A zero-trust swarm security architecture and protocols, v1.0

Alex Shafarenko

October 21, 2024

Contents

Abstract

This report presents the security protocols and general trust architecture of the SMARTEDGE swarm computing platform. Part 1 describes the coordination protocols for use in a swarm production environment, e.g. a smart factory, and Part 2 deals with crowd-sensing scenarios characteristic of traffic-control swarms.

0. Executive summary

This document contains security recommendations and a detailed description of two security protocols.

Part 1 deals with a smart-factory type of situation where a swarm of robots receives a recipe (a smart contract) from Cloud and performs state-changing actions. Every state change requires a certain combination of messages from participating actors, which must be authenticated and which it should never be able to repudiate. None of the swarm members is trusted except for progress. The swarm is coordinated by a member that can change the membership of the swarm, called Coordinator. The state transitions are controlled by a member whose job it is to collect state changing messages from relevant members to affect the transition. Every state transition is recorded on a ledger called the Winternitz stack. If members disagree, the stack can be dumped and sent to an Adjudicator, which is a Cloud agent that is not involved with any of the swarms or their recipes. Thanks to the Winternitz Stack Protocol, the Adjudicator is able to determine a rogue member of the swarm solely by analysing the stack and the recipe. Its conclusions are formal proofs requiring neither extra data nor additional semantics.

The report makes a recommendation regarding the networking infrastructure and the communication primitives required for meeting the security objectives. The central concept is a witnessed broadcast channel on which any member of the swarm can transmit (assuming the time-division schedule or robust conflict resolution). The channel is a wireless channel and it is witnessed by agents whose location is unknown to a potential attacker. The witnesses help to ensure that the broadcast split, whereby the swarm is partitioned into nonintersecting groups each receiving a different but valid message without knowing it, is practically impossible.

Part 2 attends to another difficult problem of swarm computing: anonymous reputation update. This is relevant to a swarm of free agents that navigate a physical space (e.g. the road network) and send their sensing reports which can be accurate and relevant to varying degrees. The problem that we attempt to solve is that the reports are referenced to specific locations and are time-stamped which reveals the location of the sensors as a function of time. Since in many crowd-sensing scenarios the sensors travel with a human who either wears them on their person or keeps them in their vehicle, the sensing reports divulge the human's time-referenced movements across the physical space, which tends to be private data not to be divulged. So on the one hand, the relevance and accuracy of the report depend on the reputation of the submitter, but on the other the submitter must remain anonymous. Worse still, the reports must remain disassociated in the sense that it should be very difficult if not impossible to determine that two reports have been submitted by the same, albeit anonymous, principal. The reason for that is the practical risk of breaking the principal's anonymity by association. For example, if it were possible to determine that a long queue at a certain traffic light is reported every Friday at 5pm by a principal whose first report is received on the same day at 8am about a road junction in a certain sparsely populated district, it would be relatively easy to correlate the two time-referenced locations and observe a single vehicle or a very small number of vehicles that fit the bill.

The solution described in the present report is zero-trust (in keeping with Part 1), properly anonymous and dissociative. It is based on two servers (or two groups of servers depending on the network infrastructure): a Registration Authority, RA, and the Reputation Server, RS. A client registers with the RA, either anonymously or not, either for free or for a fee. Non-anonymous registration may be required to limit access to the system to a certain type of client. At registration the client is assigned the lowest reputation and an anonymous certificate containing the reputation level and a blindly signed temporary identifier is issued to the client.

The client goes through rounds of reporting and reputation updates. The protocol, the Anonymous Reputation Update (ARU) protocol, ensures that the RS receives the sensing report and issues a reputation update coupon without knowing the client's identity, except the client's reputation as certified by the RA.

One could say that the client's anonymity is within the group of equally reputable clients. If the reputation levels are only a few and provided that there are a large number of clients, this limited anonymity should be sufficient. The ID contained in the certificate is not linked with the client's true identity but it contains a secret not available to the issuing RA since the signing is blind. The secret is used by the client to bind the certificate (and its reputation level, signed by the RA) with the sensing report. A similar trick is used to obtain a blindly signed reputation update coupon, which contains the client's new secret and a new reputation level. The client executes a random delay (to prevent an adversary's time-correlation) and submits the coupon back to the RA to obtain an updated certificate for the next round. The protocol guarantees that even if the RS and the RA are colluding, it is practically impossible for the RA to bind the new reputation in the coupon with the sensing report and thus learn the client's private travel data. The protocol enables the client to cryptographically convince the RA that the coupon belongs to the same client that the certificate was issued for, and because the former is signed by the RS, that the new reputation is properly earned — but nothing beyond that.

Part 2 contains detailed descriptions of the ancillary service required to run the protocol, the Extended Content-Addressable Storage (ECAS). This is intended as a Cloud service that makes it possible to execute the protocol completely asynchronously, with the client sending messages without return address via an unspecified number of intermediaries (for example via ZigBee) and the Server (either RA or RS) responding by deploying a self-certified tagged message in the ECAS, which can also be accessed via arbitrary intermediaries without jeopardising the security of the data contained therein.

Both parts propose protocols *specifically tailored* to support low-power IoT devices. Non-Cloud entities only require the ability to compute hash functions (Part 1) and occasionally perform ten or so modular multiplications (Part 2). These computations are available to modest microcontrollers operating on quite limited battery power.

The present document is work in progress. Questions and comments should be sent to Dr Alex Shafarenko (`a.shafarenko@gmail.com`).

1. Coordination Protocols

1.1 Introduction

Threat model. A swarm is assembled from smart nodes by a *coordinator*, which receives a *recipe* from Cloud. This document is not concerned with Cloud security. We recognise that the universe of smart nodes that can execute the recipe is finite and properly registered with the Cloud-based owner. Having said that, it is desirable to treat the smart nodes as untrustworthy to prevent insider attacks, which constitute a major threat in distributed computing environments in general and smart factories in particular.

The coordinator appoints a smart node to a role of *orchestrator* to *run* the recipe with the swarm. The recipe defines *roles* and minimum *capabilities* of nodes performing the role. The coordinator may order the orchestrator to replace a node that performs a certain role by another sufficiently capable node. There can be a role that allows multiple nodes to perform it. For such a role the coordinator may *add* a node to the swarm or *remove* it. A replacement is tantamount to removing a node and adding another in the same role as a single transaction.

The following security assumptions are made:

Coordinator is authorised to deal with the composition of the swarm as it sees fit. It is trusted for onboarding and offboarding nodes (by instructing the coordinator to execute appropriate security protocols) and for launching, suspending and terminating the recipe. In doing so it relies on the orchestrator, which is trusted for progress, but not data.

Orchestrator is appointed by the Coordinator and is responsible for maintaining a single, distributed, linearly ordered, immutable ledger of the recipe. It is trusted to run the ledger protocol without deliberately stalling execution, but this trust is not absolute. Any node can alert the Coordinator to the lack of progress if the orchestrator consistently ignores transaction requests from any swarm node involved in the recipe. The orchestrator is not trusted on data: every smart node participating in the recipe is enabled by the ledger protocol to collect a self-contained proof of inconsistency, which is then submitted to the Cloud adjudication service and notified to the Coordinator. Consequently, the Orchestrator is not trusted *post hoc.*

Smart nodes are not trusted for either progress or data. Any Byzantine behaviour of a smart node is detectable by the rest of them involved in the recipe as a result of running the ledger protocol. The minimum security requirement is for the swarm to have at least one honest smart node (which may or may not be the Orchestrator) in addition to the Coordinator as this would eliminate any chance of collusion.

Recipe. In a low-code approach to edge computing the recipe is a set of actions (programmed in high-code or pre-prepared low code graphs) that change the state of the system. The actions are *triggered* by the availability of all necessary inputs, and the inputs are messages received from specific principals (smart nodes). For example in a furniture factory a table recipe may contain an action of screwing table legs onto a tabletop, for which the availability of parts is required as well as a vacant threaded hole in the tabletop. The action AssembleTable has a state variable: an integer NHoles, initialised with 4. The assembly robot

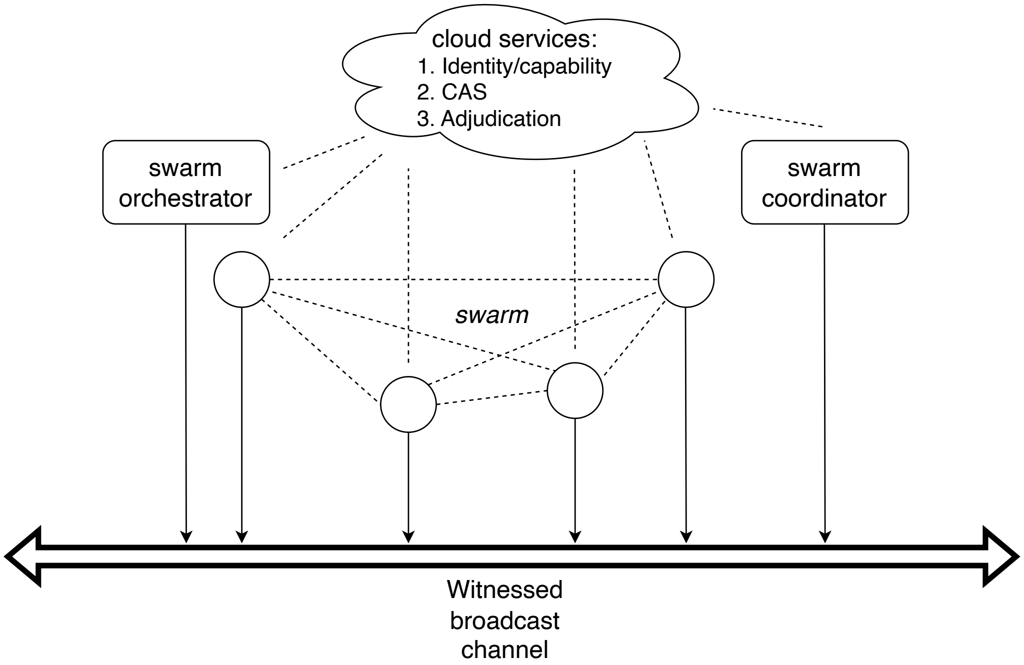

Figure 1.1: Overall architecture

is contacted by the tabletop maker robot as it supplies the tabletop, and the leg maker does the same as it produces a leg. We assume that the assembly robot does not need any fixtures to be supplied by yet another principal. So the robot forms a joint supply message, collects signatures from the two makers and adds its own, screws down the leg and sends the completed joint message to the AssesmbleTable action. As a result, the state variable NHoles is decremented. As well as any smart contract, the recipe is executed by all smart nodes involved at the same time, so each has a consistent copy of all state variables. The tabletop producer will see NHole turning to 0 as a signal to supply the next tabletop.

The overall system architecture is presented in Figure 1.1. It depicts one swarm, but there can be many that share the cloud services and the witnessed broadcast channel (WBC).

Witnessed Broadcast Channel. The functionality required of the WBC is twofold. On the one hand, it is a broadcast channel, so a message send on it is transmitted for any and all connected nodes to receive. It should be impossible to segment the WBC in such a way that a group of nodes receive a broadcast while the rest of the nodes in the swarm are either not aware that a broadcast is taking place or they receive different valid data. On the other hand, a node should not be able to falsely claim that it has broadcast some valid data while no broadcast has taken place. These security requirements are achieved by *silent witnesses*, which are smart nodes outside the swarm that listen but do not transmit on the channel until a protocol violation has been detected by at least one of them. They are connected to the channel, but may also be talking to each other in ways not exposed to the swarm, as well as talking to Cloud. When talking to each other, they may detect inconsistency in received data, which is reported to Cloud to help to detect an attack and to stop execution of the swarm contract.

The key property of a witnesses is that the attacker (including a compromised insider) does not know which node it is and where it is located, so it should be difficult to suppress it. The WBC and witnesses collectively constitute a light-weight, auxiliary consensus mechanism. It is light–weight since it does not require an expensive proof of work, stake, etc, nor any proper Byzantine protocol to validate transactions, and it is auxiliary because its purpose is to support self-contained post hoc proofs of inconsistency that the ledger protocols generate, rather than to pre-validate a transaction. The support is in terms of blocking some threats to prevent the vulnerability of the swarm protocols from being exploited by external or internal adversaries.

The life of a swarm begins with the Cloud appointment of a *coordinator*, which receives a *recipe*, i.e. a smart contract to be executed collectively by the newly created swarm.

Swarm coordinator functionality. The swarm coordinator's job is to maintain the swarm. This includes

- Appointment of the orchestrator (activity driver) and passing to it the instance of the recipe, i.e smart contract to be executed
- Appointment of smart nodes that collectively execute the contract
- Onboarding/offboarding nodes
- Terminate the contract at Cloud's order

The first step in unfurling the swarm structure is the appointment of the orchestrator. This is done using the Identity/Capability Service (ICS) via the Cloud. Then smart nodes needed for the recipe are appointed similarly. A recipe may permit proactive expansion/contraction, whereby a new smart node may request to join or a current one to leave in the process of execution. This is done via the coordinator, which receives such requests, determines their validity and passes all valid ones on to the orchestrator. In the process, ICS is used to determine not only capability but the sufficient reputation of the requester (for joining). Leaving the swarm is permitted or denied by the orchestrator on the basis of the recipe itself.

Orchestrator. The orchestrator maintains the state of the recipe (a dictionary of symbols) and a *Winternitz stack*, defined later, the latter reflecting the history of all swarm messages communicated via the WBC as well as its own.

Upon initiation, the orchestrator gathers every participating node's pseudonymous identity (a public key, which is the end-point of the node's *Winternitz chain* generated specifically for participation in the swarm under the current recipe). It then receives blocks from a group of nodes (down to a singleton) participating in a transaction, and places each block on the Winternitz stack while signing it by running the group RWS (GRWS) protocol based on [13] with all swarm members. The GRWS ensures that all stack frames are effectively digitally signed by their originators for content, and by the rest of the swarm for awareness.

To start with, each smart node sends the Orchestrator its Schnorr identity [11], which we will discuss later. We use the version of the protocol augmented by Fiat-Shamir heuristic[5] to avoid the need to provide a fresh random challenge. The identity consists of the hash of the node's Schnorr public key and the random exponential commitment from the node. In the next stack frame, the node provides the zero-knowledge proof using as the challenge the hash of the commitment concatenated to the swarm members' latest Wintenitz chain-tops. This way the pseudonym and the zero-knowledge identity are securely linked via GRWS. All other smart nodes in the swarm will then be in a position to silently verify the identity proof.

If the recipe allows participants to be anonymous, then 0 is pushed onto the Winternitz stack in place of the Schnorr identity to indicate that the Schnorr protocol is not run for this anonymous smart node.

As soon as all smart nodes in the swarm have verified every other smart node, they push the hash of the shared secret (a short hash, say 8 bytes) on the stack and invalidate the same number of bytes in the shared secret. The rest of the bytes are used from now on to produce MACs (message authentication codes) for each transmission of a swarm node on the WBC. Now the recipe can start executing.

The details of all protocols follow in the following sections.

If the Coordinator signals to the Orchestrator that a member of the swarm must be offboarded, it does so by pushing a command onto Winternitz stack. The rest of the swarm adjust their GRWS protocol tables to exclude the specified member. Similarly if the Coordinator needs to introduce a node into the swarm, it will push the appropriate command onto the stack, and the Orchestrator and the rest of the swarm adjust their protocol tables. The stack remains self-contained requiring no eternal information for its proof of valiadity.

1.2 Threat model and security objectives

Each security protocol has to make assumptions about the attacker and possible abnormal behaviour of the principals. Choosing the correct threat model is key for both effectiveness and efficiency of the cryptographic protection of open networks. Too esoteric threats, if taken on board, may result in a solution with a prohibitive resource footprint or a long latency or both. However, an unduly optimistic threat model is fraught with security breaches. Our analysis of the SmartEdge circumstances leads to the following assumptions:

A1 Any broadcast on the WBC remains effective under jamming. This means that the channel has sufficient retransmission resources to make sure that each node listening on the WBC will receive every broadcast message, either directly or via a gossip network if one exists. Conversely, there are sufficient silent witnesses on the WBC to guarantee that if none of them received a message claimed to have been broadcast, then it was not; or if they received valid and conflicting broadcasts, then malicious network segmentation has occurred.

A2 Members of a swarm can be compromised. Compromised members may collude but there will always be at least one non-colluding member in the swarm. It is not guaranteed that this honest member is necessary the Orchestrator or the Coordinator.

The first assumption requires some hardware fine-tuning. An adversary should not be able to jam members' broadcasts unnoticeably to all, and if jamming is detected, there should be an alternative unsecured broadcast channel that can be witnessed by the same witnesses, while any countermeasures are being deployed.

The second assumption is a natural constraint, since total collusion (all with all) defeats any protocol with post hoc verification: the principals are able to re-run it from beginning to end with different data and fake all their transactions. Fortunately, with minimum physical security and a firewall for outside communications it is possible to guarantee that compromised nodes are a small minority.

The following security objectives are achieved by the protocols presented in later sections:

Non-repudiation The core protocol, GRWS, maintains a single ordered immutable record (a mini-blockchain) of all transactions digitally signed by all smart nodes participating in the recipe. Assuming progress is assured, the broadcast is complete, and no deliberate non-transmission is detected (A1) no principal is able to forge any signature or roll back any transaction unless all principals collude (in violation of A2).

Robust identity Each principal is able to prove its identity cheaply (at the cost of 1 modulo exponentiation, thanks to the Schnorr protocol) and effectively: the proof of identity is executed once, at the beginning of the recipe execution. The identity is maintained for all transactions on the recipe thanks to its being linked with the GRWS pseudonym. Because Schnorr is run only once, it could be sufficient strength to secure against a possible quantum attack without degrading the overall system performance. Note that the identity protocol produces a single-bit result (genuine/fake) and does not introduce a digital signature.

Resistance to DoS An external adversary is unable to impede progress by sending fake messages on behalf of any principals. Not only are these messages unable to cause a transaction (thanks to GRWS), they cannot even waste the stack space. This is due to the use of Schnorr identity, which can be repurposed for the Static Diffie-Hellman protocol which requires no extra exchange between communicating parties. All messages addressed to the Orchestrator and Coordinator are authenticated by using shared secrets produced from the sender and receiver identities. The proposed solution continues to resist DoS after swarm contraction even when the contraction is the result of expelling a compromised node, which effectively publishes all secrets shared with it.

1.3 Orchestration Protocols

We begin with the group version of the recently published RWS protocol [13] which is a trivial extension of the RWS.

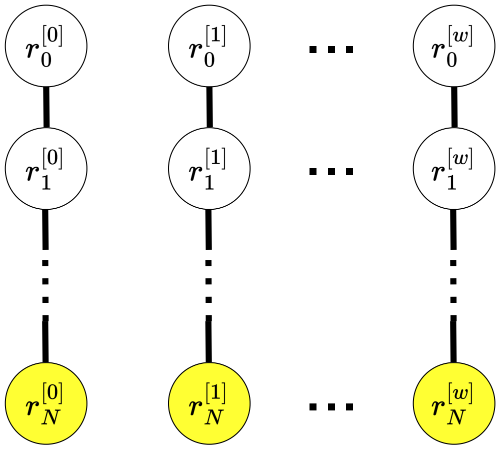

Figure 1.2: Winternitz fabric. The shaded nodes represent the fabric edge. A vertical line followed down connects a value with its hash image

1.3.1 Group RWS (GRWS) protocol

This subsection contains a brief and informal recount of the concepts pertaining to the RWS protocol; detailed definitions and proofs are available from [13].

A Winternitz chain is a sequence of integers, called *points*, each of which is a hash of the previous one, with the first integer being physically random. The end point of the chain serves as a "public key". Only the owner of the first point (which is never revealed to anyone) is able to make a backward step on the chain. We treat a hash function as a *random oracle*, which means:

1. it is a function, so if it is computed on the same value again, the result is the same;
2. the statistics of the result over the domain of the function is such that the result bits are 0 and 1 with equal frequency, and that different bit positions are uncorrelated;

Of the cryptographic hash function (as opposed to the idealised random oracle) we require in addition that it is computationally infeasible to find the argument for a given hash value, if it is unknown (preimage resistance) and that whenever the preimage is known, finding a different value that produces the same hash is computationally infeasible (second preimage resistance), both of which are believed to have been achieved with modern crypto hashes, such as SHA-2 (both 256 and 512 variety). The hash function's preimage resistance is what makes the step backwards on the chain only available to the chain owner. The second-image resistance is what makes each stack frame (see below) reliably linked with a single signed document.

A *Winternitz fabric* is an indexed set of Winternitz chains of the same length. The cardinality of the set is called the *width* w of the fabric. The end-points of all chains form a vector that we call the *edge* $E(F)$ of the fabric F.

A *Winternitz stack* over a fabric is a set of frames placed on top of one another. Each frame contains a fixed number ($\kappa \ll w$) of fabric points marked off on top of the previous frames or the fabric edge. Marking a point means for the stack owner to publish the point's content on the WBC. More than one member can be marked in any given chain of the fabric without gaps; this is done by publishing the highest point. In principle, the arity could be discovered by repeatedly hashing the value until it matches the chain member already exposed, thus obviating the explicit publication. Unfortunately as shown in [14], the probability of multiple members drops only about linearly with the number of members, so if the publication is fake, the listeners of the WBC will need to execute of the order of w hash functions before they can prove it to themselves. Still, since $\kappa \ll w$ and since there cannot be more than κ fabric members included in a frame, the calculation is affordable.

The cardinality-κ multiset of fabric indices is linked with a fixed size document digest via the current and previous frames. This is done by forming a *signature* by

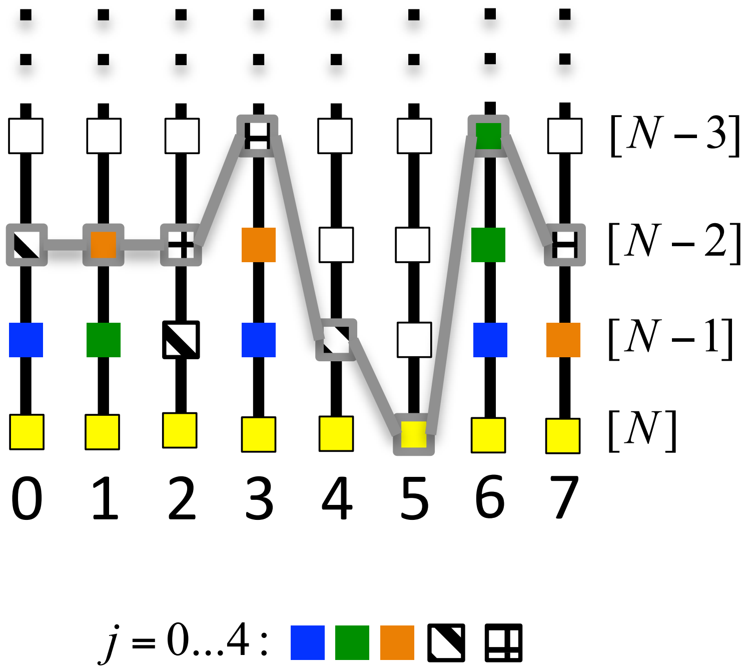

Figure 1.3: An example of a depth-5 signature stack over a Winternitz $(8, N)$-fabric using a random oracle with $\kappa = 3$. Empty boxes represent unused members of the fabric. The top of the stack T is highlighted in grey and the fabric edge in yellow. The document family is not shown.

1. taking the hash of the concatenation of the current digest and the previous signature; the hash's initial $\kappa \log_2 w$-bit segment becomes the current signature;
2. Segmenting the current signature into κ chunks of size $\log_2 w$ bits each. The result is a cardinality-κ multiset of binary numbers, $\{s_i\}$, $1 \leq i < \kappa$, each in the range $0, \ldots, w-1$. The numbers, which represent fabric indices, are not necessarily pairwise distinct, hence the multiset;
3. the current stack frame is formed by marking the chain members according to the multiset;

Figure 1.3 illustrates a signature stack of depth 5.

1.3.2 Protocol

Principals: orchestrator **R**, swarm members $\mathbf{M}_i$, $i = 1, \ldots, N$
Private (initially):

For R: Width-w fabric F of length L, except the edge $E(F)$;

For each $\mathbf{M}_i$: Chain $C_k^{[i]}$, $k = 0, \ldots, 2L-1$, of length $2L$, except the end point $C_{2L-1}^{[i]}$;

Public(initially):
L, w, R's public key $E(F)$, members' public keys $P_i = C_{2L-1}^{[i]}$, $i \in [1, N]$

For rounds $k = 1, \ldots$ **do**

1. **R** receives a digest D_k to be posted on the stack. Only one digest is accepted in the current round, the rest are declined. To guarantee progress each $\mathbf{M}_i$ is assigned priority $i + k \bmod N$; the highest priority submission wins. The sending of the digest requires booking. Node **R** receives booking requests from

one or more swarm members for the current round and satisfies the highest priority one. This helps to avoid the cost of forming a correctly signed digest, which according to step 6 below requires knowledge of which round k it will be broadcast in. We use Message Authentication Codes (discussed in section 1.3.3) to pre-share a common secret between swarm members that require authenticated messaging, in particular booking requests/confirmation and D_k, from some $k = k^*$ onward.

Note that the Digest is typically the hash of some content stored in the CAS after the booking request has been granted and the round k in which the content is to be posted has become certain. The orchestrator safeguards the content by copying it from the CAS to its local storage to be able to complete the stack dump for external adjudication. If the booking node sends a properly authenticated hash that is not a name of a CAS file at the reception time, the booking is annulled and the incident is reported to the Coordinator.

2. $\mathbf{R}$ computes the current frame message $x_k = H(x_{k-1} \,||\, D_k)$ and produces the multiset $\{s_j\}_{j\in[1,\kappa]}$ from it.

3. for $j \in [1, \kappa]$ do:

   R: strip fabric chain s_j of its end point, e_j, and broadcast e_j on WBC. No integrity control (authentication code) is necessary.

   For each $\mathbf{M}_i$: Receive one WBC broadcast r, compute $H(r)$ and find in the current fabric edge the endpoint with this value. Strip the endpoint, and note its corresponding index, s_j. Place that index in the *frame buffer*. If $H(r)$ does not match, initiate ARQ.

4. $\mathbf{R}$ broadcast D_k. No integrity control is necessary.
   For each $\mathbf{M}_i$:
   compute $H(x_{k-1} \,||\, D_k)$, segment it into a multiset and compare the result with the current frame buffer. If it matches, places it in the verification buffer. If it does not match, initiate ARQ.

5. **For each $\mathbf{M}_i$:**
   Update P_i to $C^{[i]}_{2L-2k+1}$ and broadcast it on WBC, listening at the same time to broadcasts p'_j by any other $\mathbf{M}_j$. From $H(p_j) = P_j$ find j and update P_j to p_j. If no match found, initiate ARQ. Each successful update advances the downcount from an initial $N - 1$ down to 0.
   R: Receives broadcasts from all $\mathbf{M}_i$ and updates its stored values of P_i (ARQ otherwise) counting down from N to 0.

6. Same as the previous step except the update of P_i is to $C^{[i]}_{2L-2k}$. Two near-identical steps are required to verify that all $\mathbf{M}_i$ have completed verification of D_k.

At the end of the kth round all $\mathbf{M}_i$ have received and validated the candidate digest D_k. The digest has the following structure:

$$D_k = H(\tau_k \,\big|\big|\, Q_k \,\big|\big|\, \mu_1^{[k]} \,\big|\big|\, \ldots \,\big|\big|\, \mu_m^{[k]}) \,, \tag{1.1}$$

where τ_k is the ID of an m-ary *action* of the *recipe*, Q_k is the (hash of) the *joint message* of m smart nodes specified in the action (and agreed between them using their private communication channels) and all $\mu_i^{[k]}$ are the signatures of the corresponding nodes:

$$\mu_i^{[k]} = H(\tau_k \,\big|\big|\, Q_k \,\big|\big|\, C^{[i]}_{2L-2k+1}) \,.$$

Notice that at the time that the value of $\mu_i^{[k]}$ is required to form D_k (step 1) the value of $C^{[i]}_{2L-2k+1}$ is only known to smart node i, so the presence of $\mu_i^{[k]}$ in the digest acts as that node's proper digital signature. Also since the hash function is preimage resistant, knowledge of $\mu_i^{[k]}$ by the rest of the smart nodes involved in the m-ary action does not enable them to forge the signature of smart node i under a different joint message $Q'_k \neq Q_k$ whether individually or jointly. The only way to achieve a digest with all valid signatures is to agree it. This trick, namely publication of a hash with subsequent publication of its preimage after the first publication is assuredly received by the principals is the basis of all Guy-Fawkes type of protocols [2]. The whole swarm is able to instantly validate the digest without requiring any external info at step 6, provided that the IDs of all smart nodes are either directly contained in the m-ary action identified by τ_i, or they are part of the joint message Q_k.

The argument of the hash function in Eq 1.1 is the actual *transaction* between the nodes that signed it, and it is stored in CAS before the Orchestrator receives the digest.

Notice that the validity of the digest does not affect the protocol. If the digest is found to be invalid at the end of the round, it is simply ignored and the process is repeated. A large proportion of invalid digests may trigger denial of service counter-measures, or point to an insider threat, more about it in the sequel.

The above assumes that the role of the Orchestrator is entirely passive. It receives requests to post a joint message on the stack and it fulfils it by forming the digest. However, it can also place its own content on the stack, for example the notification of swarm reconfiguration, or of suspension of execution. This is achieved by forming a special digest

$$D_k = H(0\big|\big|Q_k)\,, \tag{1.2}$$

which has zero as the Action ID and no signature at the end.

1.3.3 Identity

The identity mechanism of smart nodes is based on the hardness of discrete logarithm computation over a large cyclic group. Following the NIST recommendations we use Diffie-Hellman Group 14 [7] as sufficiently large (2048-bit public key) to guard against quantum computing in the next 10 years. Since none of the protocols requires or benefits from Forward Secrecy (i.e. assurance of confidentiality over tens of years), it seems to be appropriate to employ relatively cheap group-theoretical techniques that are only conditionally secure against quantum attacks. Post-Quantum analogues are currently more expensive than the doubling of the key length, with the latter resulting in several tens of years of expected protection.

Each smart node i produces a secret physically random integer $x_i < q$, where $q = (p-1)/2$ and p is Group 14's public modulus; and forms the public key $K_i = 4^x \bmod p$. Physical randomness is usually available from microcontrollers with on-board WiFi facilities. It is achieved by sampling microwave noise. The node identity **Id**$= H(K)$ is stored by the Idenitity/Capability Service (ICS) in Cloud along with the capability list and any other information about the physical characteristics of the device. It is assumed that introduction of new smart nodes to the pool is infrequent, and that there is a periodic schedule of slots in which such events may occur. All coordination-capable smart nodes must synchronise their internal node lists with the ICS at those times to keep consistent. A smart node joining a swarm receives from the Coordinator an up-to-date identity information, provided that the Coordinator identity is known to the node.

Onboarding The lifetime of a recipe (or recipe instance if the same recipe is executed by more than one swarm) begins when the Coordinator is appointed by the Cloud App and the combination (Recipe Id,Coordinator-Id) is signed by the Edge Server. The content of the recipe is stored in the CAS under Recipe Id as the filename, since the Id is simply the hash of the content. The Coordinator establishes the initial list of nodes based solely on the ICS list and the Recipe requirements (which must be matched with the nodes' capabilities). It posts an Announcement on the ICS site (unauthenticated) inviting or ordering nodes to join. Each candidate node posts a join request that consists of:

1. S, Ref to the Announcement
2. I_i, Node's Id
3. $C^{[i]}_{2L-1}$, Node's Winternitz chain top , where L the step limit of the Recipe.
4. A_i, node's private address (need not be unique)
5. μ_i, MAC (Message Authentication Code)

The private address is used on an open unauthenticated side channel that delivers messages to any recipient with a matching private address. The MAC is computed as following:

$$\mu_i = \text{HMAC}(\sigma_{ci}, S\big|\big|I_i\big|\big|C^{[i]}_{2L-1}\big|\big|\mu_i)\,,$$

where HMAC is a hash-based MAC function[8] $\sigma_{ci} = K_c^{x_i} \bmod p$ is a shared secret for the Coordinator and node i, and K_c is the Coordinator's public key. The Coordinator is able to verify that the post is genuine by exploiting the Diffie-Hellman property

$$\sigma_{ci} \equiv K_i^{x_c} \mod p\,,$$

where x_c is the Coordinator's private key and K_i is the node i public one. Note that σ_{ci} requires one modular exponentiation to be made by node i, which may be expensive for a low-power device, but the result is non-ephemeral: it can be used repeatedly to authenticate messages to and from the Coordinator *provided that*, as we assume in our Threat Model, the coordinator is trusted for progress.

Unfortunately, a MAC is not a digital signature and so a third-party identity proof is impossible: at best the third party will be able to assume that the post had been made by no-one else but node i **or** the coordinator itself. Consequently in a zero-trust situation, if the services of node i are engaged, the node has to provide a self-contained public proof of its identity without revealing any secrets. This is achieved by Schnorr's protocol as follows.

Schnorr's protocol. We use a non-interactive version, based on the Fiat-Shamir heuristic [11, 5]. The first posting of any node involved in the recipe is a special identity-proof record, composed as follows.

The node chooses an integer r, $0 < r < q$ at random, then sets

$$u = 4^r \bmod p\,,$$

$$c = H^*(\rho \big|\big| C_{2L-1}^{[i]} \big|\big| K_i \big|\big| u)\,,$$

and

$$z = r + cx \mod q\,.$$

Here ρ is the unique identity of the recipe instance, H^* is a cryptographic hash function with an output in the range $[1, q]$. Let us use square brackets to denote indexing the 256-bit segments of the function result. Then one possible implementation of H^* can be as follows. Given any value a

$$H^*(a)[j] = H(j||a)\,,$$

for $0 \leq j < 8$, where H is the standard SHA-2 hash.

After computing z, the pair (u, z) is sent to the Orchestrator[1] to be posted on the stack. Any third party can then reconstruct c from u and the public parameters, and check that

$$4^z \equiv u K_i^{c \bmod q} \mod p\,.$$

Recall that establishing the node identity requires the node's chain-top to be securely linked with it, because the Orchestrator is not trusted. If the node is the Orchestrator itself, the trust issue does not arise and the formula for c above is replaced by

$$c = H^*(\rho \big|\big| H(E)||K_o||u)\,, \tag{1.3}$$

where E is the fabric edge, and K_o the Orchestrator's public key.

Within the Random Oracle Model (ROM)[3], the security of this protocol for ordinary nodes and the Orchestrator is due to the lack of statistical correlation between c and u. Also note that the value c depends on node i's Wintrernitz chain and the recipe instance identity ρ, so its reuse (a replay attack) by a member of this or another swarm (executing a different recipe instance) is impossible. The security of the protocol also demands that the value of r is destroyed as soon as z has been computed. That value must never be used again in conjunction with the identity of node i.

[1] as mentioned earlier, the content is stored in CAS and its hash is sent to the Orchestrator as the Digest

1.3.4 Launch

Let us summarise the process of starting a recipe on a swarm. As mentioned before, the first step is for Cloud to appoint the Coordinator and provide it with the Recipe Instance, which contains a list of roles with expected capabilities, and a set of actions of different arities. The nature of the actions, the state of the computation that they change and the calculations this may involve are outside the scope of the present document. All we are concerned about here is the fully signed messages that trigger specific actions, their authenticity and nonrepudiation.

All point to point communication is done using direct authenticated messages.

1. The coordinator, as mentioned above on page 10, advertises the swarm on the ICS bulletin board, and waits for requests to join.

2. When it has received sufficient requests to form the swarm, it confirms each accepted request, giving the node its sequential number, Recipe Instance ID and the Orchestrator ID. The Coordinator is node 0 and the Orchestrator is node 1.

3. The Coordinator supplies the Orchestrator with the list of members in the ascending order of their sequential numbers, giving their Ids, addresses and chain-tops. Finally, it provides its own chain-top.

4. The Orchestrator generates its Winternitz fabric F and obtains its Edge E. The Edge E and the array of members' chain-tops are communicated to all members in their membership order; the Coordinator is served first. From this point on, the Winternitz stack is active.

5. The Orchestrator invites the Coordinator to post its Schnorr record (u, z) and runs a round of the protocol.

6. The Orchestrator posts its Schnorr record using the shortened digest detailed in Eqs 1.2, 1.3 and runs a round of the protocol.

7. The Orchestrator invites each member to post their Schnorr record and runs one round for each. At the end of this step, the execution of the Recipe Instance is deemed fully signed.

2. Reputation Protocols

Let us now turn to a very different swarm situation. Coordinated swarm was focused on provable and correct execution of a recipe by the principals duly authorised for such execution, under the control of an Orchestrator and as a swarm formed and periodically altered by Coordinator. Imagine a swarm that act anonymously via a set of intermediaries to collectively evaluate the environment for the purposes of environmental control. For example, a swarm of cars informing smart traffic lights of the prevailing traffic in the vicinity of the site to improve the switching of the traffic flows and reduce congestion. Here none of the smart nodes is able to significantly damage the quality of the control based on its individual reported data; however, the data being reported should not be trusted per se for at least two reasons. Firstly, the quality of the data may be dependent on the equipment of the data source, so nodes may produce data with an accuracy that varies. Secondly, as in the example of smart traffic lights, data sources are also recipients of the control decisions based on their data. Deliberately false data (e.g. reporting a traffic jam that does not exist to gain green light in one's direction of travel) may bring an advantage to a dishonest node. Finally, there is a problem of coincidental disclosure of private data: a car reporting a traffic jam reports its location at the time of the message; this may well be the car owner's private date and as such it is protected by GDPR and similar legislation. So on the one hand, the Evaluator that receives reports from the data sources need to know whether the data can be trusted or not, and on the other, has no right to know the identity of the source.

A nice solution to this problem is provided by the concept of *reputation.* The first zero-trust solution to the best of our knowledge is article [10]. The authors introduced and extensively tested a set of ad hoc formulae that govern:

location distance factor Θ is a value between 0 and 1 which indicates how irrelevant the report may be distance-wise, with 0 indicating close distance and 1 reliably remote.

time sensitivity Ω is similarly a value between 0 and 1, which is 0 for sensing reports that are fresh and 1 for those irrelevant due to the elapsed time being over the limit.

A node reputation is a value between -1 and 1, where -1 means that it is completely not trustworthy; and 1, totally trustworthy. The *sensing report* is sent from a swarm member to a Reputation Server (RS) at a certain location, and the RS uses $Theta$ and $Omega$ to determine the relevance of the sensing report. Since the RS is expected to receive such reports from a number of swarm members that happen to be close in space and time to the object of sensing, the Server is in a position to compare the reports weighted for relevance and to work out the environmental condition. It then scores each individual report r for its reliability based on the closeness or otherwise to the likeliest dataset, and gives it a reputation increment Δ_r.

Each node carries a Reputation Certificate (RC), signed by the Registration Authority. It is sent along with the sensing report. Based on Δ_r and the current reputation R the *reputation coupon* is produced, which is a certificate signed by the RS which contains the current reputation level R and the course-grain increment/decrement (a few, say 5, gradations). The smart node takes that certificate back to the registration authority and the RA updates the node's reputation.

The above is a much simplified scenario, which leaves many unanswered questions. Is the identity of the smart node protected? Can an attacker copy the coupon and redeem it with the RA at the expense of the genuine node? Can a low reputation node's certificate be stolen by a higher reputation node to obtain a larger increment. There are many more, and some of them are answered in paper [10], and the rest are eliminated by certain trust assumptions. We will develop appropriate protocols *ab initio*, bearing in mind that we have our specific threat model for swarm computing and that we aim to minimise the resource footprint to make our solution as generic as possible.

2.1 Blind Signature

Reputation management with the privacy constraints describe above require anonymity. The Reputation Server must not know the identity of the node that has submitted the report; all it needs to know is the node's reputation level certified by the Registration Authority. In issuing the reputation coupon, the RS cannot link it with the identity of the node either, neither explicitly nor implicitly. Finally when the RA updates the node's reputation record, it must have no way of knowing *where* and *when* the coupon was earned.

These security constraints are not easily resolvable. Indeed, the node can never be sure that its identity remains unknown to the signer if it cannot be reassured that the reputation certificate presented to the RS has no data on it that can be used to establish the node's identity. If it can be verified that only the signature is contained in the certificate besides the plaintext info, this makes it usable but automatically makes it transferrable to any other node with the same reputation level. The node cannot borrow a certificate with a lower level of reputation from a colluding node, because the reputation level is copied to the coupon, which would become irredeemable due to the reputation mismatch. However, the sensing report sent alongside the certificate can be modified by an attacker to cause loss of reputation, which is again something that cannot be prevented if the integrity of the session is not maintained by some methods involving trust. Finally, a coupon containing the node's current reputation and the earned increment/decrement needs to be redeemed against the node's identity, and since the latter is not contained in the former, the coupon could be intercepted and redeemed by a rogue node before it is received by the node that earned it.

All of these threats can be countered using the concept of blind and partially blind signature. We define them next.

Definition 1. *A* blind signature scheme *(BSS) is a public tuple (B, U, S, V). It involves a Signer (the signing server) and a Requester (the message source). The scheme operates a publicly known key K_p and a secret key K_s, the latter only available to the Signer. For any message m, the Requester obtains $m' = B(\sigma, K_p, m)$. Here σ is the Requester's secret seed, without which m cannot be restored from m'. Message m' is sent to the Signer, which returns the signature $s' = S(K_s, m')$. The Requester verifies the signature using the public key by checking that $V(K_p, m', s')$ is* true. *Then the Requester obtains $s = U(\sigma, K_p, s')$ and the scheme guarantees that $s = S(K_s, m)$ and so $V(K_p, m, s)$ is also true.*

The above definition of blind signature is easier to understand with the help of a commutative diagram:

$$\begin{tikzcd}
m \arrow[r, "S(K_s,\bullet)"] \arrow[d, "B(\sigma,K_p,\bullet)"'] & s \\
m' \arrow[r, "S(K_s,\bullet)"'] & s' \arrow[u, "U(\sigma,K_p,\bullet)"']
\end{tikzcd}$$

Informally, to obtain the correct signature s out of a message m one can either do it directly by letting the server apply function S, in which case the server learns the message m; **or** one can "blind the message" by applying function β with a secret random σ to obtain message m', and send m' to the server to obtain signature s', in which case the server will not learn the message m. However, the Requester is able to "unblind" the signature s' by applying function U to it using the *same* random σ, which will result in exactly the same s as with the first option.

The blind signature concept allows coupons and certificates to be signed without revealing the identity of the owner to the signing server, and in this sense they are a useful tool for anonymity. Unfortunately, complete blindness has its disadvantages. Ideally coupons and certificates should have an expiry date and other metadata, for example, reputation level or reputation delta, which should be in the clear. If they are not, the requester has no assurance that circumstances of coupon issue are not hidden under the cipher. For example, the server could include the time and location of the report, which would allow the registration authority to correlate those attributes with the node ID, thus obtaining personal data. Consequently even blind signatures would not be sufficient under a zero-trust threat model. There is, however, an extension of Definition 1, which we consider next.[1].

Definition 2. *A* **blind signature scheme with public metadata** *(BSSPM) is a public sextuple* (B, ρ, W, S, F, V)*. It also involves a Signer and a Requester and the two keys* K_p *and* K_s*. For any message* m *the Requester*

1. *obtains* $m' = B(\sigma, K_p, H(m))$ *using a random* σ*, which it keeps secret;*
2. m' *and the processed random value* $\rho(m, \sigma)$ *are sent to the Signer;*
3. *The Signer applies the pee-imager* W *to combine* m' *and* $\rho(m', \sigma)$ *and public metadata* μ *into a preimage* Ω'*;*
4. Ω' *is then encrypted by* S *using the Signer's secret key* K_s *to obtain the pre-signature* $Z' = S(K_s, m')$*, which is sent to the Requester;*
5. *the Requester applies the finalisation function* F *to the pre-signature and produces the signature triplet, which contains the signature proper* s*, the randomisation context* c*, and the public metadata* μ*, assigned by the Signer;*
6. *The Requester then verifies that* $V((s, c, \mu), m)$ *is true.*

The scheme must also guarantee that

1. ρ *does not leak information about* σ *in any significant way, so* m *cannot be obtained from* m' *in practice as long as* σ *is not divulged;*
2. *Even though different choices of* σ *result in different signature triplets (except the public metadata which remains the same), all such triplets pass the verifier* V*;*
3. *any change in the metadata (or rather its hash) will result in* V *failing the signature triplet. Since this is a signature scheme the same is true of* m*.*

The above definition of blind signature with public metadata is easier to understand with the help of a commutative diagram:

$$\begin{tikzcd}
(m, \rho(m, \sigma_0)) \arrow[r, "W(\mu,\bullet)"] \arrow[d, "B(\sigma,K_p,\bullet)"'] & \Omega \arrow[r, "S(K_s,\bullet)"] & Z \arrow[r, "F(\sigma_0,\bullet)"] & (s, c, \mu) \arrow[dr, "V(m,\bullet)"] & \\
(m', \rho(m', \sigma)) \arrow[r, "W(\mu,\bullet)"'] & \Omega' \arrow[r, "S(K_s,\bullet)"] & Z' \arrow[r, "F(\sigma,\bullet)"] & (s', c', \mu) \arrow[r, "V(m,\bullet)"'] & \mathit{true}
\end{tikzcd}$$

Here σ_0 is such that m can be obtained immediately from Ω and μ, i.e. no blinding.

It should be noted that at the core of our chosen BSSPM scheme lies standard asymmetric encryption with a secret key $S(K_s, \bullet)$, that is a standard asymmetric digital signature. This is important because it makes it possible for the requester to verify the pre-signature Z' before finalising the signature with $F(\sigma, \bullet)$, provided that the public metadata μ is available to it: reverse the encryption using the Signer's public key and match Ω' with $W(\mu, m', \rho(m', \sigma))$. In fact, in our chosen BSSPM, the preimager is degenerate:

$$W(\mu, m', \rho(m', \sigma)) = W_1(\mu, W_2(m', \rho(m', \sigma))),$$

with an invertible W_1, which further enables verification of the encryption by matching $W_2(m', \rho(m', \sigma))$ with $W_1^{-1}(\mu, \bullet)$ applied to Ω'. That fortuitous property will be used by ECAS in the next section.

[1] We are interested in randomising blind signature schemes as they make it possible to use low public exponents thus drastically reducing the complexity of cryptographic primitives for a requester. The definition below is not completely general.

2.2 Extended Content-Addressed Storage (ECAS)

In this tripartite scenario with multiple middle agent (the client) the protocol has two choices: (i) to introduce sessions with a temporary ID to maintain the session integrity for each client, bearing in mind that several such sessions can run simultaneously or (ii) introduce a zero-trust secure Cloud-based synchronisation facility. We chose the latter since it avoids any correlation between instances of communication to the extend that messages can be sent by clients via other clients acting on behalf of it, and thus further impeding the identity analysis. Option (ii) also helps to protect the protocol from denial of service attacks assuming that the synchronisation facility operates in the clear and is witnessed. Let us dwell on this in some detail.

Content-Addressed Storage (CAS) is a networked, persistent memory facility which receives datasets and stores them under the dataset hash as its name. By construction, if a retrieval request referring to the name of the dataset, the recipient has a strong assurance (as strong as the second preimage resistance properties of the hash) that the dataset has not been tampered with. The store/retrieve interface is the simplest one for a CAS; we are interested in the asynchronous version of it: publish/subscribe, which, as the name suggests, allows an interested party to subscribe to a certain name and be notified by the CAS that the content has been published (with the content supplied with the notification). When the CAS detects a publish request it checks that it has received the dataset correctly, since the request contains both the data and the name, that is the hash. If the latter matches, the dataset is stored under the name, otherwise the request fails.

The extended CAS (ECAS) that we propose here is a generalisation of the content-hash principle by utilising digital signatures in addition to the hash. The publisher provides attributes that describe the integrity control using some namespace (set of symbols) globally shared by the swarm to set the relebamy attributes. ECAS checks that the dataset and its name pass that integrity control and then it stores the dataset. The subscriber can subscribe to a name and specify some or all of the attributes. Different attribute sets generally mean a different dataset. The attributes provided by the subscriber are matched with those set by the publisher and if the match is established, the rest of the attributes are communicated to the subscriber along with the dataset.

The following request set is proposed:

publish $\mathit{value}[\mathit{attr}]$;

subto $\mathit{tag}[\mathit{attr}] \rightarrow \mathit{variable}$;

subto $\mathit{tag}[\mathit{attr}] \rightarrow \mathit{variable}, \mathit{mdvar}.$

The attributes in brackets are optional are can be in two parts: server and metadata, separated by a';'. The server attribute is either `RA` or `RS`, indicating that the *value* is signed by the corresponding principal. The signature can be either ordinary or with public metadata. In the latter case the text of the metadata follows after the semicolon.

The **publish** request carries the *value* (either plain or signed, depending on the attributes) and its hash as the tag. The ECAS checks the request before approving it:

- If the attributes are omitted, it hashes the *value* and compares the result with the tag. The pair is stored in ECAS if the match is achieved, otherwise the request fails.
- If the attribute RA/RS is specified, but no metadata, the *value* is assumed to have been enciphered with the corresponding server's private key. The ECAS uses the corresponding public key to obtain the plaintext of the *value* and then hashes the the plaintext to match the result with the tag. In the case of a match, it stores the original tag-value pair.
- If the metadata attribute is specified, the ECAS acts the same as in the previous case with *value*, and then it strips the public metadata off the plaintext by applying $W_1^{-1}(\mu, \bullet)$ to it. Then the result is hashed and compared with the tag. If it is a match, the original *value* **and** the metadata are stored under the tag. The reason for hashing without metadata is that the subscriber has no knowledge of it and would not be in a position to prepare the subscription tag it depended on it.

In all cases attributes are included in the dataset stored under the *tag*.

When a **subto** request is received by ECAS, it attempts to find a tag-value pair with the matching *tag*. If one exists, then the attributes, if any, are checked for a match.

- If the request carries the server attribute, but not the metadata, the metadata is considered successfully matched, and the metadata stored under the *tag*, if any, is included in the result dataset.
- If no attribute is provided and the stored attribute set is not empty, the outcome is not a match.

If a match is achieved, the *value* and any *metadata* are immediately returned to the subscriber and the connection is closed . Otherwise the reaction differs between Cloud and swarm. A **subto** request from RA/RS for which the tag has not been published keeps the requester's connection open. Eventually, either the tag is published or the ECAS times out and closes the connection, whichever the sooner, the latter causing the protocol to fail. It is different for a client. The client may not wish to maintain a lasting connection to avoid a correlation leak, so an active wait could be preferable: the client repeats the **subto** request, say every second, until it is fulfilled. How exactly notifications are processed in a specific scenario is up to the networking side of the solution; the security of the protocol does not depend on it.

When the **subto** request is fulfilled, the *variable* specified in it is assigned with the *value* from the tag-value pair, and the *mdvar* parameter is another variable, which is assigned with the metadata stored under the *tag*. The variables can be omitted to indicate that the requester ignores the *value* after validation; this is typical of secure synchronisation.

Validation. Zero-trust is a critical part of the ECAS. Any **subto** request presupposes that the received data will be validated by the requester in *exactly* the same way as the ECAS validates any publish request. We will not mention the validation procedure of the **subto** request explicitly.

2.3 Anonymous Reputation Update Protocol

Before looking into reputation update, we would like to introduce a slightly different form of Winternitz chain, the intention of which will be clearer later on.

Definition 3. *A* Randomised Winternitz Chain (RWC) *with x_0 its end-point is a sequence of pairs (x_i, n_i), $i \in [1, L]$, where n_i are some random nonces, such that*

$$x_k = H(x_{k+1} \big|\big| n_{k+1}), \quad k \in [0, L-1], \tag{2.1}$$

see the grey box in the middle of Fig 2.1.

The difference between that and a standard Winternitz chain is that for a standard chain, knowledge of some x_i implies knowledge of all x_k for $k < i$, whereas with an RWC it would require knowledge of all n_k, $k < i$ as well. This makes it possible to change the verifier mid-chain and not worry about the new verifier's ability to restore all earlier x_k from x_i, provided that all nonces are destroyed immediately after verification via Eq 2.1. Note that, similar to the standard Winternitz chain, an RWC is computed in advance and has a limited length L.

2.3.1 Problem setting

The Anonymous Reputation Update is performed by three principals:

- Registration Authority (RA)
- A swarm node that remains anonymous, which we will call the *client* for short
- Reputation Server (RS)

The client goes through a series of rounds. There can be more than one RS but only one is used in each round for any given client. There can be more than one RA also, but for the time being we will assume there is only one. RA is in Cloud, and RS is an Edge server. The protocol can be run for several clients at the same time, but there is no interaction between different clients in the protocol other than the fact that they all share the RA and all the RS.

Objectives The client registers with the RA. The details of registrations may vary. It is possible that an ID is checked to control access to the reputation environment or that complete anonymity is maintained from the start making the system wide open. At registration the client is assigned the lowest reputation 0 and this is made a part of its registration record. The objective of the protocol is to enable the RS to receive the client's reputation claim and situation report as the client moves in space and time and senses its environment. The RS should then be able to satisfy itself that

1. the client's claimed reputation is correct;
2. the situation report genuinely comes from the same client that has claimed its reputation;

and to work out the relevance and trustworthiness of the report based on the client's reputation and reports from other clients and their reputation[2]. Then the RS will determine the reputation update for the client based on the newly determined relevance and trustworthiness of the report and establish the client's new reputation value[3]. Finally the protocol ensures that the updated reputation will be securely returned to the client in the form of digital coupon and that the client will be able to redeem the coupon.

Above all, the protocol guarantees that the Registration Authority will not be able to determine *where* and *when* the client has visited even though the situation report contains that data (it must contain it to determine its relevance and trustworthiness) and even though the RA and the RS are not trusted not to collude.

Further objectives include

1. to prevent the client from "forgetting" an unfavourable reputation update by destroying the coupon.
2. to minimise computational cost of the protocol for the client (but not the servers) to enable low-power devices to participate in the process of reputation update;
3. to reduce the footprint of the client in the RA memory to impede time-correlation attack on anonymity.

2.3.2 Threat model

The RA and RS may collude to correlate the client's timed position reports, but neither will impede progress by not responding to a valid submission. Clients may go rogue and seek to undermine high-reputation colleagues, either individually or in groups. We assume that all communication with the RS and the RA (the latter after the initial registration) is by anonymous broadcast/bulletin boards. To implement this, the suggestion of a local TOR network is popular in literature, but we will not require such structures if a temporary network address (as in WiFi) is obtained on location every time a report is submitted.

2.3.3 ARU Protocol

The proposed protocol is illustrated in figure 2.1. It proceeds in rounds, each starting with a new reputation certificate being issues and finishing with the reputation update coupon being redeemed. The rounds are secured by the client's RWC, which means that the RA and the client cannot start a new round without finishing the current one, and that no more than one reputation certificate per client is valid at any time.

[2] this is domain specific and is not part of the protocol

[3] this is also outside the protocol

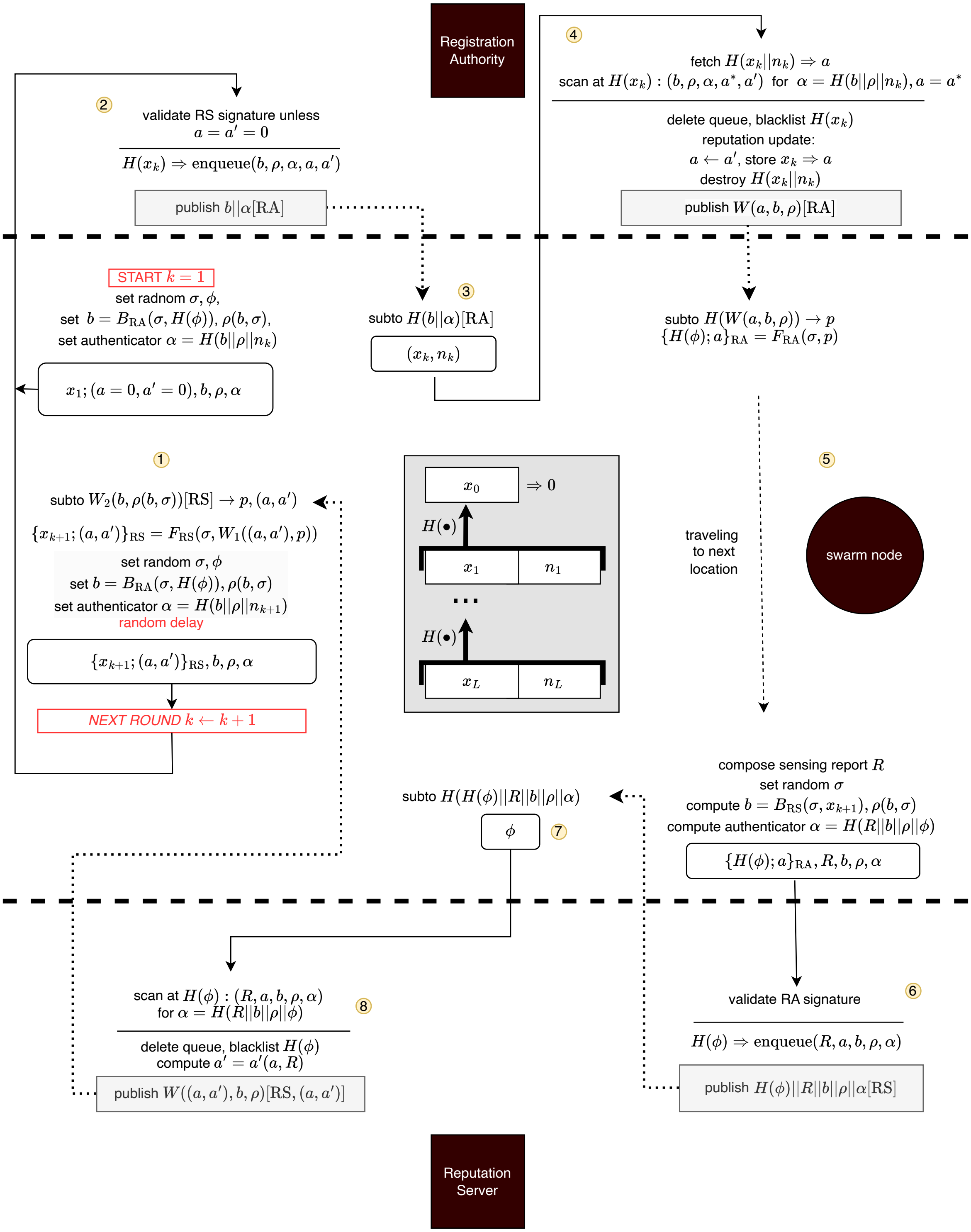
Registration Authority
4
fetch $H(x_k||n_k) \Rightarrow a$
scan at $H(x_k)$: $(b, \rho, \alpha, a^*, a')$ for $\alpha = H(b||\rho||n_k), a = a^*$
delete queue, blacklist $H(x_k)$
reputation update:
$a \leftarrow a'$, store $x_k \Rightarrow a$
destroy $H(x_k||n_k)$
publish $W(a, b, \rho)[\mathrm{RA}]$
2
validate RS signature unless
$a = a' = 0$
$H(x_k) \Rightarrow \mathrm{enqueue}(b, \rho, \alpha, a, a')$
publish $b||\alpha[\mathrm{RA}]$
START $k = 1$
set radnom σ, ϕ,
set $b = B_{\mathrm{RA}}(\sigma, H(\phi))$, $\rho(b, \sigma)$,
set authenticator $\alpha = H(b||\rho||n_k)$
$x_1; (a = 0, a' = 0), b, \rho, \alpha$
3
subto $H(b||\alpha)[\mathrm{RA}]$
(x_k, n_k)
subto $H(W(a, b, \rho)) \rightarrow p$
$\{H(\phi); a\}_{\mathrm{RA}} = F_{\mathrm{RA}}(\sigma, p)$
1
subto $W_2(b, \rho(b, \sigma))[\mathrm{RS}] \rightarrow p, (a, a')$
$\{x_{k+1}; (a, a')\}_{\mathrm{RS}} = F_{\mathrm{RS}}(\sigma, W_1((a, a'), p))$
set random σ, ϕ
set $b = B_{\mathrm{RA}}(\sigma, H(\phi)), \rho(b, \sigma)$
set authenticator $\alpha = H(b||\rho||n_{k+1})$
random delay
$\{x_{k+1}; (a, a')\}_{\mathrm{RS}}, b, \rho, \alpha$
NEXT ROUND $k \leftarrow k + 1$
x_0
$\Rightarrow 0$
$H(\bullet)$
x_1
n_1
$\cdots$
$H(\bullet)$
x_L
n_L
traveling to next location
5
swarm node
compose sensing report R
set random σ
compute $b = B_{\mathrm{RS}}(\sigma, x_{k+1}), \rho(b, \sigma)$
compute authenticator $\alpha = H(R||b||\rho||\phi)$
$\{H(\phi); a\}_{\mathrm{RA}}, R, b, \rho, \alpha$
subto $H(H(\phi)||R||b||\rho||\alpha)$
ϕ
7
scan at $H(\phi)$: (R, a, b, ρ, α)
for $\alpha = H(R||b||\rho||\phi)$
8
delete queue, blacklist $H(\phi)$
compute $a' = a'(a, R)$
publish $W((a, a'), b, \rho)[\mathrm{RS}, (a, a')]$
validate RA signature
6
$H(\phi) \Rightarrow \mathrm{enqueue}(R, a, b, \rho, \alpha)$
publish $H(\phi)||R||b||\rho||\alpha[\mathrm{RS}]$
Reputation Server

Figure 2.1: ARU protocol

Local memory. Both RA and RS use local private tag-value memory. Tags can be blacklisted; any memory operation with a blacklisted tag fails. The value is either a number or a queue of tuples. If the value is a number then we use notation *tag*⇒*value* and define two commands:

store *tag*⇒*value*;

fetch *tag*⇒*variable.*

The operation 'fetch' either succeeds or fails. If succeeds, the *variable* is assigned with the *value*stored under the same *tag.* Two operations are supported for queues:

enqueue a new tuple on the queue: $tag \Rightarrow \textbf{enqueue}(tuple)$;

scan the queue for the first tuple that satisfies a predicate: $\textbf{scan at } tag : (v_1, \ldots, v_m) \textbf{ for } pred(v_1, \ldots, v_m)$.

A queue is created implicitly when the first 'enqueue' operation is executed. The operation 'scan', if succeeds, associates the elements of the satisfying tuple with the variables listed after the colon. Tag-value pairs in memory can be destroyed using the operation

destroy *tag.*

Two kinds of signature are performed by the RA/RS using their private keys and verified by clients with the help of the corresponding certified public keys. The standard signature

$$[message]_{server}$$

and the blind signature with public metadata:

$$\{(blinded)\ message; metadata\}_{server},$$

with both notations denoting the combination of the message and its signature.

Execution. Every step of the protocol is performed by one principal, either a client or the RA or the RS. If a server is the principal for the step, the step is launched when a valid incoming message is received. Validation is step specific and may involve checking associative memory, verifying signatures or both. If the client is the principal, then the execution is triggered by a subscription request with valid data. Here validation follows the procedures defined for the ECAS.

If the message is invalid, it is ignored (but could trigger network traffic analysis). Then the principal computes its output message and sends it to the appropriate server (if the principal is a client) or publishes it on the ECAS (if the principal is a server). One can think of all messages to be broadcast in the plain without any integrity control or checksum, but possibly with FEC. The destination principal will attempt to validate all broadcast messages to find the genuine, uncorrupted one.

A round consists of 8 steps.

Step 1. In round 1, the client sends the initial coupon containing unsigned x_1 and reputation 0/0 (new/old). At this point the client provides its credentials (if access control is in place) and the top of its RWS, x_0, via a side channel. This is all the onboarding that any client needs. In round $k > 1$, at Step 1 the client send the coupon, which contains its x_k blindly signed by the RS with the public metadata (a, a') . In either case the client also sends b, a blinded representation of $H(\phi)$ computed with a fresh random nonce ϕ. This is done using a secret random nonce σ, which does not leave the client, but a function of it $\rho(b, \sigma)$ is included in the message. Finally, the Guy Fawkes authenticator α is computed to link all individual items securely together (more about it at Step 5), using the current RWC nonce n_k as a binding secret. An output message containing $\{x_{k+1}; (a, a')\}_{\mathrm{RS}}, b, \rho, \alpha$ is sent to the RA. More about Step 1 is in the paragraph **Back to Step 1** below.

Step 2. The RA extracts the current a, the awarded a' and the RWC point x_k from the RS-signed item $\{x_k; (a, a^prime)\}$ in the message, as well as the blinded data b and ρ for a new signature and the authenticator α. It validates the RS signature (except for round 1) and enqueues the tuple (b, ρ, α, a, a') under $H(x_k)$, so that it may fetch and validate it at step 4. Finally it enciphers $b||\alpha$ with its private key and publishes it on ECAS.

Step 3. It is separate from Step 2 to fend off the jam-spoof attack, whereby the attacker intercepts the Step 1 message and replaces b, ρ and α there while jamming the original. The known defence [12] is two-phase: (i) the sender confirms that the message has been received intact and by the genuine destination (hence signature on the hash), and (ii) the sender publishes the preimage of the authenticator. The receiver must check the authenticator and then find the *first*valid message that the preimage authenticates. That last action prevents the attacker from intercepting/jamming the preimage and sending a fake message immediately after that: it will not be first, since the receiver has already confirmed by signature the receipt of the first message, and since the sender has verified that the first message was received untampered with before sending the preimage. Hence the need for a queue at Step 2. Now the sender (which is the client in this case) can and should send the preimage n_k (and also x_k to maintain the virtual session).

Step 4. The step starts when the RA receives a valid tuple (x_k, n_k). The tuple is valid when $(x_k, n_k) = x_{k-1}$, received previously, for which the RA has stored a tag-value pair, using x_{k-1} as a tag and some a as the current reputation. At $k = 0$ (for which there is no round) the onboarding procedure placed $x_0 \Rightarrow 0$ in the RA associative storage (possibly after an identity check involving x_0).

Next the queue tagged with $H(x_k)$ is scanned to locate and verify the record placed on it at Step 2. Two conditions are checked: first, the authenticator has to match (to prevent a jam-spoof attack) and secondly the claimed reputation a^* (verified by the RS) should match the latest record a fetched from RA's associative storage. How could it not match?

Imagine a scenario whereby there are two colluding clients C_1 and C_2 with reputations $a_1 > a_2$. Nothing would stop C_1 from swapping their RWC's (by revealing them to each other and using each other's x_{k+1} at Step 5), which would result in C_2 acquiring an unearned higher reputation, a_1. This scenario goes against the objectives of the RS: the server derives situation awareness form adequately weighted clients' reports. While a lowered reputation does not hurt the server much (it only reduces the number of reputable inputs to the averaging algorithm, thus affecting the accuracy slightly), a reputation that has been artificially pumped up reduces the accuracy a great deal more as well as affecting the reputation update of honest clients. The solution that we have adopted is for the RS to include not only the new reputation but also the current one in the coupon. This way, the worst that can happen is that two clients with the same current reputation swap their chains, which is, interestingly, not a threat, since colluding clients can simply swap their sensing reports and that cannot be prevented by any protocol. The sensing report cannot be bound to the sensing platform without revealing (or creating) its identity, which would defeat the purpose of the ARU protocol.

Back to Step 4, the RA deletes and *blacklists* association for tag $H(x_k)$ (the blacklisting is to prevent repay attacks since n_k is now in the clear and since Step 2 can be repeated), updates the reputation by storing $x_k \Rightarrow a$ with the latest a and, to prevent correlation attacks, destroys the previous record by destroying the tag x_{k-1}. The tag destruction is not essential, but helps to obliterate the historic trace of the client in the server memory. We need not mention the destruction of n_k: since it is not written in the persistent storage, it will not survive the current step anyway.

Finally, the server publishes the pre-signature of $H(\phi)$ under its preimage $W(a, b, \rho)$, which the client will be able to subscribe to as it has knowledge of all three parameters for W.

Step 5. The client subscribes to $W(a, b, \rho)$ and fetches the pre-signature into p, then applies the finaliser $F_{\text{RA}}(\sigma, p)$ to unblind the reputation certificate $\{H(\phi); a\}_{\text{RA}}$. Now it can travel to the next location to prepare and submit its anonymous sensing report R there. At that point, the mechanics of a blind signature with public metadata are reinitialised and set in motion. The client sets the new random σ and computes a blinded message b which embodies the next RWC chain point for the RS to sign with current and new reputation of the client as metadata, using the context ρ also supplied. The new reputation will be established by the RS by looking at the sensing report and comparing it with other reports taking into account the time and place of the observations and the reputation of the observing clients.

The RS will sign the coupon blindly and include the public metadata a from the reputation certificate[4] and the newly established reputation a' at step 8.

The client's reputation certificate is bound to the client by the latter's knowledge of the random preimage ϕ, and that knowledge will be reused to bind the report R and the coupon to the certificate. As a reulst

[4] it uses it anyway in calculating the new reputation, since the latter depends on the former

the RS gets the assurance that the report was created by a reputation-a client (of unknown identity, of course) and the client gets the assurance that the coupon cannot be used by any other client even though all communication is in the clear.

The way it is done is via a standard Guy-Fawkes [2] protocol trick. First the client produces a hash of all the entities it wishes to bind to each other and to the preimage that no other principal has the knowledge of at this time. We call the result the GF authenticator or the authenticator for short. Then it gets the counterparty to sign the authenticator and the items it authenticates, publish the signature and put it on a queue. After validating the counterparty signature (to assure itself that all items have been correctly received) the client discloses the preimage. An attacker can learn the preimage and produce its own binding using it, but it will not be received by the counterparty **first**, and so it will be ignored as an instance of the *jam-spoof attack* [12]. A queue, rather than a holding area, is required because the genuine client's preimage can be intercepted and corrupted by an attacker, so it will arrive there potentially after a few attempts. The counterparty must scan the queue for a satisfying record every time a candidate preimage is received, but only the first successful match will matter.

Step 6. The RS receives a message containing five items: the reputation certificate, the sensing report, and the blinded coupon: b and ρ. It validates the certificate and extracts $H(\phi)$ and a from it and enqueues (R, a, b, ρ, α) under $H(\phi)$ to use at Step 8, and publishes signed (by enciphering with its private key) $H(\phi)||R||b||\rho||\alpha$.

Step 7. The client subscribes to the tag $H(H(\phi)||R||b||\rho||\alpha)$ expecting an RS-enciphered value. After validation it is assured that the RS has received all data correctly, so ϕ can now be released to achieve the binding.

Step 8. The RS receives ϕ and locates the first valid record on the $H(\phi)$ queue, unpacks it and deletes the queue. To prevent a replay attack, the tag becomes blacklisted. Then it forms the pre-signature of the coupon $c = [W((a, a'), b, \rho)]_{\text{RA}}$ and publishes it *supplying the metadata* (a, a'). The ECAS tag, as we mentioned before, strips the metadata from the tag, recording the hash of $W_1^{-1}((a, a'), W((a, a'), b, \rho))$ which is the same value as $W_2(b, \rho)$ that the client will subscribe to at the next step.

Back to step 1. The client subscribes to $H(W_2(b, \rho))[\text{RS}]$ and then finalises the content to an anonymous coupon carrying the RWC point x_{k+1} and the reputation pair (a, a') as metadata. Now it restarts the machinery of blind signature by setting a fresh random σ and a new random ϕ, for which it determines $H(\phi)$ and its blinded signature data b and ρ. The coupon along with b, ρ, and α is sent to the RA. At this point the client sets $k \leftarrow k + 1$ to account for the fact that the next round has started.

2.3.4 Blind signature with public metadata: the algorithm

We follow [4]. This is an old publication but the proposal has unique useful properties, which a recently codified competing algorithm [1] does not. Also the algorithm in paper [4] has been critiqued at length, even considered broken by [6], but was vindicated later [15]. This gives us confidence that the algorithm has been sufficiently exposed to the research community to be used in practice. For our purposes we have streamlined [4] in the light of [5] to make it a one-step request/response procedure rather than a two-step one with an extra random parameter.

The algorithm is an extension of RSA and as such assumes a publicly available RSA modulus n of sufficient length (the recommendation at this time is 2000 bits or more). The Signer knows its factorisation into two primes p and q, but no other principal does. The scheme assumes a very low public exponent $e = 3$, but would work with a more standard $e = 2^{16} + 1$ at a higher cost (an order of magnitude). The authors argue that there is no reason to choose the higher e, and the available literature supports their position. The Signer utilises its knowledge of p and q to find d such that $de \equiv 1 \mod (p-1)(q-1)$ and keeps the value of d a secret. It is well known that such d satisfies the following equation for all $m < n$:

$$m^{de} \equiv m \mod n\,.$$

That is why RSA encrypts by raising the plaintext to the power d and decrypts by raising the ciphertext to the power e, both modulo n. Accordingly, (n, e) is the public key and (n, d) the private one. The signature method is just the other way around: the message to be signed is raised to the d by the Signer to obtain the signature, and anybody could verify it by raising it to the power e, both modulo n.

To support the ARU protocol we need to define all the functions mentioned in it.

random nonce σ is defined as a 3-tuple $\sigma = (r, r', u)$, all three random, positive integers less than n;

the blinding function B depends on the public key (e, n) and requires five modular multiplications: $B(\sigma, m) = b = r^e(u^2 + 1)m \mod n$;

the random context ρ is originally defined as $\rho = (rr')^e(u - x) \mod n$, where x statistically independent of b. In the original protocol x would be chosen at random by the Signer. The goal is to thwart the chosen plaintext attack whereby u is chosen together with x so that $(x^2 + 1)/\rho^2 \equiv 1 \mod n$ (see function W_2 next). We rely on [5] to replace a random value by a random oracle: $x = H^*(b)$, which is a cryptographic hash function that has an output of $\lfloor \log_2 n - 1 \rfloor$ bits.

the preimager $W_2(b, \rho)$ Here $b = B(\sigma, m)$, the blinded message. This function is required by the Signer, but also by the Requester, since the latter has to compute the tag for ECAS at Step 1.

$$W_2(b, \rho) = (b(x^2 + 1)/\rho^2)^2 \mod n\,.$$

This involves four modular multiplications and, unfortunately, an inversion $\rho^{-1} \mod n$ which requires the Extended Euclid Algorithm, which has the same cost as proper exponentiation. Fortunately, we can extend the ECAS to publish and then subscribe to ρ with a new attribute [/], and it will be well within the computational power of a Cloud service to do this for an IoT client, see Section 2.3.6. The verification by the client will only take a single modular multiplication.

the preimager $W_1(\mu, \bullet)$ is a simple multiplication: for any p produced by W_2, $W_1(\mu, p) = H(\mu)p \mod n$. The metadata is represented by its hash. Combining both parts of the preimager we obtain

$$W(\mu, b, \rho) = H(\mu)(b(x^2 + 1)/\rho^2)^2 \mod n\,,$$

with the cost of five modular multiplications.

the finaliser $F(\sigma, \bullet)$ The output of the preimager goes through the standard RSA encipherment to obtain the pre-signature

$$Z = H^d(\mu)(b(x^2 + 1)/\rho^2)^{2d} \mod n\,,$$

and the pre-signature is returned to the Requester. The full signature $F(\sigma, Z)$ is a 3-tuple (s, c, μ), where μ is, as above, the public metadata and s and c is computed by the requester thus:

$$c = (rr')^e(ux + 1)/\rho \mod n\,,$$

$$s = r^2 r'^4 Z \mod n\,.$$

Assuming that the multiplicative inverse ρ^{-1} was obtained at the preimage stage, and taking it into account that $(rr')^e$ has been raised to the power e in computing ρ earlier, we find that the finaliser costs six modular multiplications: three to compute c, and three to compute s, since $(rr')^2$ has been obtained in the process of computing $(rr')^e$.

the verifier $V(m, (s, c, \mu))$ establishes the truth of the following congruence:

$$s^e \equiv H(\mu)m^2(c^2 + 1)^2 \mod n\,.$$

taking five modular multiplications.

2.3.5 Ordinary signature

The ARU protocol relies on the Server's ordinary (non-blind, no metadata) signature which may require a different private/public key arrangement, which is undesirable. We propose to use RSA and the same keys we require for the BSSPM, which implies a low RSA public exponent of 3. This makes it particularly important that the IETF recommendation for the use of RSA signatures are taken on board. Specifically, since we typically sign the hash of the document we require to be signed, except when that document *is* a hash, the padding recommendations must be followed, in particular RSASSA-PSS hash/padding mechanism, which provides a secure signature even for a low public exponent, see p.32 in [9].

2.3.6 Ancillary services

A few ancillary services could aid practical implementation of the ARU protocol, especially if low-power devices are planned.

Multiplicative inverse. In Steps 1 and 5 finalisation is performed, which depends on the ability to compute the multiplicative inverse $1/\rho$. The cost of it is similar to modular exponentiation, which works out as up to 2×10^3 modular multiplications, two orders of magnitude above the cost of any function the client has to compute in the course of running the protocol. An easy solution of this problem would be to add an attribute to the ECAS to allow storage of the data for inversion. For example, the request

$$\text{publish}\ \ z[\text{RA}, /]\,,$$

where z is a number would result in ECAS inverting that number using RA's public modulus, and storing it under $H(z)$. The subscriber would then subscribe to $H(z)$ and get $\hat{z} = z^{-1} \mod n_{\text{RA}}$ which will be validated by the requester by checking that $\hat{z} \times z \equiv 1 \mod n_{\text{RA}}$ at the cost of one modular multiplication.

There could be a security concern in that the zero-trust inversion service also makes public the number it has been requested to invert. The remedy is obvious: blinding. Instead of inverting z, the requester should request inversion of some $z_0 z \mod n$, where z_0 is a random nonzero nonce less than the appropriate n. After getting $(z_0 z)^{-1} \mod n$, the requester will simply multiply it by $z_0 \mod n$, thus getting the inverse at a cost of two modular multiplications with perfect confidentiality.

RA distribution. The Reputation Server is placed in Fog, so there will likely be several of those, perhaps sharing one signing server in Cloud. This makes it easier to protect the RS from monitoring attacks as the attacker would need to intercept local communication in several locations to correlate a client's presence to form a fuzzy trajectory.

However, the Registration Authority is in Cloud and it seems to be the nexus of continuity data. If it could be compromised, the adversary would collect all RWC points and their associated reputation changes. The ARU protocol provides a good opportunity to make it harder for the attacker to monitor the RA.

We observe that at Step 1 the client finalises a coupon which contains a fresh x_k. The value is a hash output and as such is pseudorandom. Consider the architecture of the RA whereby 2^ν (virtual) RA servers are installed, each establishing a channel with a signing authority that produces the RSA signature of either the plain or a padded preimage. All clients share a net directory where the 2^ν RA servers are defined together with their network addresses. As a variant, the RA servers may have a different public key each and not use the signing authority at all. At Step 1 the client determines which server to send the request to by taking ν least significant bits of x_k. The same approach is then followed at Step 3 and finally when at Step 4 the RA needs to fetch $x_{k-1} \Rightarrow a$, it first extracts the ν least significant bits form x_{k-1} and passes the request to the corresponding RA. Provided that n_i are not kept for longer than one step (and they are not) by an honest RA, the trace of the client ends there. If some of the RA servers are dishonest, they may collect segments of the client's trajectories, but not more than ν rounds in length, as long as at least one honest RA remains. A nice side-effect of hash-based work distribution, is that the load ballance is statistically even all by itself.

Bibliography

[1] Ghous Amjad, Kevin Yeo, and Moti Yung. RSA blind signatures with public metadata. Cryptology ePrint Archive, Paper 2023/1199, 2023. `https://eprint.iacr.org/2023/1199`. URL: `https://eprint.iacr.org/2023/1199`.

[2] Ross Anderson, Francesco Bergadano, Bruno Crispo, Jong-Hyeon Lee, Charalampos Manifavas, and Roger Needham. A new family of authentication protocols. *SIGOPS Oper. Syst. Rev.*, 32(4):9–20, October 1998. `doi:10.1145/302350.302353`.

[3] Gerrit Bleumer. *Random Oracle Model*, pages 1027–1028. Springer US, Boston, MA, 2011. `doi:10.1007/978-1-4419-5906-5_220`.

[4] Hung-Yu Chien, Jinn-Ke Jan, and Yuh-Min Tseng. Rsa-based partially blind signature with low computation. In *Proceedings. Eighth International Conference on Parallel and Distributed Systems. ICPADS 2001*, pages 385–389, 2001. URL: `https://ieeexplore.ieee.org/document/934844`, `doi:10.1109/ICPADS.2001.934844`.

[5] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*, pages 186–194. Springer, 1986. URL: `https://link.springer.com/content/pdf/10.1007/3-540-47721-7_12.pdf`.

[6] Min-Shiang Hwang, Cheng-Chi Lee, and Yan-Chi Lai. Traceability on rsa-based partially signature with low computation. *Applied Mathematics and Computation*, 145(2):465–468, 2003. URL: `https://www.sciencedirect.com/science/article/pii/S0096300302005003`, `doi:10.1016/S0096-3003(02)00500-3`.

[7] T. Kivinen and M. Kojo. RFC3526: More modular exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). Technical report, RFC Editor, USA, 2003. URL: `https://www.rfc-editor.org/rfc/rfc3526.txt`.

[8] Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997. URL: `https://www.rfc-editor.org/info/rfc2104`, `doi:10.17487/RFC2104`.

[9] Kathleen Moriarty, Burt Kaliski, Jakob Jonsson, and Andreas Rusch. PKCS #1: RSA Cryptography Specifications Version 2.2. RFC 8017, November 2016. URL: `https://www.rfc-editor.org/info/rfc8017`, `doi:10.17487/RFC8017`.

[10] Xinlei Oscar Wang, Wei Cheng, Prasant Mohapatra, and Tarek Abdelzaher. ARTSense: Anonymous reputation and trust in participatory sensing. In *2013 Proceedings IEEE INFOCOM*, pages 2517–2525, 2013. `doi:10.1109/INFCOM.2013.6567058`.

[11] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4:161–174, 1991. URL: `https://link.springer.com/article/10.1007/BF00196725`.

[12] Alex Shafarenko. A PLS blockchain for IoT applications: protocols and architecture. *Cybersecurity*, 4(1):4, 2021. URL: `https://doi.org/10.1186/s42400-020-00068-0`.

[13] Alex Shafarenko. Winternitz stack protocols for embedded systems and IoT. *Cybersecurity*, 7(1):34, 2024. URL: `https://cybersecurity.springeropen.com/articles/10.1186/s42400-024-00225-9`.

[14] Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday paradox for multi-collisions. In Min Surp Rhee and Byoungcheon Lee, editors, *Information Security and Cryptology – ICISC 2006*, pages 29–40, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. URL: `https://link.springer.com/content/pdf/10.1007/11927587.pdf`.

[15] Hsiang-An Wen, Kuo-Chang Lee, Sheng-Yu Hwang, and Tzonelih Hwang. On the traceability on rsa-based partially signature with low computation. *Applied Mathematics and Computation*, 162(1):421–425, 2005. URL: `https://www.sciencedirect.com/science/article/pii/S0096300304001043`, `doi:10.1016/j.amc.2003.12.110`.