SANA: Secure and Scalable Aggregate Network Attestation

Moreno Ambrosin\textsuperscript{1}, Mauro Conti\textsuperscript{1}, Ahmad Ibrahim\textsuperscript{2}, Gregory Neven\textsuperscript{3}, Ahmad-Reza Sadeghi\textsuperscript{2}, and Matthias Schunter\textsuperscript{4}

\textsuperscript{1}University of Padua, Italy
\textsuperscript{2}IBM Research - Zurich, Switzerland
\textsuperscript{3}Technische Universität Darmstadt, Germany
\textsuperscript{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 \cite{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 control 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 \cite{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 \cite{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.
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, which supports 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 [13], TrustLite [18], 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 $G$, with either static or dynamic topology. $G$ may not have a routing protocol in place. However, devices in $G$ are able to identify and communicate to their direct neighbors, which is a minimal assumption in such networks [31].

We formally define SANA as a protocol between the following logical entities: **prover** ($P$), **aggregator** ($A$), **owner** ($O$), and **verifier** ($V$). 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** $P_i$, while we refer to the rest of the provers as **bad provers** $\tilde{P}_i$ (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 $O$ represents the network owner or operator, responsible for the deployment, as well as the maintenance, of every prover $P_i$ in $G$. Note that, a physical device in $G$ can embed the functionalities of every logical component described above, or a combination of them. As an example, in Figure 1 device $D_1$ acts both as a prover $\tilde{P}_1$, and an aggregator $A_2$, while $D_5$ acts only as a prover $\tilde{P}_5$.

The goal of a collective attestation protocol is to assure a verifier $V$, which is typically, but not necessarily, the same entity as $O$, of the overall integrity of $G$, i.e., the software integrity of every prover $P_i$ in $G$. Provided that none of the provers is physically attacked, $G$ is considered trustworthy if all the provers in $G$ are good, i.e., have the latest non-compromised software version approved by $V$. 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 $G$ 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 $G$ 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, $\tilde{P}_1$, $\tilde{P}_2$, $\tilde{P}_3$, $\tilde{P}_4$, and $\tilde{P}_5$, the owner $O$, and a verifier $V$. Each prover $P_i$ is initialized with the cryptographic material needed to execute SANA collective attestation protocol (operation $\circled{1}$ shown in Figure 1). The initialization is performed in a secure environment, and preferably, but not necessarily, by $O$. At a given time, a verifier $V$, which possesses an appropriate attestation token generated by $O$, may attest $G$. Note that, if $V$ and $O$ are two distinct entities, the token is securely exchanged offline (operation $\circled{2}$ in Figure 1).

In order to attest the network, $V$ 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 \( P_1, P_2, P_3, P_4 \) and \( P_5 \), create their attestation response and send it to their parent nodes (operation 3 in Figure 1). Aggregators, i.e., non-leaf nodes \((A_1, A_2, A_3, A_4)\), in turn, aggregate the attestation responses received from their child nodes (operation 3 in Figure 1), and forward the result to their parents. Finally, the aggregated report is received and verified by \( V\).

### 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 verifi er 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 verifi er 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.

- **Scalability.** The protocol allows a verifi er to attest a large network of devices. The communication and computational complexity for prover and verifi er 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 verifi er 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 verifi er.

- **Privacy Preservation.** Verifi cation does not require detailed knowledge of the configuration of \( G \) (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 \( G \).

- **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.

#### Security Assumptions

We assume all provers in \( G \) correctly implement the minimal hardware features required for secure remote attestation described in [13], and adapted in [5]. A potential implementation of \( P \) 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 \( O \) 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), \( \ell_n \) indicates the bit-length of \( n \). Let \( m \leftarrow U \) 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 \( \ell \). 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 \( y \leftarrow A(x) \) means that on input \( x \), \( A \) assigns its output to variable \( y \). We occasionally overload notation to let \( A(x) \) 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 \( \text{KeyGen}(\ell_{\text{Sig}}) \).

We denote with \( A \) an algorithm \( A \) that arbitrarily interacts with algorithm \( B \) while it is executing. The term \( \text{prot} \left[ A : x_A; B : x_B; \ast : x_{\text{pub}} \right] \rightarrow \left[ A : y_A; B : y_B \right] \) denotes an interactive protocol \( \text{prot} \) between two probabilistic algorithms \( A \) and \( B \). Hereby, \( A \) (resp. \( B \)) gets private input \( x_A \) (resp. \( x_B \)) and public input \( x_{\text{pub}} \). 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 [16, 24] 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 \((\text{KeyGen}, \text{AggPK}, \text{Sign}, \text{AggSig}, \text{Verify})\). On input of the security parameter \( \ell \in \mathbb{N} \), KeyGen outputs a secret signing key \( sk \) and a public verification key \( pk \), for each device \( D \), i.e., \((sk, pk) \leftarrow \text{KeyGen}(\ell)\). On input a set of public keys \( \{pk_1, \ldots, pk_n\} \), the public key aggregation algorithm \( \text{AggPK} \) outputs an aggregate public key \( apk \). On input a secret key \( sk \), a message \( m \in \{0, 1\}^* \), and the default message \( M \in \{0, 1\}^* \), \( \text{Sign} \) outputs a signature \( \alpha \) on \( m \), i.e., \( \alpha \leftarrow \text{Sign}(sk, m, M) \). Note that, the signature \( \alpha \) is already considered an aggregate signature, containing only \( pk \)'s signature on \( m \). On input two aggregate signatures \( \alpha_1, \alpha_2 \) and the default message \( M \), the signature aggregation algorithm \( \text{AggSig} \) outputs a new aggregate signature \( \alpha \) that includes all signatures in \( \alpha_1 \) and \( \alpha_2 \). To verify an aggregate signature, the Verify algorithm takes an aggregate public key \( apk \), a set of public keys \( S_i \), of signers who did not contribute a signature to the aggregate, an aggregate signature \( \alpha \), and the default message \( M \) as input, and outputs either \( 1 \) to indicate that the signature is invalid, or a set \( B = \{(m, S_i) : i = 1, \ldots, \mu\} \), indicating that the signers with public key \( pk_i \in S_i \) signed message \( m \), and that all other signers whose keys were included in the aggregated public key \( apk \) but not in any of \( S_i \) 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 \( B_1, B_2 \) containing tuples \((m, S) \in \{0, 1\}^* \times \{0, 1\}^*\), let \( B = B_1 \sqcup B_2 \) be the “merged” set of tuples \((m, S) \) where \( S = S_1 \sqcup S_2 \) if \( \exists (m, S_1) \in B_1 \) and \( \exists (m, S_2) \in B_2 \), where \( S = S_1 \) if \( \exists (m, S_1) \in B_1 \) and \( \not\exists (m, S_2) \in B_2 \) and \( \not\exists (m, S_1) \in B_1 \). \( \not\exists (m, S_2) \in B_2 \) and \( \not\exists (m, S_1) \in B_1 \). \( \not\exists (m, S_2) \in B_2 \) and \( \not\exists (m, S_1) \in B_1 \).

**Definition 2 (Correctness of OAS).** An OAS scheme is correct if:

(i) signing works, i.e., for all \( \ell \) \in \mathbb{N} \), all \( m, M \in \{0, 1\}^* \), all \( (pk, sk) \leftarrow \text{KeyGen}(\ell) \), all sets \( S \) such that \( pk \not\in S \), it holds that \( \text{Verify}(apk, S, \alpha, M) \) returns 0 if \( m = M \) and returns \( \{m, \{pk\}\} \) if \( m \neq M \) whenever \( apk \leftarrow \text{AggPK}(S_1 \cup \{pk\}) \) and \( \alpha \leftarrow \text{Sign}(sk, m, M) \).

(ii) aggregation works, i.e., for all aggregate signatures \( \alpha_1, \alpha_2 \) of disjoint sets \( S_i, S_j \) and sets \( S_i \subseteq S_1 \) and \( S_j \subseteq S_2 \) and all messages \( M \in \{0, 1\}^* \), if \( \text{Verify}(apk_1, S_{I, 1}, \alpha_1, M) = B_1 \) and \( \text{Verify}(apk_2, S_{I, 2}, \alpha_2, M) = B_2 \) for \(apk_1 \leftarrow \text{AggPK}(S_1)\), \(apk_2 \leftarrow \text{AggPK}(S_2)\), and \(apk \leftarrow \text{AggPK}(S_1 \cup S_2)\), then it holds that \( \text{Verify}(apk, S_{U, 1} \cup S_{U, 2}, \alpha, M) = B_1 \sqcup B_2 \).

**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 \( A \), the following experiment must return 0 with negligible probability:

\[
\langle pk, sk \rangle \leftarrow \text{KeyGen}(\ell) \\
\langle \alpha, S_1, (pk_1, \ldots, pk_n), (sk_1, \ldots, sk_n) \rangle \leftarrow A^{\text{Sign}(sk, \cdot)}(pk) \\
\text{If } 3 : i : pk_i \neq pk \land (pk_i, sk_i) \not\in \text{KeyGen}(\ell) \text{ then return } 0 \\
\text{Let } S = \{pk_1, \ldots, pk_n\} \\
apk \leftarrow \text{AggPK}(S) \\
\text{B} = \text{Verify}(apk, S, \alpha, M) \\
\text{If } S \not\subseteq S \text{ or } \exists (m_i, S_i) \in B : S_i \not\subseteq S \text{ then return } 0 \\
\text{If } \exists (m_i, S_i) \in B : pk \in S_i \text{ and } m_i \not\in Q \text{ then return } 1 \\
\text{Let } S_{\not\subseteq} \leftarrow S \setminus \bigcup_{S_i \in B} S_i \\
\text{If } pk \in S_{\not\subseteq} \text{ and } M \not\in Q \text{ then return } 1 \\
\text{Else return } 0
\]

where \( Q \) is the set of messages that \( A \) queried from its \( \text{Sign}(sk, \cdot) \) 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 [30] 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 \( S_i \) and \( S_j \) 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. [34] 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 $G_1, G_2, G_3$ be multiplicative groups of prime order $p$ with generators $g_1, g_2, g_3$, respectively, with an efficiently computable bilinear map $e : G_1 \times G_2 \to G_3$ so that $e(g_1^x, g_2^y) = g_3^{xy}$ for all $x, y \in \mathbb{Z}_p$, and with an efficiently computable isomorphism $\psi : G_2 \to G_3$ so that $\psi(g_2) = g_3$. Let $H : \{0, 1\}^* \to G_1$ be a hash function modeled as a random oracle [8].

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

### 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 $\alpha = (H(m))^x \in G_1$ if $m = M$ and is $\alpha = (H(m))^x \cdot (\{(m, \{pk_i\})\})$ otherwise.

### Signature aggregation
Aggregating two aggregate signatures $\alpha_1 = (\tau_1, B_1)$ and $\alpha_2 = (\tau_2, B_2)$ can be done by computing $\tau = \tau_1 \cdot \tau_2$ and "merging" $B_1$ and $B_2$ into $B = B_1 \cup B_2$. The resulting aggregate is $\alpha = (\tau, B)$.

### Verification
To verify an aggregate signature $\alpha = (\tau, B = \{(m_1, S_1), \ldots, (m_i, S_i)\})$ under aggregate public key $apk$, non-contributing public keys $S_k$, and default message $M$, let
\[
\alpha_{apk, M} = \prod_{pk_i \in S_k} pk_i \cdot \prod_{i=1}^{\mu} e(H(m_i), \prod_{pk_i \in S_i} pk_i).
\]
Verify that
\[
e(\tau, g_2) = e(H(M), \alpha_{apk, M}) \cdot \prod_{i=1}^{\mu} e(H(m_i), \prod_{pk_i \in S_i} pk_i).
\]
If so, then return $B$, otherwise return $\bot$.

As mentioned earlier, and as is the case for other multi-signature schemes [9, 21], 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 [30] and adding it to the public key, or by including a Schnorr signature from which the corresponding secret key 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 $V$, the owner $O$, and a set of aggregators and provers in the network $G$. 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 $O$ with an OAS key pair $(sk_i, pk_i)$, and an identity certificate $cert(pk_i)$, signed by $O$, certifying that $pk_i$ is a valid OAS public key of $P_i$ with identity $id_i$. Formally:
\[
\text{init}(1^t) \rightarrow (sk_i, pk_i, cert(pk_i)).
\]

### Token request
In order to attest a network $G$, a verifier $V$ must possess a valid authorization token $T$ generated and signed by the owner $O$ of the network. $V$ acquires $T = \{H, c_1, v_1, t_{exp}, \sigma_1\}$ by executing an offline protocol tokenReq (see Figure 2) with $O$. 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 $O$ keeps a list of counters $c_{i_1}, \ldots, c_{i_n}$ with values $v_{i_1}, \ldots, v_{i_n}$. A counter can be assigned by $O$ 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

### Table 1: Variables, parameters and procedures

<table>
<thead>
<tr>
<th>Entities</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O$</td>
<td>Owner or operator of the network</td>
</tr>
<tr>
<td>$V$</td>
<td>Verifier (entity attesting the network)</td>
</tr>
<tr>
<td>$P_i$</td>
<td>Device $i$</td>
</tr>
<tr>
<td>$\hat{P}_i$</td>
<td>Good prover $i$, i.e., a prover with one of the latest non-compromised software configurations</td>
</tr>
<tr>
<td>$\hat{P}_i$</td>
<td>Bad prover $i$, i.e., a prover with an outdated or malicious software configuration</td>
</tr>
<tr>
<td>$A_i$</td>
<td>Untrusted aggregator $i$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prover $P_i$ parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id_i$</td>
<td>ID of $P_i$</td>
</tr>
<tr>
<td>$h$</td>
<td>Platform software configuration (e.g., hash digest of binary code)</td>
</tr>
<tr>
<td>$(sk_i, pk_i)$</td>
<td>OAS secret and public key pair of $P_i$</td>
</tr>
<tr>
<td>$(pk_O, \text{resp. } \psi_O)$</td>
<td>Secret signing key of $O$ (not based on OAS)</td>
</tr>
<tr>
<td>$(pk_v, \text{resp. } \psi_v)$</td>
<td>Public signature verification key of $O$ (not based on OAS)</td>
</tr>
<tr>
<td>$(cert(pk_v), cert(pk_O))$</td>
<td>Identity certificate of $P_i$ (issued by $O$)</td>
</tr>
<tr>
<td>$({c_{i_1}, v_{i_1}}, \ldots, {c_{i_n}, v_{i_n}})$</td>
<td>List of attestation counters and corresponding values</td>
</tr>
</tbody>
</table>

**SANA parameters**

- **$T$**: Token used by $V$ to perform attestation ($T = \{H, c_{i_1}, v_{i_1}, t_{exp}, \sigma_1\}$)
- **$N$**: A random nonce
- **$H$**: The set of software configurations for the latest software versions of all devices in $G$
- **$t_{exp}$**: Expiry time of a token $T$
- **$\delta_i$**: Expiry period of a token $T$
- **$C_b$**: An attestation challenge ($C_b = \{N, T\}$)
- **$apk$**: Aggregate public key of all provers in $G$
- **$S_k$**: Set of public keys grouped by same message signed $m_i$
- **$S_\bot$**: 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
- **$\alpha$**: 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 $\delta > 0$ if the request is accepted
- **getFreeCounter()**: Searches for an unused counter $c_l$ in $c_{i_1} \ldots c_{i_n}$. 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 $H$ of software configuration for the latest software versions of all devices in $G$
Figure 2: Protocol tokenReq

request from $V$, $O$ searches for a free (i.e., not busy) counter $c_i$, with value $v_i$, increments $v_i$ by one, and returns the tuple $(c_i, v_i)$ 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: $V$ initiates the protocol by sending $O$ a random challenge $N_V$, showing its interest in attesting $G$. Upon receiving $N_V$, $O$ creates a random challenge $N_O$ and sends it to $V$. $V$ then creates a signature $\sigma_V$ on $N_O$ and a protocol parameter $\delta_t$, and sends it back to $O$ along with $h_1$, and its identity certificate $\text{cert}(pk_V)$. Parameter $\delta_t$ indicates the required expiration period of the requested $T$. Based on $\delta_t$, $V$ creates $\delta_t$, $O$, and determines whether to accept $V$’s request, according to an application specific policy – checkPolicy(). If the request is accepted and $\sigma_V$ verified correctly, $O$ retrieves the set $\mathcal{H} = \{h_1, \ldots, h_s\}$ of software configuration of benign software in $G$ (i.e., the software configuration of latest software version on different devices in $G$) – getGoodConfigs(). The list is then hashed into one single good configuration $h_g = \text{hash}(h_1) \ldots h_s$). Finally, $O$ sends to $V$: (1) the aggregate public key $apk$ of all provers in $G$; (2) A signature $\sigma_2$ over $apk$; and (3) An encrypted token $\epsilon$. Finally, $V$ verifies $\sigma_2$, decrypts and verifies $T$, and stores it along with $apk$. Formally:

$$\text{tokenReq} = [V: \delta_t, sk_V; O: sk_O, apk; \epsilon: \text{cert}(pk_O), \text{cert}(pk_V)] \rightarrow [V: T, apk; O: t_{exp}].$$

**Attestation:** After obtaining an attestation token $T$, $V$ can attest the network. Before $t_{exp}$, $V$ chooses a random (gateway) aggregator $A_1$, through which it runs the collective attestation attest of the whole network (see Figure 3). In detail, $V$ sends $A_1$ an attestation request

$$\text{attest} = [V: T, apk; A_1: -; \epsilon: pk_O] \rightarrow [V: r; A_1: Ch].$$

$$\text{attest}_2 = [A_1: Ch; D_j: (sk_j); \epsilon: pk_O] \rightarrow [A_i: \alpha_j; D_j: Ch].$$

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

We implemented SANA based on three recently proposed security architecture for low-end embedded devices: SMART [13], TrustLite [18], 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]](image)

TyTAN [11] is a security architecture for embedded systems, that is based on TrustLite [18]. 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 tasks 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 $\text{sk}_i$ (which resides in memory address $\text{Addr}_0$ to $\text{Addr}_{i-1}$) is only read accessible to $\text{createResponse}$() (i.e., code residing in memory address $\text{Addr}_3$ to $\text{Addr}_4$). 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 [36] 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 [36], which provides a strong security level of 128-bit.

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 $h_y$. The gateway aggregator $A_1$, which directly communicates to the verifier $V$, 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 $\ell_{\text{sign}} = 256$ bits. We also use $\ell_N = 160$, $\ell_c = 64$, and $\ell_{\text{sign}} = 320$. 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 $\alpha_i$ has size $32 + 32w + 20\mu$ bytes, where $\mu$ 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 $G$ stores the ids $(v_1 \ldots v_s)$ 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 $pk_O$ of $O$. The storage overhead for every $P_i$ is estimated as $10s + 228$ bytes, where $s$ is the number of counters used by $O$. 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 $G$ 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 operation 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 $d = f(n+a) \in O((\log(n+a)))$ 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 $t_{agg}$, $t_{ver}$, $t_{hash}$, and $t_{sign}$ 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:

$$t \leq \left(110d + \sum_{i=0}^{d}(32w_i + 20m_j)\right) \cdot t_{ver} +$$

$$+ \left(\sum_{i=0}^{d}p_i \cdot t_{agg} + d \cdot t_{ver} + t_{hash}\right) + t_{sign}.$$  

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, $\mu$ is the number of configurations in $G$, and $n$ is the number of provers in $G$.

Energy costs. Let $E_{\text{send}}$, $E_{\text{rec}}$, $E_{\text{sign}}$, $E_{\text{agg}}$, $E_{\text{ver}}$, and $E_{\text{hash}}$ 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 $G$ is:

$$E(A_i) \leq \left(32 + (20z + 78)g_i + 32w + 20m\right) \cdot E_{\text{send}} +$$

$$+ \left(20z + 78 + 32g_i + 32w + 20m\right) \cdot E_{\text{rec}} +$$

$$+ p_i \cdot E_{\text{agg}} + E_{\text{ver}} + E_{\text{hash}}.$$  

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

$$E(P_i) \leq 84 \cdot E_{\text{send}} + \left(20z + 78\right) \cdot E_{\text{rec}} + E_{\text{sign}} + E_{\text{hash}}.$$  

\(^5\)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.

Simulation results. We simulated SANA for large networks using the OMNeT++ [26] 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-1D. 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-
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.

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...
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 [27]. Examples are the works in [12, 28]. 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 aggregate 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 [25], require a sequential order [9, 21], or all the signatures to be created on the same message [6]. As a consequence, we consider none of the known schemes as suitable for our collective attestation scheme. Recently, the work in [34] 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 [34] 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 [34].

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 7th 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 ICh+/2014/342-896). Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980).

11. REFERENCES

APPENDIX

A. SECURITY OF SANA

The security goal of any collective attestation scheme is to ensure a verifier \( V \) that all the devices in a network are running a software that is known to, and accepted by \( V \). In other words, \( V \) should accept the attestation result, and declare the network \( G \) as benign, if and only if all devices in \( G \) (at least those that are not physically attacked) are running a software it agrees on. We formalize this goal as a security experiment \( \text{Exp}_A \) between an adversary \( A \) (as defined in Section 2.3), \( G \) and \( V \). In this experiment, \( A \) physically attacks a number of provers \( P_1, \ldots, P_r \) and modifies the software of at least one prover \( P_i \) that it did not physically attack. \( A \) may have full control of all aggregators. After a polynomial number (in terms of the security parameters \( \ell_N, \ell_r, \) and \( \ell_{\text{Sign}} \)) of steps by \( A \), \( V \) 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 \( V \), i.e., \( \text{Exp}_A = r \). A secure collective attestation scheme is defined as follows:

**Definition 4 (Secure Collective Attestation).** A collective attestation scheme is secure if \( \Pr[ r = 1 | \text{Exp}_A(V^*) = r ] \) is negligible in \( \ell \) if \( f(\ell_N, \ell_r, \ell_{\text{Sign}}) \), where the function \( f \) is polynomial in \( \ell_N, \ell_r, \) and \( \ell_{\text{Sign}} \).

**Theorem 1 (Security of SANA).** \( \text{SANA} \) 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 \( V \) returns \( r = 1 \), i.e., accepts an attestation response \( \alpha_3 \) only if \( B = \{ (s_1, \alpha_1), \ldots, (s_n, \alpha_1) \} = \text{Verify}(apk, S_\alpha, M, \alpha_1) \), where \( apk \) is the OAS aggregate public key of all provers in \( G \), \( S_\alpha = \phi \), \( B = \phi \), and \( M = (\text{hash}(h_1 \ldots h_{\ell_N})) | c_i \). \( N \) is the nonce, \( h_1, \ldots, h_{\ell_N} \) are all the benign software configurations included in \( H \), and \( c_i \) (resp. \( v_i \)) is the id (resp. value) of the counter previously sent by \( V \). To avoid detection of a compromised prover \( P_i \), \( A \) can use one of the following strategies: (1) \( A \) does not alter the response from \( P_i \); (2) \( A \) uses an old signature \( \alpha_{\text{old}} \) previously generated by \( P_i \) on the good software configuration \( h_{\text{old}} \); (3) \( A \) modifies \( H \) to include the software configuration of \( P_i \); (4) \( A \) generates an aggregate signature that attributes a response \( M = h_{\text{old}} | c_i | v_i \) to \( P_i \); or (5) \( A \) modifies the aggregate public keys \( apk \) sent from the owner \( O \) to \( V \) during tokenReq protocol.

We start with strategy (1). According to our assumptions in Section 2, \( A \) can neither tamper with the code performing integrity measurements on \( P_i \), nor extract its secret signing key \( s_{\text{old}} \). Consequently, the response of \( P_i \) will include an OAS signature on \( h_{\text{old}} | c_i | v_i \). Since \( h_{\text{old}} \) is the current software configuration of a maliciously modified software, \( V \) will always return \( r = 0 \), if the underlying integrity measurement mechanism that generates the software configuration can detect this modification (i.e., \( h_{\text{old}} \not\in H \)). On the other hand, if \( A \) uses an old signature \( \alpha_{\text{old}} \) over \( M_{\text{old}} = h_{\text{old}} | c_i | v_i \) (Strategy 2), \( B \) will be equal to \( \phi \) only if \( N = N_{\text{old}} \), which is negligible in \( \ell_N \), where \( \ell_N \) represents the size of the nonce. Consequently, \( V \) will always return \( r = 0 \).

Next, we consider strategy (3), where the adversary modifies \( H \) to include the software configuration \( h_{\text{old}} \) of \( P_i \). Since \( H \) is part of the authenticated token \( T \), \( A \) should forge the signature \( \sigma_1 \) generated by the owner \( O \). The success probability of this strategy is negligible in \( \ell_{\text{Sign}} \), where \( \ell_{\text{Sign}} \) represents the security parameter of the digital signature scheme used. Note that, \( H \) is also included in \( \alpha_3 \). Therefore, any modification to \( H \) consequently changes \( \alpha_3 \). According to strategy (4), \( A \) may generate an aggregate signature that attributes \( M = h_{\text{old}} | c_i | v_i \) to \( P_i \), where \( N \) is a fresh nonce sent by \( V \). Since the current software of \( P_i \) is not in \( H \), and since \( N \) is a fresh nonce, the probability of \( P_i \) signing \( h_{\text{old}} | c_i | v_i \) is negligible in \( \ell_N \). Consequently, as OAS is unforgeable according to Definition 3, the probability of finding such an aggregate is negligible in \( \ell_{\text{Sign}} \), where \( \ell_{\text{Sign}} \) represents OAS security parameter.

Finally, we consider \( A \) modifying the aggregate public key \( apk \) and replacing \( pk_c \) with \( pk_{\text{old}} \), to which it knows the secret key \( s_{\text{old}} \). \( A \) can at this point sign arbitrary messages (including attestation
responses) on behalf of $P_a$, regardless of $P_a$’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 $\ell_N$ and $\ell_{Sign}$.

Consequently, the probability of $A$ making $V$ return $r = 1$, when it has maliciously modified the software of one prover $P_a$ in $G$ (that it did not physically attacked), is negligible in $\ell_N$, $\ell_{Sign}$, and $\ell_{Sign}$. Similarly, if none of the provers is physically attacked, the probability of $A$ making $V$ return $r = 1$, when at least one device has a malicious software is negligible in $\ell_N$, $\ell_{Sign}$, and $\ell_{Sign}$. □

### B. CORRECTNESS AND SECURITY OAS

**Correctness.** If $sk = x$ and all parties behave honestly, then we have that $pk = g_2^x$ and $apk = \text{AggPK}(S_{\perp} \cup \{pk\}) = g_2^x \prod_{apk_i \in S_{\perp}} pk_i$. If $m = M$, then $\alpha = \text{Sign}(sk, m, M) = (\tau, B) = (H(M)^r, \emptyset)$ so that during verification we have that $\mu = 0$, $apk_{M} = \text{apk} / \prod_{apk_i \in S_{\perp}} pk_i = g_2^x$, and $e(\tau, g_2) = e(H(M)^r, g_2) = e(H(M), \text{apk}_{M})$ as required. If $m \neq M$, then $\alpha = \text{Sign}(sk, m, M) = (\tau, B) = (H(m)^r, \{m, \{pk\}\})$ so that during verification we have that $\mu = 1$, $apk_{i} = pk = g_2^x$, $apk_{M} = \text{apk} / \prod_{apk_i \in \{pk\}} pk_i = 1$, and, since $e(\tau, g_2)$, it is $e(H(m)^r, g_2) = e(H(m), \text{apk}_{i})$ as required.

To see that aggregation works, if $\text{Verify}(\text{AggPK}(S_{\perp}), S_{\perp}, \text{apk}_{M}) = B_{j}$ for $j = 1, 2$, by the verification equation (2) it holds that

$$e(\tau_{j}, g_2) = e(H(M), \text{apk}_{M_{j}}) \prod_{i = 1}^{\mu_j} e(H(m_{i,j}), \text{apk}_{i,j}),$$

where $\text{apk}_{i,j} = \prod_{apk_i \in S_{\perp}} pk$ and $\text{apk}_{M_{j}} = \prod_{pk \in S_{M,j}} pk$ for $S_{M,j} = S_{\perp} \setminus S_{\perp,j} \setminus \{m_i \mid S_i \in B_j \}$. Multiplying the verification equation for $j = 1, 2$ gives

$$e(\tau_{1}, g_2) = e(H(M), \text{apk}_{M_{1}}) \prod_{i = 1}^{\mu_1} e(H(m_{i,1}), \text{apk}_{i,1}) \prod_{i = 1}^{\mu_2} e(H(m_{i,2}), \text{apk}_{i,2}) = e(H(M), \text{apk}_{M}) \prod_{apk_i \in B} e(H(m_i), \text{apk}_{i}).$$

where $\text{apk}_{M} = \prod_{pk \in S_{M,1} \cup S_{M,2}} pk$, $\text{apk}_{i} = \prod_{apk_i \in S_{i}} pk$ and $B = B_1 \cup B_2$. 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 $G_1, G_2$ is hard if the probability that a polynomial-time adversary $A$, on input $g_2, g_3^h$, where $h \leftarrow \mathbb{Z}_p$ and $h \leftarrow G_2$, outputs $h^a$ is negligible.

**Theorem 2.** If the computational co-Diffie-Hellman (co-CDH) problem is hard in $G_1, G_2$, then the above OAS scheme is unforgeable in the random-oracle model.

**Proof.** Given an adversary $A$ against the unforgeability of the OAS scheme as per Definition 3, we construct an adversary $B$ that solves the co-CDH problem in $G_1, G_2$. Algorithm $B$ gets as input $g_2, g_3^h$. It runs $A$ on input $pk = g_3^h$, answering its oracle queries as follows:

- Random-oracle queries $H(m)$: Without loss of generality, we assume that $A$ 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. $B$ guesses one query index $q^* \leftarrow \{1, \ldots, q_M\}$, where $q_M$ is an upper bound on the number of random-oracle queries issued by $A$. $B$ answers all of $A$’s random-oracle queries other than the $q^*$th query so that it can generate a valid signature by choosing $r \leftarrow \mathbb{Z}_p$, storing $(m, r)$ and returning $g_2^r$ as random-oracle response. For the $q^*$th query, $B$ stores $(m, \_)$ and returns $h$ as response.

- Signature queries $\text{Sign}(sk, m)$: $B$ looks up a tuple $(m, r)$. If $r = \_ \text{ then } B$ aborts. Otherwise, it returns $\psi((g_2^r)^i) = g_2^{ar}$ as the target signer’s signature for $m$.

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

$$e(\tau, g_2) = e(H(M), \text{apk}_{M}) \prod_{apk_i \in S_{\perp}} e(H(m_i), \prod_{apk_i \in S_{i}} pk_i).$$

One can therefore compute $h^a$ as

$$\tau \cdot \left( \prod_{apk_i \in S_{\perp}} e(H(m^*), \prod_{apk_i \in S_{i}} pk_i) \right)^{-1}.$$