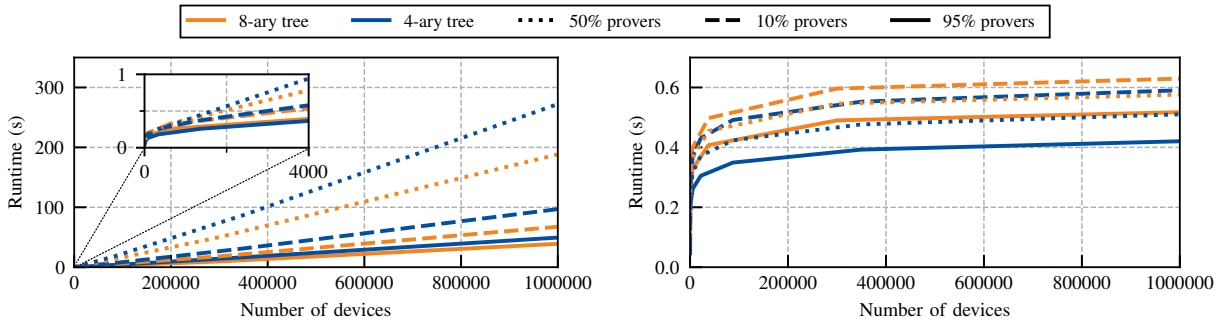


Fig. 3: Runtime of a token generation in various static topologies. Dotted lines represent topology changes between token generations.

connections between devices remain (almost) fixed. Figure 3 shows our simulation results for the runtime of the token generation phase with a varying number of network devices and different network topologies. As shown, the runtime heavily depends on the network topology. Tree topologies enable more provers to perform computations and communication simultaneously in the network. In fact, whereas more than one million prover devices arranged in a tree topology can be attested in less than 1.0 s, the attestation of 10000 provers in a chain topology requires 192.1 s. Fortunately, devices are in typical application scenarios much more likely connected in some form of tree topology than in long chains.

Furthermore, we also investigated the performance in quasi dynamic networks, in which the network topology changes in random ways before the token generation protocol is executed. In completely static networks, the initiator device can assume a static set of participating provers, such that provers do not have to transmit their identifiers during token generation. In quasi dynamic networks, which are represented as dotted lines in Figure 3, this is not possible, since provers may leave or join the network between runs of the token generation protocol. Due to the additional communication of device identifiers, the token generation runtime is slightly higher in quasi dynamic networks and increases with an increasing number of provers.

Token Exchange. Figure 4 illustrates the runtime required to exchange a single token between all devices in the network. The exchanged token was generated by all provers and attests their integrity. We varied the total number of devices, network topology, and ratio between prover and verifier devices. In addition, devices either computed the aggregated public key of provers ad-hoc, depicted in Figure 4a, or used a precomputed



(a) Devices compute the aggregated public key of provers ad-hoc.

(b) Devices use a stored aggregated public key of provers.

Fig. 4: Token exchange runtimes in various static topologies and with a varying ratio between prover and verifier devices.

and stored aggregated public key, depicted in Figure 4b. During token exchange, the aggregated public key is required to verify the token signature, which protects the token integrity. The key must be computed whenever a token is received from an untrustworthy device and the token lists a new set of prover devices for which the receiver does not yet store the aggregated public key. As shown, the difference between computing the key ad-hoc and using a precomputed key is huge. Whereas in the first case the runtime to exchange and verify a token in a network with one million devices amounts to a few minutes, in the latter case it is less than 0.7 s. However, we would like to emphasize that in networks with up to 4000 devices, the token exchange runtime is even in the worst-case, that is, with new sets of provers in each token exchange, always below 1 s.

Another essential factor for the performance of the token exchange is the ratio between provers and verifiers in the network. In case the network contains many provers and only few untrustworthy devices, devices need to verify the signature of received tokens less often and the runtime decreases, as shown by the solid lines in Figure 4a. This is because devices can omit verifying the signature of tokens that are received from healthy provers (IV-C). On the contrary, if there are only few provers and many verifiers, devices need to verify tokens frequently. Nonetheless, in this case, each token contains less prover devices and the aggregated public key required to verify the token can be computed much faster. In total, this also results in a low runtime, as shown by the dashed lines. A balanced number of healthy provers and verifiers results in the highest runtime. Yet, when devices store the precomputed public key required to verify a received token, the impact of the ratio between provers and verifiers is less significant, because verifying tokens is then much faster, as shown in Figure 4b.

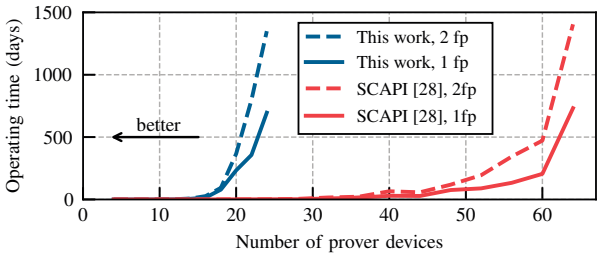
Conclusion. We showed that PASTA is scalable to very large networks due to its small communication and computational overhead. In the best case, which is a static network with a uniform tree topology and only prover devices, one million provers can attest and verify each others' integrity in less than 2.91 s (token generation and exchange). Yet, even in the worst case, which is a network with circa 41 % verifier devices and a topology that changes between runs of the protocol, 1000 devices (410 verifiers and 590 provers) are able to verify the integrity of all 590 provers within 71.7 s. For comparison, SANA [2] would in this case require a runtime of 1421.0 s, when projecting the evaluation results of SANA to our setup.

C. Dynamic Network Simulations

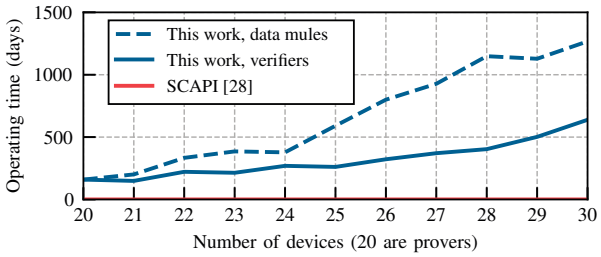
Simulation Setup. We extended our simulation setup for static networks (§ V-B) to simulate dynamic and disruptive network topologies. Instead of using static connections between devices, we set the device communication range to 50 m and deployed devices randomly in a 1000m x 1000m area. During simulation, devices repeatedly select a random destination within the area and then move towards this destination at a specified speed, which is repeatedly set to a random value between 5 and 15 m/s (random waypoint mobility model). PASTA's parameters δ_{gen} and δ_{join} to initiate and join a token generation were set to 10 s and 5 s. In addition, we assumed that $\mathcal{A}dv_{hw}$ requires at least 10 min to physically tamper with a prover, i.e., δ_a = 10 min, as in the evaluation of SCAPI [28].

Robustness of Attestation. To assess the robustness of PASTA, we investigated the runtime until prover devices are mistakenly regarded as physically compromised. In dynamic and disruptive networks, provers only occasionally have a connection to each other. Hence, a healthy prover may be unable to communicate with other provers for longer than δ_a time, whereupon the prover is regarded as physically compromised. Note that the considered scenarios deliberately go beyond typical application scenarios for autonomous embedded systems to determine the boundaries of PASTA.

Figure 5a shows the runtime until either one or two provers are falsely regarded as physically compromised. The network only consists of prover devices, whose number we varied. Furthermore, we set the concurrency factor β, which defines the number of provers $\mathcal{A}dv_{hw}$ can physically compromise within δ_a time, to ∞. Thus, we assumed that $\mathcal{A}dv_{hw}$ is able to physically attack all provers concurrently. In addition, we compared PASTA with SCAPI [28], the most robust existing attestation protocol that detects physical attacks. As shown, the robustness of both protocols exponentially increases with the number of provers in the network. With more provers, the network becomes denser, which allows for more communication



(a) Varying number of false positives (fp).



(b) Single false positive (1 fp) and increasing number of non-prover devices.

Fig. 5: Error-free runtime of prover attestation, i.e., runtime until prover devices are mistakenly regarded as physically compromised.

between the devices. By contrast, in sparser networks, chances are higher that a particular prover does not encounter any other prover within δ_a time, so that all other provers will regard the isolated prover as absent, hence, physically compromised. With the random movement of devices, it is significantly more likely that a single prover is isolated, as opposed to a group of provers. This is why the runtime for a misclassification of two provers is on average almost twice as high as for a single prover. In comparison, SCAPI requires circa three times more provers to achieve the same error-free runtime as PASTA, or, with the same number of provers, operates two orders of magnitude less robustly. This is because in SCAPI each prover must have a connection to other provers at least every $\delta_a/2$ time, as opposed to δ_a time in PASTA. Therefore, PASTA is significantly more robust to network dynamics and disruptions.

In practice, autonomous embedded systems usually not only contain provers, but also (potentially untrustworthy) devices that are not attested. To evaluate SCAPI and PASTA in these scenarios, we set the number of provers to 20 and then added an increasing number of either data mules (dashed lines) or verifiers (solid lines). Whereas verifiers continuously verify the integrity of provers, data mules (e.g., access points) merely store and forward tokens. As depicted in Figure 5b, PASTA runs much more robustly with an increasing number of non-prover devices. This is due to the fact that PASTA enables any device to propagate tokens in the network. By contrast, non-prover devices have no impact on the robustness of SCAPI. In SCAPI, provers communicate encrypted with a secret key that cannot be shared with (potentially untrustworthy) non-prover devices, which are therefore unable to participate in the attestation protocol. Figure 5b also shows that the error-free runtime of PASTA is higher in networks with data mules than with verifiers. The reason for this is that data mules do not verify the integrity of provers, so that less devices in the network can falsely classify a healthy prover as compromised.

In the next simulation, whose results are shown in Figure 6, we arranged an impassable horizontal barrier in the middle of a 800m x 800m area, and deployed half of all devices in each of the two partitions. As a result, connections between devices from the same partition are much more likely to occur than connections between devices from different partitions. In addition, we varied the number of provers and the concurrency factor β. If β is set to ∞, $\mathcal{A}dv_{hw}$ can compromise all provers

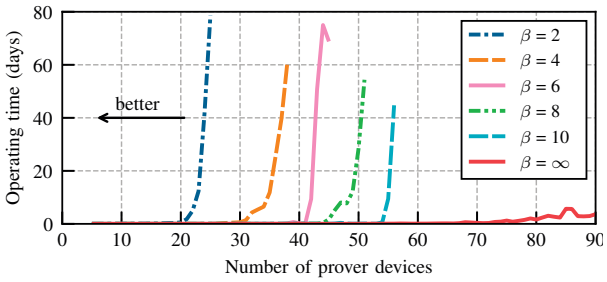


Fig. 6: Error-free runtime of prover attestation with an impassable barrier in the deployment area that separates the network in two parts.

concurrently, as in all previous simulations. With the barrier, it is more likely that whole groups of provers are separated from each other for longer than δ_a time, than single provers. This allows PASTA to make use of its unique feature to reunite separated groups of prover devices. A lower concurrency factor β allows separated groups of provers to be smaller and be separated from other groups for longer times. In fact, setting a slightly lower β in our simulations leads to an increased robustness of up to multiple orders of magnitude. In contrast, SCAPI immediately produces false positives after $\delta_a/2$ time, i.e., 5 min, with up to 90 provers (not shown in Figure 6). This is no surprise, since Figure 5a already showed that SCAPI performs significantly worse than PASTA with $\beta = \infty$.

Conclusion. We demonstrated that PASTA is significantly more robust to network disruptions than SCAPI [28], the most robust existing protocol that detects physical attacks. In the same networks, PASTA has shown to achieve an average error-free operating time that is up to 450 times higher than SCAPI. This huge gap in the robustness even further increases when (i) the network contains additional non-prover devices, e.g., verifier or network infrastructure devices, or (ii) it is assumed that $\mathcal{A}dv_{hw}$ can only physically tamper with a limited number of provers simultaneously. Whereas in the first case, PASTA outperforms SCAPI by three orders of magnitude, the second case enables PASTA to run reliable in network topologies in which an attestation with SCAPI is impossible.

VI. Conclusion & Future Work

In this work, we presented PASTA, an attestation protocol that is particularly suited for autonomous networks of embed-

ded devices. It is the first protocol that (i) allows many provers to attest their integrity towards many verifiers in a scalable and efficient way, (ii) is decentralized, hence, independent of any entity that manages the attestation, and (iii) is able to detect physical attacks in a much more robust way than any existing protocol. In simulations based on real-world measurements, we showed that one million low-end embedded prover devices are able to attest their software and hardware integrity within 0.5 s in a token of only 68 bytes. The token can be verified by a low-end embedded device within 0.06 ms or 21 ms, depending whether the token is received from a prover or a potentially untrustworthy network device. Furthermore, we demonstrated that PASTA is much more robust than any existing protocol that detects physical attacks. In disruptive networks, it achieves an error-free operating time that is several orders of magnitude higher than the best existing protocol.

In future work, we would like to conduct real-world experiments in larger networks to provide a non-synthetic evaluation of PASTA's scalability and robustness. Another interesting direction of research is to further investigate the resistance of embedded systems against physical attacks, with the aim to narrow down the capabilities of a physical adversary.

ACKNOWLEDGMENT

This work has been co-funded by the Federal Ministry of Education and Research of Germany (BMBF) within the UNICARagil project and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

REFERENCES

[1] 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 *ACM CCS*, 2016.

[2] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, "SANA: secure and scalable aggregate network attestation," in *ACM CCS*, 2016.

[3] ARM Holdings, "mbed TLS," https://tls.mbed.org/.

[4] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "Seda: Scalable embedded device attestation," in *ACM CCS*, 2015.

[5] A. Becher, Z. Benenson, and M. Dornseif, "Tampering with motes: Real-world physical attacks on wireless sensor networks," in *SPC*, 2006.

[6] M. Bellare and G. Neven, "Multi-signatures in the plain public-key model and a general forking lemma," in *ACM CCS*, 2006.

[7] D. Boneh, C. Gentry, B. Lynn, and H. Shacham, "Aggregate and verifiably encrypted signatures from bilinear maps," in *EUROCRYPT*, 2003.

[8] X. Carpent, K. ElDefrawy, N. Rattanavipanon, and G. Tsudik, "Lightweight swarm attestation: a tale of two lisa-s," in *ACM ASIACCS*, 2017.

[9] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *ACM CCS*, 2009.

[10] J. Chen, X. Cao, P. Cheng, Y. Xiao, and Y. Sun, "Distributed collaborative control for industrial automation with wireless sensor and actuator networks," *IEEE Transactions on Industrial Electronics*, 2010.

[11] M. Conti, R. Di Pietro, A. Gabrielli, L. V. Mancini, and A. Mei, "The smallville effect: social ties make mobile networks more secure against node capture attack," in *ACM MSWiM*, 2010.

[12] M. Conti, R. Di Pietro, L. V. Mancini, and A. Mei, "Emergent properties: detection of the node-capture attack in mobile wireless sensor networks," in *ACM WiSec*, 2008.

[13] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares," in *USENIX Security*, 2014.

[14] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-overhead control flow attestation in hardware," in *DAC*, 2017.

[15] R. El Bansarkhani and J. Sturm, "An efficient lattice-based multisignature scheme with applications to bitcoins," in *CANS*. Springer, 2016.

[16] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "HYDRA: hybrid design for remote attestation (using a formally verified microkernel)," in *ACM WiSec*, 2017.

[17] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *DATE*, 2014.

[18] W. Fuertes, D. Carrera, C. Villacís, T. Toulkeridis, F. Galárraga, E. Torres, and H. Aules, "Distributed system as internet of things for a new low-cost, air pollution wireless monitoring on real time," in *IEEE/ACM Symposium on Distributed Simulation and Real Time Applications*, 2015.

[19] GitHub: bitcoin-core/secp256k1, "Optimized C library for EC operations on curve secp256k1," https://github.com/bitcoin-core/secp256k1.

[20] L. Hu and D. Evans, "Secure aggregation for wireless networks," in *IEEE SAINT*, 2003.

[21] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "Darpa: Device attestation resilient to physical attacks," in *ACM WiSec*, 2016.

[22] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "Seed: secure non-interactive attestation for embedded devices," in *ACM WiSec*, 2017.

[23] K. Itakura, "A public-key cryptosystem suitable for digital multisignatures," *NEC J. Res. Dev.*, vol. 71, 1983.

[24] R. Jedermann, C. Behrens, D. Westphal, and W. Lang, "Applying autonomous sensor systems in logistics - combining sensor networks, rfids and software agents," *Sensors and Actuators A: Physical*, 2006.

[25] J. Jin, J. Gubbi, S. Marusic, and M. Palaniswami, "An information framework for creating a smart city through internet of things," *IEEE Internet of Things Journal*, 2014.

[26] J. Katz and A. Lindell, "Aggregate message authentication codes," *Topics in Cryptology–CT-RSA 2008*, 2008.

[27] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "Trustlite: A security architecture for tiny embedded devices," in *ACM EuroSys*, 2014.

[28] F. Kohnhäuser, N. Büscher, S. Gabmeyer, and S. Katzenbeisser, "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks," in *ACM WiSec*, 2017.

[29] F. Kohnhäuser, N. Büscher, and S. Katzenbeisser, "Salad: Secure and lightweight attestation of highly dynamic and disruptive networks," in *ACM ASIACCS*, 2018.

[30] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *IEEE S&P*, 2012.

[31] KrebsOnSecurity, "Reaper: Calm Before the IoT Security Storm?" 2017, https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/.

[32] Y. Li, J. M. McCune, and A. Perrig, "Viper: verifying the integrity of peripherals' firmware," in *ACM CCS*, 2011.

[33] S. Lu, R. Ostrovsky, A. Sahai, H. Shacham, and B. Waters, "Sequential Aggregate Signatures and Multisignatures Without Random Oracles," in *EUROCRYPT*, 2006.

[34] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille, "Simple schnorr multi-signatures with applications to bitcoin," in *ePrint Archive*, 2018.

[35] S. Micali, K. Ohta, and L. Reyzin, "Accountable-subgroup multisignatures," in *ACM CCS*, 2001.

[36] T. Morris, "Trusted platform module," in *Encyclopedia of cryptography and security*. Springer, 2011, pp. 1332–1335.

[37] B. Przydatek, D. Song, and A. Perrig, "Sia: Secure information aggregation in sensor networks," in *ACM Sensys*, 2003.

[38] B. Qiao, K. Liu, and C. Guy, "A multi-agent system for building control," in *IEEE/WIC/ACM Conference on Intelligent Agent Technology*, 2006.

[39] S. Ravi, A. Raghunathan, and S. Chakradhar, "Tamper resistance mechanisms for secure embedded systems," in *IEEE VLSID*, 2004.

[40] C.-P. Schnorr, "Efficient signature generation by smart cards," *Journal of cryptology*, vol. 4, no. 3, pp. 161–174, 1991.

[41] S. Schulz, A. Schaller, F. Kohnhäuser, and S. Katzenbeisser, "Boot Attestation: Secure Remote Reporting with Off-The-Shelf IoT Sensors," in *ESORICS*, 2017.

[42] S. Skorobogatov, "Physical attacks and tamper resistance," in *Introduction to Hardware Security and Trust*. Springer, 2012.

[43] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, P. Jovanovic, L. Gasser, N. Gailly, I. Khoffi, and B. Ford, "Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning," in *IEEE S&P*, 2016.

[44] G. Taban and V. Gligor, "Efficient handling of adversary attacks in aggregation applications," in *ESORICS*, 2008.
[45] Texas Instruments, "MSP430x5xx and MSP430x6xx Family – User's Guide Chapter 24.2.4 RTC Protection." 2016.
[46] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *SIMUTools*, 2008.
[47] Y. Yang, X. Wang, S. Zhu, and G. Cao, "Sdap: A secure hop-by-hop data aggregation protocol for sensor networks," *ACM TISSEC*, 2008.
[48] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi, "Atrium: Runtime attestation resilient under memory attacks," in *ICCAD*, 2017.

APPENDIX

According to the adversary model (§ III-C), PASTA is secure, if $\mathcal{A}dv_{sw}$ and $\mathcal{A}dv_{hw}$ are unable to fake a healthy system state for a prover $\mathcal{P}_a$ that is at the time of its own attestation in a compromised software and/or hardware state.

A. Security Analysis for $\mathcal{A}dv_{sw}$

To break PASTA, $\mathcal{A}dv_{sw}$ must compromise the software of a prover $\mathcal{P}_a$ and then generate a token T that makes IsHealthy($\mathcal{P}_a$) return *true* on a healthy device $\mathcal{D}_i$. For this purpose, $\mathcal{A}dv_{sw}$ must interfere with the token exchange and/or the token generation.

Attacks on the Token Exchange. To make IsHealthy($\mathcal{P}_a$) return *true* on $\mathcal{D}_i$, $\mathcal{A}dv_{sw}$ may try to craft a malicious T and then use the token exchange to transfer T to $\mathcal{D}_i$. Recall that each token T stores a token signature $T.sig$, which is computed over the token device identifiers $T.ids$ and the token timestamp $T.ts$. $T.sig$ consists of the aggregated partial signature of each prover that is listed in $T.ids$. The token signature is verified during token exchange and tokens with a wrong signature are discarded. Hence, to make $\mathcal{D}_i$ accept a token T during token exchange, T must contain a valid signature. Furthermore, in order that IsHealthy($\mathcal{P}_a$) returns *true* on $\mathcal{D}_i$, T must list $\mathcal{P}_a$ in $T.ids$ and contain a timestamp $T.ts$ that is more recent than the attack time δ_a (Time() $< T.ts + \delta_a$). To craft such a token T, $\mathcal{A}dv_{sw}$ needs to forge a valid signature for T, since $T.sig$ is dependent on $T.ids$ and $T.ts$. However, because the private token generation key of provers is stored inside their TEE, $\mathcal{A}dv_{sw}$ lacks the signature key to compute a valid signature. Consequently, $\mathcal{A}dv_{sw}$ is unable to forge or manipulate any token that gets accepted by healthy provers.

Attacks on the Token Generation. Since all messages exchanged during token generation are authenticated and verified using the channelkey *ck* of involved provers, $\mathcal{A}dv_{sw}$ requires a valid *ck* to forge or manipulate messages that get accepted by healthy devices. However, as any protocol data is stored inside the TEE of provers and does not leave the TEE, $\mathcal{A}dv_{sw}$ is unable to obtain a valid *ck*. Thus, the only way $\mathcal{A}dv_{sw}$ can interfere with the token generation protocol is by dropping or replaying messages. By dropping messages, $\mathcal{A}dv_{sw}$ can only obstruct provers from participating in the token generation, which does not lead to the generation of a token T that lists $\mathcal{P}_a$ in $T.ids$. Replaying messages is likewise unpromising, due to the following reasons: Healthy provers are unable to generate a valid aggregate signature s_a for $\mathcal{P}_a$ (in msg_{resp}). Therefore, it only makes sense to replay messages from $\mathcal{P}_a$. Yet, s_a is dependent on the timestamp ts, which is unique for each token generation session. Hence, replaying messages from the time when $\mathcal{P}_a$ was in a healthy software state leads to a failed signature check at the initiator, so that no token is generated.

$\mathcal{A}dv_{sw}$ may also attempt to participate in the token generation with a software-compromised prover $\mathcal{P}_a$. However, executing the token generation code, each prover checks its own software integrity using VerifySW() and aborts the token generation protocol in case it is in an untrustworthy software state. Note that this check cannot be circumvented by $\mathcal{A}dv_{sw}$, since the protocol code is executed inside the protected TEE.

B. Security Analysis for $\mathcal{A}dv_{hw}$

Unlike $\mathcal{A}dv_{sw}$, $\mathcal{A}dv_{hw}$ can physically tamper with provers. After physically attacking a prover $\mathcal{P}_a$, $\mathcal{A}dv_{hw}$ has access to secrets stored inside $\mathcal{P}_a$'s TEE and can manipulate the execution of code in $\mathcal{P}_a$'s TEE. A successful physical compromise of $\mathcal{P}_a$ requires $\mathcal{A}dv_{hw}$ to shutdown $\mathcal{P}_a$ for at least the attack time δ_a (§ III-B). Hence, during the physical attack, $\mathcal{P}_a$ misses generating a token for at least δ_a time. Consequently, after the physical attack is completed, all devices in the network only store outdated tokens with $T.ts >$ Time() $- \delta_a$ from $\mathcal{P}_a$ and therefore regard $\mathcal{P}_a$ as physically compromised when executing IsHealthy($\mathcal{P}_a$). To break PASTA, $\mathcal{A}dv_{hw}$ can interfere with the token exchange and/or the token generation.

Attacks on the Token Exchange. Because $\mathcal{A}dv_{hw}$ can access the private token signature key that is stored in the TEE of a physically compromised prover $\mathcal{P}_a$, $\mathcal{A}dv_{hw}$ is, in contrast to $\mathcal{A}dv_{sw}$, able to forge a token T that contains a valid token signature $T.sig$ for $\mathcal{P}_a$, lists $\mathcal{P}_a$ in $T.ids$, and contains a fresh timestamp $T.ts$. Devices that receive such a forged token T during the token synchronization, will accept T due to its valid signature and store T in their token set TS, just like a genuine token. However, during the subsequently executed token validation (§ IV-D), each device that stores T will notice that $\mathcal{P}_a$ has been absent from the network for longer than δ_a time and therefore will regard T as an invalid token. Thus, T is ignored when determining the integrity of $\mathcal{P}_a$ with IsHealthy($\mathcal{P}_a$). To successfully pass the token validation, T needs to contain at least one prover that has not been absent from the network for longer than δ_a time. However, $\mathcal{A}dv_{hw}$ is only able to forge valid signatures for provers that $\mathcal{A}dv_{hw}$ has physically compromised, which all have been absent for longer than δ_a time. Furthermore, $\mathcal{A}dv_{hw}$ is unable to make healthy provers perform the token generation with physically compromised provers, as explained below. For these reasons, $\mathcal{A}dv_{hw}$ is never able to generate a token that passes the token validation, meaning that IsHealthy($\mathcal{P}_a$) will on healthy devices return *false* for any physically compromised prover $\mathcal{P}_a$.

Attacks on the Token Generation. During token generation protocol, each prover checks the integrity of neighboring provers using IsHealthy(). Physically compromised provers fail this check, whereupon they are excluded from the token generation with healthy provers. Thus, $\mathcal{A}dv_{hw}$ has the same capabilities to interfere with the token generation as $\mathcal{A}dv_{sw}$, which does not allow $\mathcal{A}dv_{hw}$ to break the security of PASTA.