SANA: Secure and Scalable Aggregate Network Attestation

Moreno Ambrosin[1], Mauro Conti[1], Ahmad Ibrahim[2]*, Gregory Neven[3], Ahmad-Reza Sadeghi[2], and Matthias Schunter[4]

[1]University of Padua, Italy
[2]Technische Universität Darmstadt, Germany
[3]IBM Research - Zurich, Switzerland
[4]Intel Labs - Darmstadt, Germany

{ahmad.ibrahim, ahmad.sadeghi}@trust.tu-darmstadt.de,
{ambrosin, conti}@math.unipd.it, schunter@acm.org, NEV@zurich.ibm.com

ABSTRACT

Large numbers of smart connected devices, also named as the Internet of Things (IoT), are permeating our environments (homes, factories, cars, and also our body—with wearable devices) to collect data and act on the insight derived. Ensuring software integrity (including OS, apps, and configurations) on such smart devices is then essential to guarantee both privacy and safety. A key mechanism to protect the software integrity of these devices is remote attestation: A process that allows a remote verifier to validate the integrity of the software of a device. This process usually makes use of a signed hash value of the actual device's software, generated by dedicated hardware. While individual device attestation is a well-established technique, to date integrity verification of a very large number of devices remains an open problem, due to scalability issues.

In this paper, we present SANA, the first secure and scalable protocol for efficient attestation of large sets of devices that works under realistic assumptions. SANA relies on a novel signature scheme to allow anyone to publicly verify a collective attestation in constant time and space, for virtually an unlimited number of devices. We substantially improve existing *swarm attestation schemes* [5] by supporting a realistic trust model where: (1) only the targeted devices are required to implement attestation; (2) compromising any device does not harm others; and (3) all aggregators can be untrusted. We implemented SANA and demonstrated its efficiency on tiny sensor devices. Furthermore, we simulated SANA at large scale, to assess its scalability. Our results show that SANA can provide efficient attestation of networks of $1,000,000$ devices, in only 2.5 seconds.

1. INTRODUCTION

Smart devices are rapidly spreading into every domain of our life. These devices range from tiny wearables to larger industrial devices, which could be also integrated among them, e.g., setting up building automation (which involves physical access control, lighting, sheathing, ventilating, and air conditioning). Unlike traditional computing devices, smart devices that are deployed in massive numbers are often limited in cost, computing power, and size. Moreover, embedded devices are often security and privacy critical, since they sense the environment, collect private information, or controls physical equipment, possibly causing damages also in the physical world. Unfortunately, smart devices usually lack the security capabilities of general purpose computers. Indeed, an adversary can easily attack such devices, and compromise both their privacy and safety. One common attack is to modify or replace a device's firmware, as part of a larger attack scenario [2, 1]. In order to prevent such attacks and ensure the safe and secure operation of a device, it is important to guarantee its *software* integrity, e.g., via remote software attestation.

Remote software attestation is an interactive protocol that allows a *prover* to prove its software integrity to a remote *verifier*. The prover demonstrates to the verifier that its software is in a known "good state", which has not been modified. This is usually achieved by signing integrity-protected memory regions. Attestation of individual smart devices is a well established research area. However, to date there is a lack of viable approaches to securely scale device attestation to *a very large number* of devices: Indeed, today's Internet of Things (IoT) infrastructures often rely on a cloud backend to handle *each individual* device. However, this traditional approach has some shortcomings, in particular in terms of communication and computation cost for the cloud infrastructure, which is linear in the number of attested devices. Recently, one proposed approach, SEDA [5], moved a first step towards a more scalable and efficient protocol for attesting a large population of devices. SEDA assumes a *software-only* adversary, i.e., an adversary that can compromise only the software of the target device. It uses (symmetric key based) hop-by-hop attestation within a group of devices, transitively collecting and aggregating attestation responses along an aggregation tree. SEDA merely reports the number of devices that passed attestation, and does not provide additional information about the identity of devices that failed attestation.

Unfortunately, while SEDA substantially increased the scalability of network attestation, it also requires trust in all intermediaries. As a consequence, all devices involved in the attestation protocol are required to: (1) be equipped with a *trusted execution environment*; and (2) participate in the attestation process. These requirements represent a significant limitation when not all the devices in a given area are "trusted" to the same single entity (e.g., to the entity that acts as a verifier of the attestation process). Moreover, in presence of a stronger adversary capable of physical attacks, i.e., capable of tampering with the hardware of even a limited number of devices, SEDA fails to guarantee the security of all other devices.

Contributions. In this paper, we propose SANA, the first attestation scheme for large collections of devices that: (i) is *scalable*, i.e., it efficiently verifies the integrity of a large collection of devices

*Corresponding author.

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978335

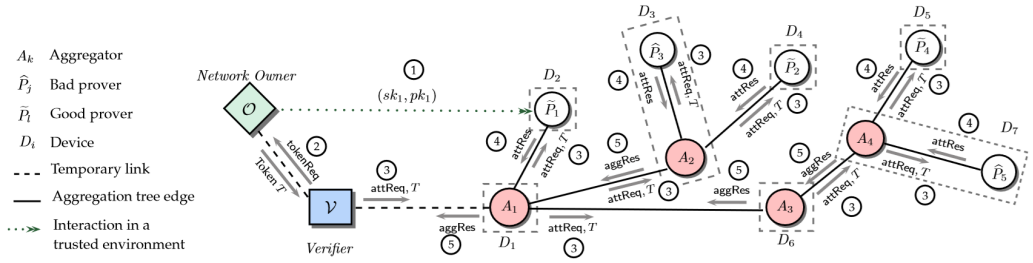


Figure 1: Collective attestation in a network of seven devices (four aggregators and five provers)

by means of a novel signature scheme, which allows aggregation of attestation proofs; (ii) is *publicly verifiable*, i.e., the produced aggregate signature can be verified by any one knowing the (aggregate) public key; and (iii) enables *untrusted aggregation*, i.e., compromise (including physical tampering) of aggregating nodes *does not* affect the integrity of the attestation process. Similar to all other conventional attestation schemes, SANA does not deal with privacy concerns, such as linking the software configuration with a device's identity. Our main focus is providing standard attestation capabilities for large IoT deployments. Required privacy can be provided using secure channels. Our new approach brings the following three technical contributions:

Novel Optimistic Aggregate Signature Scheme. We present a novel signature scheme, called Optimistic Aggregate Signature (OAS). OAS allows the aggregation of signatures on different attestation responses, while having a verification overhead that is *constant in the size of the network*. The idea of combining aggregate and multi-signature is to take the best-of-both-worlds. This has been necessary since none of the existing schemes satisfied the novel requirements we identified for secure collective attestation.

Secure Collective Attestation Scheme. We designed SANA, the first collective attestation scheme for networks of embedded devices that supports high dynamicity and adheres to common assumptions made in single-prover attestation. SANA leverages OAS over aggregation trees to provide highly scalable attestation of large device populations, in a single round-trip. SANA is applicable in settings consistent with large scale IoT device deployments, where aggregator devices can be untrusted routers or cloud servers and is secure in presence of a strong adversary capable of physical attacks.

Evaluation and Performance Analysis. We analyze the performance of SANA on three state-of-the-art security architectures for low-end embedded devices (e.g., SMART [12], TrustLite [17], and TyTAN [11]), and present the simulation results for networks of up to $1,000,000$ devices, in order to demonstrate its scalability.

Outline. We introduce SANA in Section 2 and we present our notation in Section 3. We describe our signature scheme in Section 4, and present the SANA protocol in Section 5. We describe an implementation of SANA in Section 6, and report performance results in Section 7. In Section 8, we describe an extension of SANA. In Section 9 we overview the related work, and the paper concludes in Section 10.

2. SANA

2.1 System Model

We consider large groups of embedded devices, e.g., industrial control systems, IoT devices in smart environments, and prospecting robots. Each group consists of a number of interconnected devices D_i forming a network $\mathcal{G}$, with either static or dynamic topology. $\mathcal{G}$ may not have a routing protocol in place. However, devices in $\mathcal{G}$ are able to identify and communicate to their direct neighbors, which is a minimal assumption in such networks [29].

We formally define SANA as a protocol between the following logical entities: *prover* (P), *aggregator* (A), *owner* ($\mathcal{O}$), and *verifier* ($\mathcal{V}$). As shown in Figure 1, a prover P_i composes a proof of integrity of its software configuration, i.e., an attestation response, to be delivered via aggregators to a remote verifier. In our setting, provers can have different software and hardware configuration. However, we expect the majority of them to have a good software configuration (i.e., the latest non-compromised software version). We refer to this type of provers as *good provers* $\widetilde{P}_i$, while we refer to the rest of the provers as *bad provers* $\widehat{P}_i$ (i.e., devices with malicious or outdated software). An aggregator A_i has the purpose of relaying messages between entities, and collecting and aggregating attestation responses from provers, or other aggregators. The entity $\mathcal{O}$ represents the network owner or operator, responsible for the deployment, as well as the maintenance, of every prover P_i in $\mathcal{G}$. Note that, a physical device in $\mathcal{G}$ can embed the functionalities of every logical component described above, or a combination of them. As an example, in Figure 1 device D_3 acts both as a prover $\widehat{P}_3$, and an aggregator A_2, while D_5 acts only as a prover $\widehat{P}_4$.

The goal of a collective attestation protocol is to assure a verifier $\mathcal{V}$, which is typically, but not necessarily, the same entity as $\mathcal{O}$, of the overall integrity of $\mathcal{G}$, i.e., the software integrity of every prover P_i in $\mathcal{G}$. Provided that none of the provers is physically attacked, $\mathcal{G}$ is considered trustworthy if all the provers in $\mathcal{G}$ are good, i.e., have the latest non-compromised software version approved by $\mathcal{V}$. However, unlike existing attestation schemes that assume a software-only attacker, SANA also considers the presence of physically tampered devices, which may evade their detection. Consequently, we consider $\mathcal{G}$ to be trustworthy, if at least all but physically tampered provers are good. SANA can identify bad devices, as well as their current software configuration.

2.2 Protocol Overview

Figure 1 illustrates the concept of SANA, in a setting where $\mathcal{G}$ is composed of seven physical devices D_1, D_2, D_3, D_4, D_5, D_6, and D_7. SANA is executed between the following (logical) entities: four aggregators, A_1, A_2, A_3, and A_4, five provers, $\widetilde{P}_1$, $\widehat{P}_2$, $\widetilde{P}_3$, $\widetilde{P}_4$, and $\widehat{P}_5$, the owner $\mathcal{O}$, and a verifier $\mathcal{V}$. Each prover P_i is initialized with the cryptographic material needed to execute SANA collective attestation protocol (operation ① shown for $\widetilde{P}_1$ in Figure 1). The initialization is performed in a secure environment, and preferably, but not necessarily, by $\mathcal{O}$. At a given time, a verifier $\mathcal{V}$, which possesses an appropriate attestation token generated by $\mathcal{O}$, may attest $\mathcal{G}$. Note that, if $\mathcal{V}$ and $\mathcal{O}$ are two distinct entities, the token is securely exchanged offline (operation ② in Figure 1).

In order to attest the network, $\mathcal{V}$ chooses an aggregator (randomly or based on physical proximity with the corresponding device; A_1 in Figure 1) and sends it an attestation request containing an attestation

token (operation ③ in Figure 1). The request is flooded in the network forming a logical *aggregation tree*, that has provers as leaf nodes, and aggregators as intermediate nodes.

Leaf nodes of the aggregation tree, i.e., provers $\widetilde{P}_1$, $\widehat{P}_2$, $\widetilde{P}_3$, $\widehat{P}_4$ and $\widetilde{P}_5$, create their attestation response and send it to their parent nodes (operation ④ in Figure 1). Aggregators, i.e., non-leaf nodes (A_1, A_2, A_3 and A_4), in turn, aggregate the attestation responses received from their child nodes (operation ⑤ in Figure 1), and forward the result to their parents. Finally, the aggregated report is received and verified by $\mathcal{V}$.

2.3 Requirements and Assumptions

Objectives. Following the collective attestation literature and the discussion in Section 1, a scalable attestation protocol for collections of devices should, under a strong adversary model, possess the following properties:

- *Unforgeability and Freshness.* If the attestation hardware of a prover is unchanged and a correct verifier was able to validate the aggregate attestation result including a given prover, then the claimed integrity measurement reflects an actual software configuration of this prover at a time during this protocol run.
- *Completeness.* If the attestation hardware of provers is unchanged and a correct verifier was able to validate the aggregate attestation result for a given set of provers, then all provers actually reported their software configuration in the given protocol run.
- *Scalablity.* The protocol allows a verifier to attest a large network of devices. The communication and computational complexity for prover and verifier must be at most logarithmic in the number of devices in the collection.
- *Public Verifiability.* In a public key domain, the collective attestation evidence collected by a verifier can be verified by any party. In this case, the *Unforgeability* requirement only proves the state of the prover within the time window between generation of the challenge by the owner, and the receipt of the evidence from the verifier.
- *Privacy Preservation.* Verification does not require detailed knowledge of the configuration of $\mathcal{G}$ (e.g., its topology).
- *Heterogeneity.* The protocol is applicable to networks with heterogeneous devices. The scheme can use any integrity measurement mechanism used by devices in $\mathcal{G}$.
- *Availability.* If all participants are honest and the network is available then the protocol produces collective attestation evidence.
- *Limiting DoS.* It should not be possible to run a global DoS attack over the whole network through one device.

Unforgeability, *completeness*, and *scalability* are the main objectives of collective attestation. However, a collective attestation scheme should also be *DoS limiting*. These four properties form the crux of SANA's contribution. *Public verifiability* and *privacy preservation* are required in scenarios where network configuration should not be disclosed to the verifying entity (e.g., when the maintenance in a smart factory is outsourced), and can be achieved through digital signatures and aggregation. *Heterogeneity*, on the other hand, is needed to support new device types and future attestation schemes, and is achieved by separating the measurement and the reporting mechanisms used for attestation [5].

Adversary Model. During initial key exchange, we assume the existence of secure channels between any two honest participants. Afterwards, we assume an adversary $\mathcal{A}$ can eavesdrop, insert, or modify all messages exchanged between all devices in $\mathcal{G}$. Furthermore, we assume two types of attacker: (1) a software only attacker, as common in the attestation literature, which can manipulate (i.e., compromise) the software of all provers in $\mathcal{G}$, but not physically attack them; and (2) an attacker capable of physically tampering with aggregator devices, i.e., extract their cryptographic material or modify their software. However, in both cases we assume $\mathcal{A}$ is not capable of forging an Optimistic Aggregate Signature (OAS) according to Definition 3. Finally, while we consider DoS attacks in general to be out of scope, we aim to limit these attacks by preventing $\mathcal{A}$ from running a global DoS on the whole network through one single device.

Security Assumptions. We assume all provers in $\mathcal{G}$ correctly implement the minimal hardware features required for secure remote attestation described in [12], and adapted in [5]. A potential implementation of P_i could have: (1) a Read Only Memory (ROM) that stores the protocol code and the related cryptographic key(s); and (2) a simple Memory Protection Unit (MPU), that restricts access to cryptographic key(s) to protocol code only, and ensures secrecy of the key(s) through non-interruptible, and clean execution of the protocol code. We also assume the owner $\mathcal{O}$ to be trusted. Finally, we assume all cryptographic schemes used in our protocol are secure.

3. PRELIMINARIES AND NOTATIONS

Let $|M|$ denote the number of elements in a finite set M. If n is an integer (or a bit-string), ℓ_n indicates the bit-length of n. Let $m \leftarrow_{\mathrm{R}} M$ denote the assignment of a uniformly sampled element of M to variable m. Furthermore, let $\{0,1\}^{\ell}$ be the set of all bit-strings of length ℓ. If E is some event (e.g., the result of a security experiment), then $\Pr[E]$ denotes the probability that E occurs. Probability $\epsilon(\ell)$ is called *negligible* if, for all polynomials f, $\epsilon(\ell) \leq 1/f(\ell)$ for all sufficiently large $\ell \in \mathbb{N}$.

Let A be a probabilistic algorithm. Then $y \leftarrow \mathsf{A}(x)$ means that on input x, A assigns its output to variable y. We occasionally overload notation to let $\mathsf{A}(x)$ denote the set of all outputs y that have non-zero probability of being returned by A on input x. K is the set of key pairs (pk, sk) that have non-zero probability of being returned by $\mathsf{KeyGen}(\ell_{\mathsf{Sign}})$.

We denote with A^{B} an algorithm A that arbitrarily interacts with algorithm B while it is executing. The term $\mathsf{prot}\,[\mathsf{A} : x_{\mathsf{A}};\ \mathsf{B} : x_{\mathsf{B}};\ * : x_{\mathrm{pub}}] \rightarrow [\mathsf{A} : y_{\mathsf{A}};\ \mathsf{B} : y_{\mathsf{B}}]$ denotes an interactive protocol prot between two probabilistic algorithms A and B. Hereby, A (resp. B) gets private input x_{A} (resp. x_{B}) and public input x_{pub}. While A (resp. B) is operating, it can interact with B (resp. A). As a result, A (resp. B) outputs y_{A} (resp. y_{B}).

A multi-signature scheme [15, 23] allows n different signers to sign the *same* message m in a constant-size signature, i.e., with signature length independent of n. Most multi-signature schemes also have verification time quasi-independent of n, meaning that the number of core cryptographic operations (e.g., exponentiations or pairing computations) is independent of n. An aggregate signature scheme [10] allows n different signers to sign n *different* messages $m_1, \ldots, m_n$ with a constant-size signature, but all known schemes have verification time linear in n.

4. PROPOSED OAS SCHEME

We propose *Optimistic Aggregate Signatures* (OAS) as a generalization of aggregate and multi-signatures, where n different signers can sign different messages $m_1, \ldots, m_n$, but most signers are expected to sign the "default" message M. Individual signatures can be aggregated into an aggregate signature that, in the optimistic case where most signers sign M, is significantly shorter and can be verified significantly faster than n separate signatures. In particular, aggregate signature size and verification time should be independent of the number of signers who signed M. Our construction has aggre-

gate signature size linear in the number of messages different from M and in the number of signers signing those messages. Verification time is linear in the number of different messages that were signed, but independent of the number of signers.

4.1 Definition of an OAS Schemes

An OAS scheme provides both scalability and heterogeneity and is thus applicable for secure collective attestation. Definition 1 provides a formal definition of OAS schemes, while Definition 3 defines the unforgeability property for OAS schemes.

DEFINITION 1 (OPTIMISTIC AGGREGATE SIGNATURES). *An* Optimistic Aggregate Signature (OAS) scheme *is a tuple of probabilistic polynomial time algorithms* $(\mathsf{KeyGen}, \mathsf{AggPK}, \mathsf{Sign}, \mathsf{AggSig}, \mathsf{Verify})$. *On input of the security parameter* $\ell_{\mathsf{Sign}} \in \mathbb{N}$, KeyGen *outputs a secret signing key* sk_i *and a public verification key* pk_i*, for each device* D_i*, i.e.,* $(sk_i, pk_i) \leftarrow \mathsf{KeyGen}(1^{\ell_{\mathsf{Sign}}})$. *On input a set of public keys* $\{pk_1, \ldots, pk_n\}$*, the public key aggregation algorithm* AggPK *outputs an aggregate public key* apk*. On input a secret key* sk*, a message* $m \in \{0,1\}^*$*, and the default message* $M \in \{0,1\}^*$*,* Sign *outputs a signature* α *on* m*, i.e.,* $\alpha \leftarrow \mathsf{Sign}(sk, m, M)$*. Note that, the signature* α *is already considered an aggregate signature, containing only* pk*'s signature on* m*. On input two aggregate signatures* α_1, α_2 *and the default message* M*, the signature aggregation algorithm* AggSig *outputs a new aggregate signature* α *that includes all signatures in* α_1 *and* α_2*. To verify an aggregate signature, the* Verify *algorithm takes an aggregate public key* apk*, a set of public keys* $S_\perp$ *of signers who did not contribute a signature to the aggregate, an aggregate signature* α*, and the default message* M *as input, and outputs either* $\perp$ *to indicate that the signature is invalid, or a set* $\mathcal{B} = \{(m_i, S_i) : i = 1, \ldots, \mu\}$*, indicating that the signers with public key* $pk_i \in S_i$ *signed message* m_i*, and that all other signers whose keys were included in the aggregated public key* apk *but not in any of* $S_\perp, S_1, \ldots, S_\mu$ *signed the default message* M*.*

Intuitively, the correctness of an OAS scheme is defined as follows: if all signers behave honestly and contribute at most one signature to the aggregate, then the verification algorithm produces the output that attributes the correct message to the correct signer. Defining this formally is a bit tedious, however, due to the many different orders in which signatures can be aggregated. To simplify notation, for two sets $\mathcal{B}_1, \mathcal{B}_2$ containing tuples $(m, S) \in \{0,1\}^* \times (\{0,1\}^*)^*$, let $\mathcal{B} = \mathcal{B}_1 \sqcup \mathcal{B}_2$ be the "merged" set of tuples (m, S) where $S = S_1 \cup S_2$ if $\exists (m, S_1) \in \mathcal{B}_1$ and $\exists (m, S_2) \in \mathcal{B}_2$, where $S = S_1$ if $\exists (m, S_1) \in \mathcal{B}_1$ and $\nexists (m, S_2) \in \mathcal{B}_2$, and where $S = S_2$ if $\exists (m, S_2) \in \mathcal{B}_2$ and $\nexists (m, S_1) \in \mathcal{B}_1$.

DEFINITION 2 (CORRECTNESS OF OAS). *An OAS scheme is correct if:*

(i) signing works, i.e., for all $\ell_{\mathsf{Sign}} \in \mathbb{N}$*, all* $m, M \in \{0,1\}^*$*, all* $(pk, sk) \leftarrow \mathsf{KeyGen}(\ell_{\mathsf{Sign}})$*, all sets* $S_\perp$ *such that* $pk \notin S_\perp$*, it holds that* $\mathsf{Verify}(apk, S_\perp, \alpha, M)$ *returns* $\emptyset$ *if* $m = M$ *and returns* $\{m, \{pk\}\}$ *if* $m \neq M$ *whenever* $apk \leftarrow \mathsf{AggPK}(S_\perp \cup \{pk\})$ *and* $\alpha \leftarrow \mathsf{Sign}(sk, m, M)$*.*

(ii) aggregation works, i.e., for all aggregate signatures α_1, α_2*, all disjoint sets* S_1, S_2*, all subsets* $S_{\perp,1} \subseteq S_1$ *and* $S_{\perp,2} \subseteq S_2$*, and all messages* $M \in \{0,1\}^*$*, if* $\mathsf{Verify}(apk_1, S_{\perp,1}, \alpha_1, M) = \mathcal{B}_1$ *and* $\mathsf{Verify}(apk_2, S_{\perp,2}, \alpha_2, M) = \mathcal{B}_2$ *for* $apk_1 \leftarrow \mathsf{AggPK}(S_1)$*,* $apk_2 \leftarrow \mathsf{AggPK}(S_2)$*, and* $apk \leftarrow \mathsf{AggPK}(S_1 \cup S_2)$*, then it holds that* $\mathsf{Verify}(apk, S_{\perp,1} \cup S_{\perp,2}, \alpha, M) = \mathcal{B}_1 \sqcup \mathcal{B}_2$*.*

DEFINITION 3 (UNFORGEABILITY OF OAS). *Unforgeability of an OAS scheme requires that, even if all other signers are dishonest, an adversary cannot produce an aggregate signature that attributes a message to an honest signer that never signed such message. More formally, for any polynomial-time adversary* $\mathcal{A}$*, the following experiment must return 0 with negligible probability:*

$(pk, sk) \leftarrow \mathsf{KeyGen}(\ell_{\mathsf{Sign}})$
$(\alpha, S_\perp, (pk_1, \ldots, pk_n), (sk_1, \ldots, sk_n)) \leftarrow \mathcal{A}^{\mathsf{Sign}(sk,\cdot)}(pk)$
If $\exists\, i : pk_i \neq pk \wedge (pk_i, sk_i) \notin \mathsf{KeyGen}(\ell_{\mathsf{Sign}})$ *then return 0*
Let $S \leftarrow \{pk_1, \ldots, pk_n\}$
$apk \leftarrow \mathsf{AggPK}(S)$
$\mathcal{B} \leftarrow \mathsf{Verify}(apk, S_\perp, \alpha, M)$
If $S_\perp \not\subseteq S$ *or* $\exists (m_i, S_i) \in \mathcal{B} : S_i \not\subseteq S$ *then return 0*
If $\exists (m_i, S_i) \in \mathcal{B} : pk \in S_i$ *and* $m_i \notin Q$ *then return 1*
Let $S_M \leftarrow S \setminus \left(S_\perp \cup \bigcup_{(m_i, S_i) \in \mathcal{B}} S_i\right)$
If $pk \in S_M$ *and* $M \notin Q$ *then return 1*
Else return 0

where Q *is the set of messages that* $\mathcal{A}$ *queried from its* $\mathsf{Sign}(sk, \cdot)$ *oracle.*

The unforgeability notion in Definition 3 requires the adversary to know the secret keys of all corrupt signers, which is modeled in the game by requiring the adversary to output those secret keys as part of his forgery. In practice, this can either be realized by letting a trusted entity generate the keys of all signers, or by letting all signers perform an extractable proof of knowledge of their secret key, either interactively with a trusted entity, or non-interactively and include it in their public keys. Alternatively, Ristenpart and Yilek [28] showed that with minor modifications to some schemes, including Boldyreva's multi-signature scheme [9], it suffices to let signers create a simpler proof of possession, that is essentially a signature on a challenge message. Therefore, being our OAS construction in Section 4.2 based on Boldyreva's multi-signature scheme, this technique can be applied to our scheme as well. Also, note that the above definition insists that the sets of public keys $S_\perp$ and S_i are subsets of $S = \{pk_1, \ldots, pk_n\}$. It is up to the verifier to perform this check, either by looking up the relevant keys in S, or, if the verifier does not know S, by letting signers prove that their keys were included in apk, e.g., through a certificate.

4.2 Our OAS Scheme from pairings

In what follows, we introduce our OAS construction from pairings. Our scheme can be seen as a combination of Boldyreva's multi-signature scheme [9] and Boneh et al.'s aggregate signature scheme [10]. In a multi-signature scheme, all signers sign the same message and the signature can be verified in constant time. In the aggregate signature scheme of Boneh et al., all signers have to sign different messages and verification is linear in the number of aggregated signatures. Our construction essentially uses Boldyreva's multi-signature scheme to compress the signatures of those signers who sign the same message, and uses Boneh et al.'s scheme on top of it to aggregate the multi-signatures. Even though anyone familiar with these schemes immediately sees that the algebra works out, one has to tread very carefully in terms of security. Indeed, aggregate signatures are notorious for having subtle restrictions on key setup, signer composition, and the messages being signed, which, when not adhered to, can ruin the scheme's security [7]. We refer the reader to Appendix B for a formal proof that our construction indeed does satisfy the security notion of Definition 3. We note that Syta et al. [32] already suggested to use Boneh et al.'s aggregate signature scheme in distributed signing applications, but they require all nodes to sign the same message and lacked a detailed security proof, which, given the subtleties mentioned above, is more than just a formality.

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t$ be multiplicative groups of prime order p with generators g_1, g_2, g_t, respectively, with an efficiently computable bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_t$ so that $e(g_1^x, g_2^y) = g_t^{xy}$ for all $x, y \in \mathbb{Z}_p$, and with an efficiently computable isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ so that $\psi(g_2) = g_1$. Let $H : \{0,1\}^* \to \mathbb{G}_1$ be a hash function modeled as a random oracle [8].

Key generation. Each signer chooses random secret key $x \leftarrow_R \mathbb{Z}_p$ and sets its public key to $pk \leftarrow g_2^x$.

Public key aggregation. The aggregate public key for individual public keys $pk_1, \ldots, pk_n$ is $apk = \prod_{i=1}^{n} pk_i$.

Signing. The signature on a message m is $\alpha \leftarrow (H(m)^x, \emptyset)$ if $m = M$ and is $\alpha \leftarrow (H(m)^x, \{(m, \{pk\})\})$ otherwise.

Signature aggregation. Aggregating two aggregate signatures $\alpha_1 = (\tau_1, \mathcal{B}_1)$ and $\alpha_2 = (\tau_2, \mathcal{B}_2)$ can be done by computing $\tau \leftarrow \tau_1 \cdot \tau_2$ and "merging" $\mathcal{B}_1$ and $\mathcal{B}_2$ into $\mathcal{B} \leftarrow \mathcal{B}_1 \sqcup \mathcal{B}_2$. The resulting aggregate is $\alpha = (\tau, \mathcal{B})$.

Verification. To verify an aggregate signature $\alpha = (\tau, \mathcal{B} = \{(m_1, S_1), \ldots, (m_\mu, S_\mu)\})$ under aggregate public key apk, non-contributing public keys $S_\perp$, and default message M, let

$$apk_M \leftarrow \frac{apk}{\prod_{pk \in S_\perp} pk \cdot \prod_{i=1}^{\mu} \prod_{pk \in S_i} pk} . \quad (1)$$

Verify that

$$e(\tau, g_2) = e\big(H(M), apk_M\big) \cdot \prod_{i=1}^{\mu} e\big(H(m_i), \prod_{pk \in S_i} pk\big) . \quad (2)$$

If so, then return $\mathcal{B}$, otherwise return $\perp$.

As mentioned earlier, and as is the case for other multi-signature schemes [9, 20], the signers' keys either have to be generated by a trusted entity, or the signers have to prove possession of their secret keys. For our scheme, the latter is most easily achieved by signing an arbitrary message using a different hash function than for normal signatures [28][1] and adding it to the public key, or by including a Schnorr signature from which the corresponding secret keys can be extracted by applying the generalized forking lemma of Bagherzandi et al. [6].

5. PROTOCOL DESCRIPTION

SANA consists of algorithms and protocols executed by a verifier $\mathcal{V}$, the owner $\mathcal{O}$, and a set of aggregators and provers in the network $\mathcal{G}$. Table 1 provides an overview of the variables and parameters used in the protocol specification.

At its core, SANA distributes a challenge, asks each prover to produce a signed attestation, and aggregates the resulting attestation signatures. Since Denial of Service (DoS) attacks on tiny devices are easy, SANA additionally provides an authorization scheme that allows only authorized verifiers to execute this protocol.

Initialization. Each prover P_i is initialized in a trusted environment by the network owner $\mathcal{O}$ with an OAS key pair ($sk_i \leftarrow_R \mathbb{Z}_p$, $pk_i \leftarrow g_2^{sk_i}$), and an identity certificate $cert(pk_i)$, signed by $\mathcal{O}$, certifying that pk_i is a valid OAS public key of P_i with identity id_i. Formally:

$$\mathsf{init}(1^\ell) \to (sk_i, pk_i, cert(pk_i)) .$$

Token request. In order to attest a network $\mathcal{G}$, a verifier $\mathcal{V}$ must possess a valid authorization token T generated and signed by the owner $\mathcal{O}$ of the network. $\mathcal{V}$ acquires $T = \{\mathcal{H},\ c_l,\ v_l,\ t_{exp},\ \sigma_1\}$ by executing an offline protocol tokenReq (see Figure 2) with $\mathcal{O}$. The main purpose of tokenReq is at the same time mitigating DoS attacks (that are based on the attestation protocol, and can be launched through one single device on the entire network), while allowing attestation service to be public.

The network owner $\mathcal{O}$ keeps a list of counters $c_1, \ldots, c_s$ with values $v_1, \ldots, v_s$. A counter can be assigned by $\mathcal{O}$ to a valid token request until an expiry time t_{exp}, associated to the request, i.e., the counter is marked as "busy" until t_{exp}. After receiving a valid token

[1]The same hash function also works as long as the message space for proofs of possession is separated from that of regular signatures.

Table 1: Variables, parameters and procedures

Entities	
$\mathcal{O}$	Owner or operator of the network
$\mathcal{V}$	Verifier (entity attesting the network)
D_i	Device i
$\widetilde{P}_i$	Good prover i, i.e., a prover with one of the latest non-compromised software configurations
$\widehat{P}_i$	Bad prover i, i.e., a prover with an outdated or malicious software configuration.
A_i	Untrusted aggregator i
Network $\mathcal{G}$ parameters	
a	Total number of aggregators in $\mathcal{G}$
n	Total number of provers in $\mathcal{G}$
g_i	Number of neighbors of A_i
$p_i \leq g_i - 1$	Number of children of A_i in the aggregation tree
Prover P_i parameters	
id_i	ID of P_i
h	Platform software configuration (e.g., hash digest of binary code)
$(sk_i,\ pk_i)$	OAS secret and public key pair of P_i
$sk_{\mathcal{O}}$ (resp. $\mathcal{V}$)	Secret signing key of $\mathcal{O}$ (resp. $\mathcal{V}$) (not based on OAS)
$pk_{\mathcal{O}}$ (resp. $\mathcal{V}$)	Public signature verification key $\mathcal{O}$ (resp. $\mathcal{V}$) (not based on OAS)
$cert(pk_i)$	Identity certificate of P_i (issued by $\mathcal{O}$)
$cert(pk_{\mathcal{O}})$ (resp. $\mathcal{V}$)	Public key certificates for $\mathcal{O}$ (resp. $\mathcal{V}$) (issued by a trusted third party)
$(c_1, v_1) \ldots (c_s, v_s)$	List of attestation counters and corresponding values
SANA parameters	
T	Token used by $\mathcal{V}$ to perform attestation ($T = \{\mathcal{H},\ c_l,\ v_l,\ t_{exp},\ \sigma_1\}$)
N	A random nonce
$\mathcal{H}$	The set of software configurations for the latest software versions of all devices in $\mathcal{G}$
t_{exp}	Expiry time of a token T
δ_t	Expiry period of a token T
Ch	An attestation challenge ($Ch = \{N, T\}$)
apk	Aggregate public key of all provers in $\mathcal{G}$
$\mathcal{S}_i$	Set of public keys grouped by same message signed m_i
$\mathcal{S}_\perp$	Set of public keys that did not participate in generating the OAS signature
m_i	Software configuration on which public keys are grouped
M	The default message signed by OAS
α	An OAS signature
Procedures	
Enc(), Dec()	Public key encryption and decryption
Sign()	Creating a digital (or OAS) signature
Verify()	Verification of a digital (or OAS) signature
checkPolicy()	Application specific procedure that determines whether to accept a token request. Outputs $\delta_t > 0$ if the request is accepted
getFreeCounter()	Searches for an unused counter c_l in $c_1 \ldots c_s$. Sets c_l status to "busy", increments v_l and outputs c_l and v_l
checkCounter()	Checks whether the value of the received counter is greater than the value of the local counter; in this case, sets value of local counter to value of received ones
getSoftConf()	Measures the software configuration
getGoodConfigs()	Retrieves the set $\mathcal{H}$ of software configuration for the latest software versions of all devices in $\mathcal{G}$

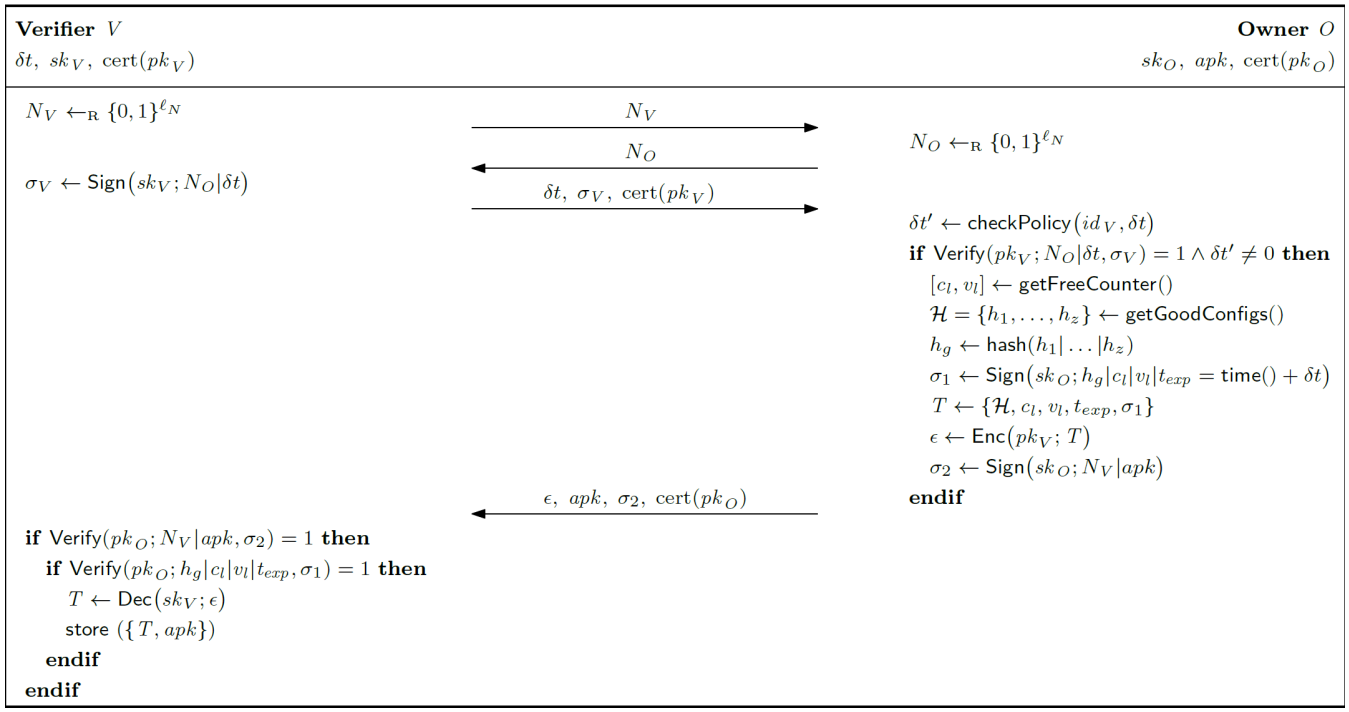


Figure 2: Protocol tokenReq

request from $\mathcal{V}$, $\mathcal{O}$ searches for a free (i.e., not busy) counter c_l, with value v_l, increments v_l by one, and returns the tuple (c_l, v_l) to the requesting verifier – getFreeCounter(). Counters are necessary to protect the network against replay attacks: Indeed, each prover P_i also keeps a list of s counters with corresponding values; once it received an attestation request, P_i checks whether the counter value associated with the request is greater than the value locally stored, and, only in this case, updates its local value and proceeds with the evaluation of the attestation request – checkCounter().

The details of tokenReq are as follows: $\mathcal{V}$ initiates the protocol by sending $\mathcal{O}$ a random challenge $N_\mathcal{V}$, showing its interest in attesting $\mathcal{G}$. Upon receiving $N_\mathcal{V}$, $\mathcal{O}$ creates a random challenge $N_\mathcal{O}$ and sends it to $\mathcal{V}$. $\mathcal{V}$ then creates a signature $\sigma_\mathcal{V}$[2] on $N_\mathcal{O}$ and a protocol parameter δ_t and sends it back to $\mathcal{O}$ along with δ_t, and its identity certificate $cert(pk_\mathcal{V})$. Parameter δ_t indicates the required expiration period of the requested T. Based on $id_\mathcal{V}$ and the requested δ_t, $\mathcal{O}$ decides whether to accept $\mathcal{V}$'s request, according to an application specific policy – checkPolicy(). If the request is accepted and $\sigma_\mathcal{V}$ verified correctly, $\mathcal{O}$ retrieves the set $\mathcal{H} = \{h_1, \ldots, h_z\}$ of software configuration of benign software in $\mathcal{G}$ (i.e., the software configuration of latest software version on different devices in $\mathcal{G}$) – getGoodConfigs(). The list is then hashed into one single good configuration $h_g = \mathsf{hash}(h1| \ldots |h_z)$. Finally, $\mathcal{O}$ sends to $\mathcal{V}$: (1) the aggregate public key apk of all provers in $\mathcal{G}$; (2) A signature σ_2 over apk; and (3) An encrypted [3] token ϵ. Finally, $\mathcal{V}$ verifies σ_2, decrypts and verifies T, and stores it along with apk. Formally:

$$\mathsf{tokenReq}[\mathcal{V} : \delta_t, sk_\mathcal{V}; \mathcal{O} : sk_\mathcal{O}, apk; \\ * : cert(pk_\mathcal{O}), cert(pk_\mathcal{V})] \rightarrow [\mathcal{V} : T, apk; \mathcal{O} : t_{exp}]\,.$$

Attestation: After obtaining an attestation token T, $\mathcal{V}$ can attest the network. Before t_{exp}, $\mathcal{V}$ chooses a random (gateway) aggregator A_1, through which it runs the collective attestation attest of the whole network (see Figure 3). In detail, $\mathcal{V}$ sends A_1 an attestation request $Ch = \{N,\ T\}$ including a random challenge N. Upon receiving this request, A_1 verifies the counter value v_l – checkCounter(), and the signature $\sigma_\mathcal{O}$ using owner's $\mathcal{O}$ public key. We denote this procedure by verifyChallenge(). If the verification succeeds, A_1 forwards the request to its neighbors. Each neighbor, in turn, verifies and forwards the request to its neighbors, and so forth, until the request is received by all provers in the network. Consequently an aggregation tree rooted at A_1 is formed.

As a next step in the protocol, each prover P_i (at the leaf nodes) in the generated aggregation tree, generates its own software configuration h_i – getSoftConf(). If h_i is a benign software configuration (i.e., $h_i \in \mathcal{H}$), D_i creates an Optimistic Aggregate Signature (OAS) α_i over the good software configuration h_g, the challenge N, the counter id c_l, and the counter value v_l (using its OAS secret key sk_i), Otherwise (if $h_i \notin \mathcal{H}$), α_i is created over D_i's software configuration h_i. α_i is then sent to D_i's parent in the aggregation tree. We denote this procedure as createResponse().

Aggregators at intermediate nodes of the tree aggregate responses coming from their children according to the procedure AggSig defined in Definition 1, which we denote as aggregateResponse(), i.e., signatures are aggregated by $M = h_g|N|c_l|v_l$ being the default message. Consequently, attestation responses from provers are propagated, in reverse, along the aggregation tree toward the root A_1. Upon receiving all the responses from its children, node A_1 forwards the final aggregated signature α_1 to $\mathcal{V}$.

Finally, $\mathcal{V}$ verifies α_1 according to Verify in Definition 1. If the verification succeeds and $\mathcal{B} = \phi$, $\mathcal{V}$ concludes that the network is trustworthy. If $\mathcal{B} \neq \phi$, $\mathcal{V}$ learns the identity and the software configuration of all bad devices (i.e., with malicious or outdated software). Formally:

$$\mathsf{attest}_1\,[\mathcal{V} : T, apk; A_1 : -; * : pk_\mathcal{O}] \rightarrow [\mathcal{V} : r; A_1 : Ch]\,.$$
$$\mathsf{attest}_2\,[A_i : Ch; D_j : (sk_j); * : pk_\mathcal{O}] \rightarrow [A_i : \alpha_j; D_j : Ch]\,.$$

The reader may refer to Appendix A for the security analysis of SANA.

[2]Signatures in tokenReq are not based on our OAS scheme, but use an existing public key infrastructure (PKI) between $\mathcal{O}$ and $\mathcal{V}$.
[3]Encryption is based on the public key $pk_\mathcal{V}$ of $\mathcal{V}$.

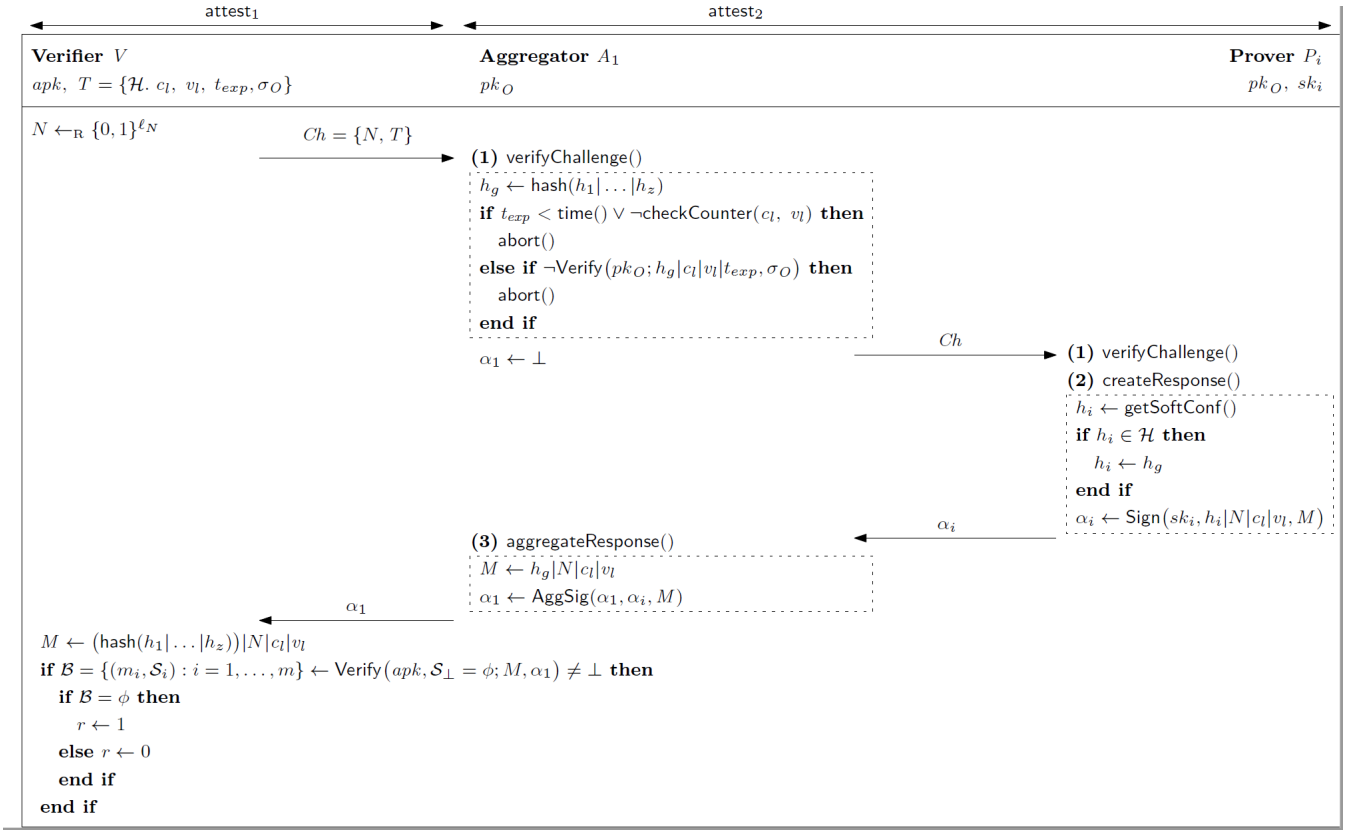


Figure 3: Protocol attest (attest$_1$ **and** attest$_2$)

6. SANA IMPLEMENTATION

We implemented SANA based on three recently proposed security architecture for low end embedded devices: SMART [12], TrustLite [17], and TyTAN [11]. In this section, we discuss our implementation based on TyTAN shown in Figure 4.

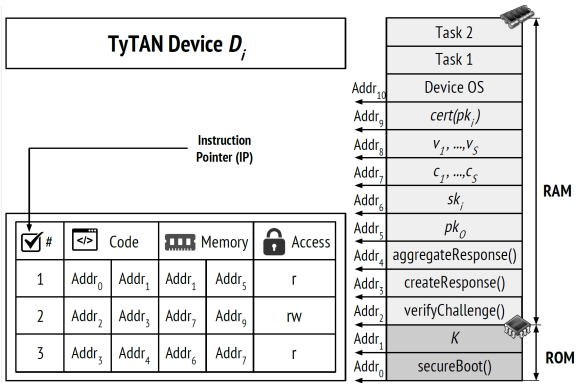


Figure 4: Implementation of SANA based on TyTAN [11]

TyTAN [11] is a security architectures for embedded systems, that is based on TrustLite [17][4]. TyTAN provides hardware-assisted isolation of system components with real-time execution. Isolation is fundamental to protect critical components against unintended access by other potentially malicious components. In TyTAN, a Memory Protection Unit (MPU) restricts access to data, to the task that owns this data. Moreover, both authenticity and confidentiality of the tasks' code and data are based on secure boot. We implemented the components of SANA (i.e., verifyChallenge(), createResponse(), and aggregateResponse()) on TyTAN as isolated tasks, which are protected via secure boot. Further, we configured the MPU such that only SANA's tasks can access the protocols secret data. For example, according to rule #2 in the MPU table in Figure 4, the OAS secret key sk_i (which resides in memory address $Addr_6$ to $Addr_7$) is only read accessible to createResponse() (i.e., code residing in memory address $Addr_3$ to $Addr_4$). Finally, we developed a proof-of-concept implementation of our OAS scheme for both the low-end device in exam (i.e., TyTAN [11]), and for commodity hardware. Our OAS scheme implementation uses the library in [34] for pairing-based cryptographic operations, which we found particularly suitable for our target platforms. OAS operations are defined over the BN254 pairing-friendly elliptic curve [34], which provides a strong security level of 128-bit.

[4]TrustLite is based on Intel's Siskiyou Peak research platform.

7. PERFORMANCE EVALUATION

We now evaluate computational, memory, communication, and energy costs of SANA based on our implementation in Section 6.

Computational cost. The major part of the computational cost on provers and aggregators, is due to the cryptographic operations, i.e., creating and aggregating Optimistic Aggregate Signature (OAS) signatures, and creating the good software configuration h_g. The gateway aggregator A_1, which directly communicates to the verifier $\mathcal{V}$, aggregates at most g_1 signatures, where g_1 is the number of neighbors of A_1 and creates one hash. Every aggregator A_i also creates one hash, and aggregates at most p_i signatures, where $p_i \leq g_i - 1$ and g_i is the number of neighbors of A_i in the network. Finally, each P_i creates one OAS signature and one hash.

Communication cost. Our OAS implementation has a signature size of $\ell_{\mathsf{Sign}} = 256$ bits. We also use $\ell_N = 160$, $\ell_c = 64$, and $\ell'_{\mathsf{Sign}} = 320$. Consequently, counter values are 8 bytes, counter ids are 2 bytes, nonces are 20 bytes, OAS signatures are 32 bytes, public keys are 32 bytes, digital signatures are 40 bytes, and software configurations are 20 bytes. A token T with z good configurations consists of $20z + 58$ bytes, and a challenge Ch of $20z + 78$ bytes. A response α_i has size $32 + 32w + 20\mu$ bytes, where μ is the number of distinct bad software configurations $h_1, \ldots, h_\mu$ and w is the number of distinct OAS public keys of bad provers. The communication overhead of the each aggregator A_i is, sending at most $32 + (20z + 78)g_i + 32w + 20\mu$ bytes and receiving at most $20z + 78 + 32g_i + 32w + 20\mu$ bytes. Finally, every prover P_i sends 84 bytes and receives $20z + 78$ bytes.

Memory cost. Each P_i in $\mathcal{G}$ stores the ids $(c_1 \ldots c_s)$ and values $(v_1 \ldots v_s)$ of s counters, its OAS secret key (sk_i), its identity certificate $(cert(pk_i))$, and the public key $pk_{\mathcal{O}}$ of $\mathcal{O}$. The storage overhead for every P_i is estimated as $10s + 228$ bytes, where s is the number of counters used by $\mathcal{O}$. Low-end embedded devices targeted by SANA (e.g., the TI MSP430) have at least 1024 bytes of non-volatile memory. SANA consumes less than 32% of this memory, assuming that ten verifiers could attest $\mathcal{G}$ within the same time frame.

Run-time. SANA is optimized so that the communication overhead is constant when all provers are correctly configured. On the other hand, the aggregation tree approach allows provers, and aggregators on the same depth of the tree, to perform their computations in parallel. However, the OAS signature aggregation at depth d depends on the signature creation computations at depth $d + 1$. Consequently, the overall run-time of the SANA depends on the depth ($d = f(n + a) \in \mathcal{O}(\log(n + a))$) of the aggregation tree generated for the graph of the network, the number of neighbors of each aggregator, and the number of bad provers. Let t_{sign}, t_{agg}, t_{ver}, t_{hash} and t_{tx} be the times needed to create and aggregate an OAS signatures, verify a digital signature, create the good configuration h_g, and transmit a single byte to a neighbor, respectively. The run-time t of SANA is estimated as:

$$t \leq \big[110d + \sum_{i=0}^{d}(32w_i + 20m_i)\big] \cdot t_{\mathsf{tx}} + \\ + \big(\sum_{i=0}^{d} p_i\big) \cdot t_{\mathsf{agg}} + d \cdot (t_{\mathsf{ver}} + t_{\mathsf{hash}}) + t_{\mathsf{sign}}.$$

Table 2 presents an evaluation of the cryptographic operations required by our OAS, on both TyTAN [11] and a t2.micro Amazon EC2 instance [3][5]. Results are an average over 100 executions. Table 3 shows the estimated execution time for each of the OAS algorithms we presented in Section 4, where p_i is the number of children of a node in the aggregation tree, μ is the number of configurations in $\mathcal{G}$, and n is the number of provers in $\mathcal{G}$.

Energy costs. Let E_{send}, E_{rcv}, E_{sign}, E_{agg}, E_{ver}, and E_{hash} be the energy required to send one byte, receive one byte, create or aggregate OAS signatures, verify a digital signature, and create the good configuration h_g respectively.

Then the energy consumption $E(A_i)$ of each aggregator in $\mathcal{G}$ is:

$$E(A_i) \leq \big(32 + (20z + 78)g_i + 32w + 20m\big) \cdot E_{\mathsf{send}} + \\ + \big(20z + 78 + 32g_i + 32w + 20m\big) \cdot E_{\mathsf{rcv}} + \\ + p_i \cdot E_{\mathsf{agg}} + E_{\mathsf{ver}} + E_{\mathsf{hash}}.$$

And the energy consumption $E(P_i)$ of each prover in $\mathcal{G}$ is:

$$E(P_i) \leq 84 \cdot E_{\mathsf{send}} + (20z + 78) \cdot E_{\mathsf{rcv}} + E_{\mathsf{sign}} + E_{\mathsf{hash}}.$$

[5]Amazon EC2 is running Ubuntu server 14.04 equipped with an Intel Xeon Processor CPU (up to 3.3 GHz), and 1 Gbyte of RAM.

Table 2: Performance of cryptographic operations

Function	TyTAN [11] Run-time (ms)	EC2 t2.micro [3] Run-time (ms)
$H : \{0,1\}^l \rightarrow \mathbb{G}_1$	921.52	3.39
$g^x,\ g \in \mathbb{G}_1$	1282.71	4.71
$g^x,\ g \in \mathbb{G}_2$	(*)	11.60
$ab,\ a, b \in \mathbb{G}_1$	86.48	0.32
$ab,\ a, b \in \mathbb{G}_2$	(*)	0.33
$ab,\ a, b \in \mathbb{G}_T$	(*)	0.07
$e : \mathbb{G}_1 \cdot \mathbb{G}_2 \rightarrow \mathbb{G}_T$	(*)	7.67

(*) Operation not performed by provers or aggregators in SANA

Table 3: Performance of OAS algorithms.

Function	TyTAN [11] Run-time (ms)	EC2 t2.micro [3] Run-time (ms)
Sign	2204.23	8.1
PubKeyGen	(*)	11.60
PubKeyAggr	(*)	$0.33 \cdot n$
SignatureAggr	$86.48 \cdot p_i$	$0.32 \cdot p_i$
AggregateVerify	(*)	$0.33 \cdot \sum_2^\mu (\lvert S_i \rvert - 1) + (8.16) \cdot \mu$

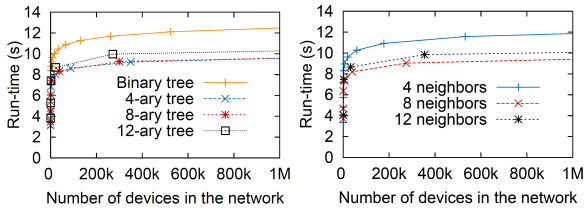
(*) Operation not performed by provers or aggregators in SANA

Simulation results. We simulated SANA for large networks using the OMNeT++ [25] event simulator. We implemented our protocol at the application layer and used delays, based on measurements for TyTAN (Table 2), to simulate the different cryptographic operations. We set the communications rate for links between two devices to 250 Kbps, which is the defined data rate of ZigBee – a common communication protocol for IoT devices. We simulated different network topologies including trees (with fan-out degree 2, 4, 8, and 12), and networks with fixed number of neighbors (4, 8 and 12). We varied the size of the network from ten to $1,000,000$ devices. For a fair comparison with SEDA [5], we carried out our simulations assuming all devices in the network to be low-ended devices that needs to be attested. Figure 5 and Figure 6 show the results of our simulations. To better assess the performance of SANA, we also simulated it in its targeted setting, where untrusted aggregator devices are more powerful, i.e., 50% Raspberry Pi devices[6], 30% Intel Galileo devices[7], and 20% t2.micro EC2 instances [3], and the communication rate is 5 Mbps. We also simulated the extension of SEDA, described in [5], that is capable of identifying malicious devices. We denote this extension by SEDA-ID. From our results, we can see that, if the number of bad provers is fixed, then the run-time of SANA, for a tree topologies (Figure 5(a)) and for networks with fixed number of neighbors (Figure 5(b)), is logarithmic in the size of the network.

As shown in Figure 6(a), SEDA shows better performance compared to SANA. However, while SEDA imposes a strong requirement on the devices participating in the attestation protocol, which are low-end devices equipped with trusted hardware, SANA gives a significantly improved flexibility in the type of devices that can act as aggregators, and resiliency to a stronger attacker model. In fact, SANA does not impose any constraint or requirement on the devices acting as aggregators, which can be completely untrusted. This better serves typical practical deployments, where data ag-

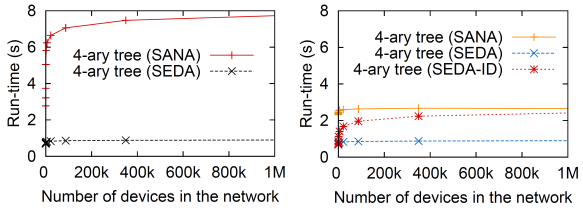
[6]Raspberry Pi has a 700 MHz CPU and 512 MByte of RAM

[7]Intel Galileo has a 400MHz CPU and 256 MByte of RAM



(a) Tree topologies (b) Fixed number of neighbors

Figure 5: Performance evaluation of SANA



(a) Low-end only devices (b) Realistic setting for SANA

Figure 6: Performance comparison between SANA and SEDA

gregation is performed by more powerful but untrusted devices in the network, such as routers or cloud servers. Figure 6(b) further stresses the advantage of SANA in more realistic scenarios, showing its run-time adopting more powerful devices as aggregators. As can be seen from Figure 6(b), the difference in the run-time of the two schemes in such scenario, can become as little as 1.5 seconds. Finally, while SEDA merely reports the number of devices that failed attestation, SANA enables the verifier to identify bad provers as well as their software configuration. Figure 6(b) shows the performance of SEDA when modified to report the ids of devices that failed attestation as suggested in the original paper [5]. Our performance analysis demonstrates that, in its targeted setting, SANA can perform as good as SEDA, regardless of the digital signature, and DoS mitigation included. Note that, digital signatures may leverage hardware acceleration (as already done for base symmetric crypto operations) leading to a better performance than SEDA.

8. THRESHOLD ATTESTATION

Although constant in the size of the network, the overhead of SANA is linear in the number of bad provers (i.e., malicious devices). However, as described in Section 1, we aim at providing a constant-time network attestation protocol, through which a resource constrained verifier $\mathcal{V}$ (e.g., a smartphone) can verify a very large (e.g., in order of millions) network of devices. In this section we briefly discuss a possible extension of SANA that allows such a constant-time verification.

We base this extension on the following observation: While in some applications the number of malicious devices might be linear in the size of the network (i.e., a certain percentage), typically the maximum number of accepted compromise is fixed. We believe that this assumption is reasonable, since the number of devices to tolerate is related to the redundancy rather than to the size of the network. Consequently, we can set a threshold (i.e., an upper bound) for the number of bad devices to verify. The threshold is set by $\mathcal{V}$, and embedded inside the token T. During attestation responses aggregation, aggregators keep aggregating signatures on bad software configurations only until the threshold is reached. Every signature received afterwards is simply dropped. Consequently, since dropping signatures renders the final report unverifiable, if report verification fails, $\mathcal{V}$ deduces that the number of bad devices has exceeded the threshold.

As shown in Figure 7, threshold attestation run in constant- time. However, besides reaching the upper bound in the number of allowed bad configurations, other factors, such as benign errors and active (DoS) attacks, may cause a failure in the verification of the aggregate attestation response. For this reason, it is important to verify that this result is indeed due to the existence of *too many* bad devices. In order to tackle this problem, a potential solution is to split the OAS signature into two parts: An aggregate signature (over bad configurations), and a multi-signature (over the good configuration). In this way, verifying the aggregate signature (in constant time), gives $\mathcal{V}$ an assurance that the number of devices with bad configuration has exceeded the threshold. Additionally, $\mathcal{V}$ learns identities and software configurations of such bad devices.

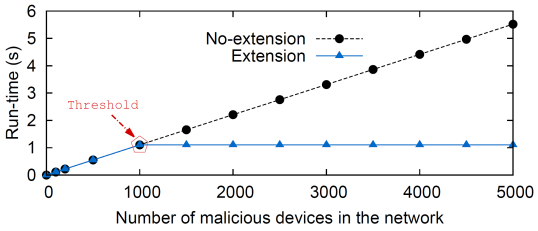


Figure 7: Run-time of SANA on an EC2 t2.micro [3] verifier

9. BACKGROUND AND RELATED WORK

Individual Device Attestation is a well-established research area. The main goal of an attestation process is to allow a verifier to check the integrity of the software running on a (potentially remote) device (prover). We can distinguish three main approaches of attestation: software-based, co-processor-based, and hybrid. Software-based attestation [16, 31, 14, 19] requires no secure hardware and does not rely on cryptographic secrets, making it particularly attractive for low-end devices with limited resources. Unfortunately, the security of software-based attestation has been challenged [35], since it is based on strong assumptions that are hard to achieve in practice [4]. As an example, software-based attestation assumes that the attestation algorithm and its implementation are optimal, and that the adversary is passive during the whole execution of the attestation protocol. Moreover, software-based attestation relies on strict estimation of round trip times, requires an out-of-band authentication channel, and is thus restricted to one hop communication, and not suitable for remote attestation. Co-processor based attestation schemes [27, 33, 18, 30, 22, 21], on the other hand, offer improved security guarantees. However, they are more suitable for general-purpose computing platforms, since their underlying security hardware is often too complex and/or expensive for low-end embedded devices. A third recently presented approach for attestation is based on a hardware/software co-design [12, 17, 13, 11], and aims for minimizing the hardware security features required for enabling secure remote attestation. Such security features can be as simple as a Read Only Memory (ROM), and a simple Memory Protection Unit (MPU). SANA requires the same minimal hardware support to implement collective attestation on the attested devices.

Collective Attestation. SEDA, the solution recently proposed in [5], made a first step towards a *collective attestation*, i.e., the scalable attestation of large groups of interconnected devices. The main focus of SEDA is efficiency and applicability to low-end devices, rather than security in the presence of a strong adversary. To that end, SEDA extends the software-only attacker assumed by most

single-prover attestation schemes to, so-called, swarms of devices. With this in mind, security hardware, used for protecting attestation code, is exploited to enable: (1) neighbors' verification, which decreases the load on the verifier; and (2) secure hop-by-hop aggregation, which reduces the communication overhead. SEDA achieves high efficiency and scalability through the distribution of the attestation burden across the whole network. It merely reports the number of devices in the network that passed attestation. SEDA requires (minimal) trust anchor in hardware for all involved nodes [5], and therefore, it cannot operate in presence of a stronger attacker, i.e., an attacker capable of physically tampering devices. Indeed, if an attacker violates the hardware security of one node, it may break the overall security of the scheme for all other devices.

Our proposed collective attestation protocol SANA overcomes the limitations of SEDA by: (i) requiring minimal trust anchor in hardware only for the attested devices, (ii) allowing aggregation to be performed by largely untrusted nodes, which are only required for availability, and (iii) limiting the effect of successful attacks on the hardware of an attested device to the device itself, i.e., it will not affect the attestation of other devices. Similarly, Denial-of-Service attacks on one device in SANA will not affect other devices. Finally, SANA informs the verifier with ids as well as software configurations of the devices that failed attestation.

In Network Aggregation. To meet the aforementioned goals, SANA employs in network aggregation. Several secure in network aggregation protocols have been proposed in the sensor network area, and in wireless sensor networks in particular, to provide scalable data collection from sensors [26]. In general, such aggregation schemes allow a collection of sensors to collaboratively and securely compute arbitrary aggregation functions on collected data, to reduce message complexity. However, all these protocols have either a verification complexity which is linear in the number of nodes in the network, or are built in multiple protocol rounds.

Aggregate Multi-Signatures. SANA requires a scalable aggregatable signature scheme, to allow attested devices in the network to sign their state. Such scheme allows different signers with different public keys to sign distinct messages. Moreover, it allows intermediate nodes to securely aggregate individual signatures into a single verifiable short signature. Unfortunately, all known aggregate signature schemes have a verification overhead that is linear in the number of messages and signers, which renders them unsuitable for scalable attestation. Additionally, they either require all messages to be distinct, or a complex key agreement protocol [24], require a sequential order [9, 20], or all the signatures to be created on the same message [6]. As a consequence, we consider none of the known scheme as suitable for our collective attestation scheme. Recently, the work in [32] proposed CoSi, a scalable witness cosigning system for certification, logging, and timestamping authorities that combines existing multi-signature schemes with communication trees. However, the protocol proposed in [32] considers static communication trees, constitutes multiple round-trips, and has communication and computational overhead that are also linear in the number of signers. Moreover, CoSi does not allow signing different messages, and is thus not applicable to large scale network attestation.

For the reasons above, in this paper we present a new signature scheme, Optimistic Aggregate Signature (OAS), that: (1) allows signatures on distinct messages to be aggregated; and (2) provides a signature verification algorithm that is constant in the number of signers. The communication overhead of the scheme is linear in the number of different messages, while the computational overhead is linear in the number signers who signed a different message than the default one. However, this number is assumed limited. To the best of our knowledge, our proposed OAS is the first scheme that satisfies the requirements of a secure collective attestation. Finally, we present a pairings-based construction of OAS, and combine it with aggregation trees, providing unlimited scalability. Our OAS construction is also applicable to witness cosigning at certification, logging, and timestamping authorities [32].

10. CONCLUSIONS

Collective attestation is a key building block for securing the Internet of Things. For very large numbers of devices, to enable enterprises to validate the configuration and software and ensure that all devices are indeed up-to-date. In this paper, we have proposed the first practical and secure collective attestation scheme SANA. It substantially improves the state of the art (e.g. SEDA [5]) in three aspects: (1) it is easy to deploy since it can use any untrusted aggregator; (2) its output is publicly verifiable since a short aggregate attestation can be publicly verified by anyone; (3) it provides superior security since it ensures that if a device is fully compromised (including its hardware and keys), then other devices are not affected at all; and (4) it allows a realistic trust model, where only the attested devices are required to be trusted. We have demonstrated that the protocol is truly scalable and can be implemented on lightweight devices.

Acknowledgements

We thank anonymous reviewers for their useful comments. This research was co-funded by the German Science Foundation, as part of project S2 within CRC 1119 CROSSING, EC-SPRIDE, the European Union's 7^{th} Framework Programme, under grant agreement No. 609611, PRACTICE project, and Intel Collaborative Research Institute for Secure Computing (ICRI-SC). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), and the EU-India REACH Project (agreement ICI+/2014/342-896). Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980).

11. REFERENCES

[1] Target attack shows danger of remotely accessible HVAC systems. http://www.computerworld.com/article/2487452/cybercrime-hacking/target-attack-shows-danger-of-remotely-accessible-hvac-systems.html, 2014.
[2] Jeep Hacking 101. http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101, 2015.
[3] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, 2016.
[4] F. Armknecht et al. A security framework for the analysis and design of software attestation. In *ACM CCS'13*.
[5] N. Asokan et al. SEDA: Scalable Embedded Device Attestation. In *ACM CCS'15*.
[6] A. Bagherzandi et al. Multisignatures secure under the discrete logarithm assumption and a generalized forking lemma. In *ACM CCS'08*.
[7] M. Bellare, C. Namprempre, and G. Neven. Unrestricted aggregate signatures. In *ICALP '07*.
[8] M. Bellare et al. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS '93*.
[9] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC '03*.
[10] D. Boneh et al. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT'03*.
[11] F. Brasser et al. Tytan: Tiny trust anchor for tiny devices. In *ACM/EDAC/IEEE DAC'15*.
[12] K. Eldefrawy et al. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *NDSS'12*.

[13] A. Francillon et al. A minimalist approach to remote attestation. In *DATE'14*.
[14] R. Gardner et al. Detecting code alteration by creating a temporary memory bottleneck. *IEEE Transactions on Information Forensics and Security*, 2009.
[15] K. Itakura et al. A public-key cryptosystem suitable for digital multisignatures. *NEC Research & Development*, 1983.
[16] R. Kennell et al. Establishing the genuinity of remote computer systems. In *USENIXSec'03*.
[17] P. Koeberl et al. TrustLite: A security architecture for tiny embedded devices. In *ACM EuroSys'14*.
[18] X. Kovah et al. New results for timing-based attestation. In *IEEE S&P'12*.
[19] Y. Li et al. VIPER: Verifying the integrity of peripherals' firmware. In *ACM CCS'11*.
[20] S. Lu et al. Sequential aggregate signatures, multisignatures, and verifiably encrypted signatures without random oracles. *Journal of Cryptology*, 2012.
[21] J. McCune et al. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P '10*.
[22] J. McCune et al. Flicker: An execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 2008.
[23] S. Micali et al. Accountable-subgroup multisignatures: extended abstract. In *ACM CCS'01*.
[24] S. Micali et al. Accountable-subgroup multisignatures: Extended abstract. In *ACM CCS'01*.
[25] OpenSim Ltd. OMNeT++ discrete event simulator. http://omnetpp.org/, 2015.
[26] S. Ozdemir and Y. Xiao. Secure data aggregation in wireless sensor networks: A comprehensive overview. *Computer Networks*, 2009.
[27] J. Petroni et al. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIXSec'04*.
[28] T. Ristenpart et al. The power of proofs-of-possession: Securing multiparty signatures against rogue-key attacks. In *EUROCRYPT'07*.
[29] M. Rubenstein et al. Programmable self-assembly in a thousand-robot swarm. *Science*, 2014.
[30] D. Schellekens et al. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 2008.
[31] A. Seshadri et al. SAKE: Software attestation for key establishment in sensor networks. In *Ad Hoc Networks*. 2011.
[32] E. Syta et al. Keeping authorities"honest or bust"with decentralized witness cosigning. In *IEEE S&P '15*.
[33] Trusted Computing Group (TCG). Website. http://www.trustedcomputinggroup.org, 2015.
[34] T. Unterluggauer et al. Efficient pairings and ECC for embedded systems. In *CHES '14*.
[35] G. Wurster et al. A generic attack on checksumming-based software tamper resistance. In *IEEE S&P '05*.

APPENDIX

A. SECURITY OF SANA

The security goal of any collective attestation scheme is to ensure a verifier $\mathcal{V}$ that all the devices in a network are running a software that is known to, and accepted by $\mathcal{V}$. In other words, $\mathcal{V}$ should accept the attestation result, and declare the network $\mathcal{G}$ as benign, if and only if all devices in $\mathcal{G}$ (at least those that are not physically attacked) are running a software it agrees on. We formalize this goal as a security experiment $\mathbf{Exp}_{\mathcal{A}}$ between an adversary $\mathcal{A}$ (as defined in Section 2.3), $\mathcal{G}$ and $\mathcal{V}$. In this experiment, $\mathcal{A}$ physically attacks a number of provers $P_1, \dots P_p$, and modifies the software of at least one prover P_c that it did not physically attack. $\mathcal{A}$ may have full control of all aggregators. After a polynomial number (in terms of the security parameters ℓ_N, ℓ_c, and ℓ_{Sign}) of steps by $\mathcal{A}$, $\mathcal{V}$ outputs its decision $r = 1$ indicating it accepts the attestation result, or $r = 0$ otherwise. The result of the experiment is defined as the output of $\mathcal{V}$, i.e., $\mathbf{Exp}_{\mathcal{A}} = r$. A secure collective attestation scheme is defined as follows:

DEFINITION 4 (SECURE COLLECTIVE ATTESTATION). *A collective attestation scheme is secure if* $\Pr\left[r = 1 | \boldsymbol{Exp}_{\mathcal{A}}(1^{\ell}) = r\right]$ *is negligible in* $\ell = f(\ell_N, \ell_c, \ell_{\mathsf{Sign}})$*, where the function* f *is polynomial in* ℓ_N*,* ℓ_c*, and* ℓ_{Sign}*.*

THEOREM 1 (SECURITY OF SANA). *SANA is a secure collective attestation scheme (Definition 4) if the underlying Aggregate-Multi Signature (OAS) scheme is not forgeable (Definition 3).*

PROOF (SKETCH) OF THEOREM 1. The verifier $\mathcal{V}$ returns $r = 1$, i.e., accepts an attestation response α_1 only if $\mathcal{B} = \{(m_1, S_1), \dots, (m_\mu, S_\mu)\} = \mathsf{Verify}(apk, S_\perp, M, \alpha_1)$, where apk is the OAS aggregate public key of all provers in $\mathcal{G}$, $S_\perp = \phi$, $\mathcal{B} = \phi$, and $M = \big(\mathsf{hash}(h_1|\dots|h_z)\big)|N|c_l|v_l$. N is the nonce, $h_1, \dots, h_z$ are all the benign software configurations included in $\mathcal{H}$, and c_l (resp. v_l) is the id (resp. value) of the counter previously sent by $\mathcal{V}$. To avoid detection of a compromised prover P_c, $\mathcal{A}$ can use one of the following strategies: (1) $\mathcal{A}$ does not alter the response from P_c; (2) $\mathcal{A}$ uses an old signature α_{old} previously generated by P_c on the good software configuration h_g; (3) $\mathcal{A}$ modifies $\mathcal{H}$ to include the software configuration of P_c; (4) $\mathcal{A}$ generates an aggregate signature that attributes a response $M = h_g|N|c_l|v_l$ to P_c; or (5) $\mathcal{A}$ modifies the aggregate public keys apk sent from the owner $\mathcal{O}$ to $\mathcal{V}$ during $\mathsf{tokenReq}$ protocol.

We start with strategy (1). According to our assumptions in Section 2, $\mathcal{A}$ can neither tamper with the code performing integrity measurements on P_c nor extract its secret signing key sk_c. Consequently, the response of P_c will include an OAS signature on $h_c|N|c_l|v_l$, where h_c is the current software configuration of P_c. Since h_c is the security configuration of a maliciously modified software, $\mathcal{V}$ will always return $r = 0$, if the underlying integrity measurement mechanism that generates the software configuration can detect this modification (i.e., $h_c \notin \mathcal{H}$). On the other hand, if $\mathcal{A}$ uses an old signature α_{old} over $M_{old} = h_g|N_{old}|c_l|v_l$ (Strategy (2)), $\mathcal{B}$ will be equal to ϕ only if $N = N_{old}$, which is negligible in ℓ_N, where ℓ_N represents the size of the nonce. Consequently, $\mathcal{V}$ will always return $r = 0$.

Next, we consider strategy (3), where the adversary modifies $\mathcal{H}$ to include the software configuration h_c of P_c. Since $\mathcal{H}$ is part of the authenticated token T, $\mathcal{A}$ should forge the signature σ_1 generated by the owner $\mathcal{O}$. The success probability of this strategy is negligible in ℓ'_{Sign}, where ℓ'_{Sign} represents the security parameter of the digital signature scheme used. Note that, $\mathcal{H}$ is also included in α_c. Therefore, any modification to $\mathcal{H}$ consequently changes α_c.

According to strategy (4), $\mathcal{A}$ may generate an aggregate signature that attributes $m = h_g|N|c_l|v_l$ to P_c, where N is the fresh nonce sent by $\mathcal{V}$. Since the current software of P_c is not in $\mathcal{H}$, and since N is a fresh nonce, the probability of P_c signing $h_g|N|c_l|v_l$ is negligible in ℓ_N. Consequently, as OAS is unforgeable accorfing to Definition 3, the probability of finding such an aggregate is negligible in ℓ_{Sign}, where ℓ_{Sign} represents OAS security parameter.

Finally, we consider $\mathcal{A}$ modifying the aggregate public key apk, and replacing pk_c with $pk_{\mathcal{A}}$, to which it knows the secret key $sk_{\mathcal{A}}$. $\mathcal{A}$ can at this point sign arbitrary messages (including attestation responses) on behalf of P_c, regardless of P_c's current software configuration. However, since the integrity of apk is protected with a digital signature, the probability of success of this attack is negligible in ℓ_N and ℓ'_{Sign}.

Consequently, the probability of $\mathcal{A}$ making $\mathcal{V}$ return $r = 1$, when it has maliciously modified the software of one prover P_c in $\mathcal{G}$

(that it did not physically attacked), is negligible in ℓ_N, ℓ_{Sign}, and ℓ'_{Sign}. Similarly, if none of the provers is physically attacked, the probability of $\mathcal{A}$ making $\mathcal{V}$ return $r = 1$, when at least one device has a malicious software is negligible in ℓ_N, ℓ_{Sign}, and ℓ'_{Sign}. □

B. CORRECTNESS AND SECURITY OAS

Correctness. If $sk = x$ and all parties behave honestly, then we have that $pk = g_2^x$ and $apk = \mathsf{AggPK}(S_\perp \cup \{pk\}) = g_2^x \prod_{pk' \in S_\perp} pk'$. If $m = M$, then $\alpha = \mathsf{Sign}(sk, m, M) = (\tau, \mathcal{B}) = (H(M)^x, \emptyset)$ so that during verification we have that $\mu = 0$, $apk_M = apk / \prod_{pk' \in S_\perp} pk' = g_2^x$, and $e(\tau, g_2) = e(H(M)^x, g_2) = e(H(M), apk_M)$ as required. If $m \neq M$, then $\alpha = \mathsf{Sign}(sk, m, M) = (\tau, \mathcal{B}) = (H(m)^x, \{(m, \{pk\})\}$ so that during verification we have that $\mu = 1$, $apk_1 = pk = g_2^x$, $apk_M = apk/(pk \cdot \prod_{pk' in S_\perp} pk') = 1$, and, since $e(H(M), 1) = 1$, that $e(\tau, g_2) = e(H(m)^x, g_2) = e(H(m), apk_1)$ as required.

To see that aggregation works, if $\mathsf{Verify}(\mathsf{AggPK}(S_j), S_{\perp,j}, \alpha_j, M) = \mathcal{B}_j$ for $j = 1, 2$, by the verification equation (2) it holds that

$$e(\tau_j, g_2) \;=\; e\big(H(M), apk_{M,j}\big) \cdot \prod_{i=1}^{\mu_j} e\big(H(m_{i,j}), apk_{i,j}\big)\,,$$

where $apk_{i,j} = \prod_{pk \in S_{i,j}} pk$ and $apk_{M,j} = \prod_{pk \in S_{M,j}} pk$ for $S_{M,j} = S_j \setminus S_{\perp,j} \setminus \bigcup_{(m_{i,j}, S_{i,j}) \in \mathcal{B}_j} S_{i,j}$. Multiplying the verification equations for $j = 1, 2$ gives

$$\begin{aligned} e(\tau_1\tau_2, g_2) &= e\big(H(M), apk_{M,1}\, apk_{M,2}\big) \\ &\quad \cdot \prod_{i=1}^{\mu_1} e\big(H(m_{i,1}), apk_{i,1}\big) \cdot \prod_{i=1}^{\mu_2} e\big(H(m_{i,2}), apk_{i,2}\big) \\ &= e\big(H(M), apk_M\big) \cdot \prod_{(m_i, S_i) \in \mathcal{B}} e\big(H(m_i), apk_i\big)\,. \end{aligned}$$

where $apk_M = \prod_{pk \in S_{M,1} \cup S_{M,2}} pk$, $apk_i = \prod_{pk \in S_i} pk$ and $\mathcal{B} = \mathcal{B}_1 \sqcup \mathcal{B}_2$. The second equality follows from the disjointness of S_1 and S_2 and the definition of the merging operator $\sqcup$.

Security. We now prove the security of our OAS scheme based on the computational co-Diffie-Hellman assumption, that is defined as follows.

DEFINITION 5. *We say that the computational co-Diffie-Hellman problem in $(\mathbb{G}_1, \mathbb{G}_2)$ is hard if the probability that a polynomial-time adversary $\mathcal{A}$, on input g_2, g_2^a, h where $a \leftarrow_R \mathbb{Z}_p$ and $h \leftarrow_R \mathbb{G}_1$, outputs h^a is negligible.*

THEOREM 2. *If the computational co-Diffie-Hellman (co-CDH) problem is hard in $(\mathbb{G}_1, \mathbb{G}_2)$, then the above OAS scheme is unforgeable in the random-oracle model.*

PROOF. Given an adversary $\mathcal{A}$ against the unforgeability of the OAS scheme as per Definition 3, we construct an adversary $\mathcal{B}$ that solves the co-CDH problem in $(\mathbb{G}_1, \mathbb{G}_2)$. Algorithm $\mathcal{B}$ gets as input g_2, g_2^a, h. It runs $\mathcal{A}$ on input $pk = g_2^a$, answering its oracle queries as follows:

- Random-oracle queries $H(m)$: Without loss of generality, we assume that $\mathcal{A}$ never asks the same random-oracle query twice and always queries $H(m)$ before querying a signature on m or using it as part of its forgery. $\mathcal{B}$ guesses one query index $q^* \leftarrow_R \{1, \ldots, q_H\}$, where q_H is (an upper bound on) the number of random-oracle queries issued by $\mathcal{A}$. $\mathcal{B}$ answers all of $\mathcal{A}$'s random-oracle queries other than the q^*th query so that it can generate a valid signature by choosing $r \leftarrow_R \mathbb{Z}_p$, storing (m, r) and returning g_1^r as random-oracle response. For the q^*th query, $\mathcal{B}$ stores $(m, \perp)$ and returns h as response.
- Signature queries $\mathsf{Sign}(sk, m)$: $\mathcal{B}$ looks up a tuple (m, r). If $r = \perp$ then $\mathcal{B}$ aborts. Otherwise, it returns $\psi((g_2^a)^r) = g_1^{ar}$ as the target signer's signature for m.

When $\mathcal{A}$ produces its output $(\alpha = (\tau, \mathcal{B}), S_\perp, (pk_1, \ldots, pk_n), (sk_1, \ldots, sk_n))$, it checks that $pk_i = g_2^{sk_i}$ for all $i = 1, \ldots, n$ such that $pk_i \neq pk$, and runs $apk \leftarrow \mathsf{AggPK}(\{pk_1, \ldots, pk_n\})$. It verifies the signature as $\mathcal{B} \leftarrow \mathsf{Verify}(apk, S_\perp, \alpha, M)$. Let $S_M = \{pk_1, \ldots, pk_n\} \setminus S_\perp \setminus \bigcup_{(m_i, S_i) \in \mathcal{B}} S_i$. For $\mathcal{A}$ to be successful in its attack, it must hold that $S_\perp \subseteq S$ and $S_i \subseteq S$ for all $(m_i, S_i) \in \mathcal{B}$, and hence by consequence that $S_M \subseteq S$. Also, there must either exist $(m^*, S^*) \in \mathcal{B}$ such that $pk \in S^*$ and m^* was never queried from the Sign oracle, or it must hold that $pk \in S_M$ and $\mathcal{A}$ never queried M from the Sign oracle. In the first case, with probability $1/q_H$, m^* was $\mathcal{A}$'s q^*th random-oracle query. If not, $\mathcal{B}$ aborts; otherwise, $\mathcal{B}$ knows the discrete logarithms r_M of $H(M)$ and r_i of $H(m_i)$ for $m_i \neq m^*$. It can compute a solution to the co-CDH problem as follows. From the verification equation, we know that

$$\begin{aligned} e(\tau, g_2) &= e\big(H(M), apk_M\big) \cdot \prod_{(m_i, S_i) \in \mathcal{B}} e\big(H(m_i), \prod_{pk \in S_i} pk\big) \\ &= e\big(\psi(apk_M)^{r_M}, g_2\big) \cdot \prod_{\substack{(m_i,\, S_i) \,\in\, \mathcal{B} \\ m_i \,\neq\, m^*}} e\big(\prod_{pk \in S_i} \psi(pk)^{r_i}, g_2\big) \\ &\quad \cdot e\big(\prod_{pk_i \in S_i \setminus \{pk\}} h^{sk_i}, g_2\big) \cdot e\big(h^a, g_2\big). \end{aligned}$$

One can therefore compute h^a as

$$\tau \cdot \Big(\psi(apk_M)^{r_M} \cdot \prod_{\substack{(m_i,\, S_i) \,\in\, \mathcal{B} \\ m_i \,\neq\, m^*}} \prod_{pk \in S_i} \psi(pk)^{r_i} \cdot \prod_{pk_i \in S_i \setminus \{pk\}} h^{sk_i}\Big)^{-1}.$$

In the second case that $m^* = M$, again with probability $1/q_H$ we have that M was $\mathcal{A}$'s q^*th random-oracle query. If so, then $\mathcal{B}$ knows all discrete logarithms r_i of $H(m_i)$, so that the verification equation becomes

$$\begin{aligned} e(\tau, g_2) &= e\big(\prod_{pk_i \in S_M} h^{sk_i}, g_2\big) \cdot e\big(h^a, g_2\big) \\ &\quad \cdot \prod_{(m_i, S_i) \in \mathcal{B}} e\big(\prod_{pk \in S_i} \psi(pk)^{r_i}, g_2\big). \end{aligned}$$

$\mathcal{B}$ can compute h^a as

$$\tau \cdot \Big(\prod_{pk_i \in S_M} h^{sk_i} \cdot \prod_{(m_i, S_i) \in \mathcal{B}} \prod_{pk \in S_i} \psi(pk)^{r_i}\Big)^{-1}.$$

$$\omega \leftarrow \left(\prod_{pk_j \in S^* \setminus \{pk\}} H(m^*) \prod_{(m_i, S_i) \in \mathcal{B}} \prod_{pk_j \in S_i} H(m_i)^{sk_j} \cdot \prod_{\substack{i\,:\, pk_i \,=\, pk \\ \wedge m_i \,\neq\, m_{i^*}}} (g_2^a)^{r_i}\right)$$

$$\sigma \;\leftarrow\; \tau/\omega$$

Note that, for this reduction to work, we really rely on the fact that $S_\perp$ and S_i are all subsets of S, because otherwise $\mathcal{B}$ does not know the necessary secret keys. □