BADGER - Blockchain Auditable Distributed (RSA) key GEneRation

Naomi Farley[1], Robert Fitzpatrick[1], and Duncan Jones[1]

Thales UK Limited
www.thalesesecurity.com
{naomi.farley, robert.fitzpatrick, duncan.jones}@thalesesecurity.com

Abstract. Migration of security applications to the cloud poses unique challenges in key management and protection: asymmetric keys which would previously have resided in tamper-resistant, on-premise Hardware Security Modules (HSM) now must either continue to reside in non-cloud HSMs (with attendant communication and integration issues) or must be removed from HSMs and exposed to cloud-based threats beyond an organization's control, e.g. accidental loss, warranted seizure, theft etc.
Threshold schemes offer a halfway house between traditional HSM-based key protection and native cloud-based usage. Threshold *signature* schemes allow a set of actors to share a common public key, generate fragments of the private key and to collaboratively sign messages, such that as long as a sufficient quorum of actors sign a message, the partial signatures can be combined into a valid signature.
However, threshold schemes, while being a mature idea, suffer from large protocol transcripts and complex communication-based requirements. This consequently makes it a more difficult task for a user to verify that a public key is, in fact, a genuine product of the protocol and that the protocol has been executed validly. In this work, we propose a solution to these auditability and verification problems, reporting on a prototype cloud-based implementation of a threshold RSA key generation and signing system tightly integrated with modern distributed ledger and consensus techniques.

1 Introduction

Threshold signature schemes are useful mechanisms for gaining consensus or 'approval' of a message from a set of parties. A (t, n)-threshold signature scheme enables any set of at least $t+1$ parties (out of the specified n) to collaboratively construct a *single* digital signature of a given message M, which can be verified by anyone (who possesses the public verification key). A set of t or fewer parties out of n parties should not be able to construct a valid signature for M.

Informally, in a threshold signature scheme each entity holds a share of a secret signing key; $t + 1$ (or more) parties sign their own copy of the message M under their signature key shares. These partial signatures are then combined to produce a single signature of M, effectively signed using the secret signing

key. Furthermore, some schemes do not require the secret signing key to ever be generated or reconstructed; only shares of the signing key are required to produce valid signatures.

An alternative signature scheme for gaining consensus from a set of parties is a *multi-signature* scheme. Multi-signature schemes require a sufficient number of parties (i.e. $t + 1$) to sign their own copy of a message M, producing a list of signatures, each of which must be verified individually. Thus the main advantage of threshold schemes over multi-signature schemes is that they simplify the verification process: (i) verifiers only have to verify a single digital signature; and (ii) the verifier does not have to check whether each entity is authorised to participate, since only parties with a key share of the private signing key can contribute towards producing valid threshold signatures. Whereas the key generation protocol for threshold signature schemes neccessarily identifies the entities involved (in order to generate key shares for such entities), in multi-signature schemes, any entity can generate their own digital signature key pairs. Thus, as there is no authorisation policy ingrained in the key generation protocol of multi-signature schemes, one must manually check that such signatures have been created by authorised entities.

In the cloud setting, threshold signature schemes would allow a high value private signing key to be distributed between cloud providers, jurisdictions, operating systems, hardware etc. As long as $t+1$ honest devices exist in the system, a threshold scheme can continue functioning, even in the event of seizure, loss, theft of a minority of the fragments. Furthermore, the holder of an insufficient number of signing fragments would be unable to create any valid signatures.

Traditional threshold signature schemes typically rely on a *trusted dealer* to generate and distribute key material to a set of mutually distrusting parties. When no such trusted dealer exists, the set of parties must generate such key material themselves in a distributed manner. Unfortunately, due to lack of trust amongst such parties, lengthy transcripts are generated (containing zero-knowledge proofs, commitments etc.) and must be verified in order for parties to prove that they have behaved honestly and that their resulting key shares are valid. This consequently results in the scheme having a high communication and computation overhead (in comparison to the amount of communication required if all parties were trusted). Such transcripts may be sent through various channels, asynchronously, and can thus be difficult to order and process. In the event of suspected anomalies, agreement between a majority of parties as to the nature of the anomaly is also required, yet the reaching of such agreements is complicated by Byzantine consensus problems. In particular, any entity wanting to audit the entire setup protocol to verify the secure and correct execution of the protocol could potentially face the laborious task of tracing through transcripts of every party involved to discover if/when the protocol went wrong/a malicious party misbehaved. Furthermore, without a root of trust, it may be difficult to achieve consensus regarding which party was to blame for any deviations in the protocol. Throughout this paper, we refer to these hitherto-unaddressed difficulties as 'the auditability problem'.

In practice, verifying gigabytes-worth of protocol transcript to unambigously detect malicious parties proves challenging. Distributed ledger techniques solve many of the encountered problems and are not a gratuitous application of blockchain techniques. We report on the building and testing of a blockchain-auditable system for jointly generating, verifying and using RSA signing keys.

1.1 Our Contribution

In this work, we propose a solution to the auditability problem. We develop a RSA threshold signature system in which all communication between parties in a distributed setup protocol for a (t, n)-threshold signature scheme is recorded using a distributed ledger. Not only does this help parties in the protocol gain assurance that they have a valid partial signing key, but it allows *any* party (e.g. those requesting and verifying threshold signatures) to gain greater assurance that the threshold signing key has been generated correctly and securely, and that resulting threshold signatures are constructed by authorised parties. This is particularly important in order to provide value to any (threshold) signature scheme; without such a mechanism, it is unclear what assurance can be gained from verifying a signature.

Such a system also gives a clear overview of a threshold scheme's setup protocol; by recording all communication, every party can observe what messages every party receives from every other party. Furthermore, recording communication on a distributed ledger means that all such communication is immutable, attributable, non-repuditable and is chronologically sequenced, in contrast to simply sending such transcripts between parties. Use of a distributed ledger therefore make it easier to audit the protocol, and for honest parties to identify and agree any parties that misbehave. Furthermore, using a consensus engine (in this work, we use *Tendermint*), parties may vote on whether they believe a constructed signature is valid; if consensus is reached, the signature is added to the blockchain, allowing 'public' viewing and verification.

In this work, we implement the RSA threshold scheme proposed by Damgård et al. [7], and the distributed RSA setup protocol proposed by Frankel et al. [15] which do not require a trusted dealer. The intention of this work is provide a proof of concept; we expect that the techniques introduced could apply more widely to threshold schemes beyond the one implemented.

1.2 Structure

We provide useful background material and an overview of related work in Section 2. In Section 3 we provide more details on the auditability and consensus problems in standard threshold signature scheme implementations. In Section 4, we formally describe our scheme. In Section 5 we describe our implementation, in Section 6 we report on the performance of the constructed system, while in Section 7 we report on the security properties of the implemented scheme.

2 Background and Definitions

In this section, we provide an overview of relevant background material and useful definitions.

2.1 Background

Threshold signatures were first introduced by Desmedt in [9]. RSA-based threshold signature schemes were first considered by Desmedt and Frankel in [10], who identified some difficulties in constructing an RSA-based threshold signature scheme. Although Frankel et al. [12] and De Santis et al. [25] were the first to fully propose RSA-based threshold schemes, such schemes were not robust. Since these works, several robust RSA schemes have been proposed [2, 6, 11, 17, 24, 27], however, most of these schemes either:

- assume the existence of a trusted dealer;
- make strong assumptions of the RSA modulus (e.g. require it to be a product of safe primes[1]);
- require the signing protocol to be interactive.

Damgård et al. introduce a robust threshold signature scheme in [7] which makes use of the distributed key generation protocol described in [15]. Their scheme is *fully distributed* (does not depend on the existence of a trusted dealer), doesn't require the underlying RSA modulus to be product of safe primes and whose signing protocol is non-interactive. The scheme inherits the robust, efficient, distributed RSA key generation (setup) protocol of Yung et al. [15] (with some additional assumptions/restrictions), which describes how a set of untrusted parties can collaboratively generate key material required for an RSA-based threshold signature scheme. Informally, the key generation protocol described by Yung et al. enables a set of (mutually distrusting) parties to distributively:

- generate an RSA modulus N;
- generate a public verification key for the threshold scheme;
- generate partial signing keys for n parties such that at least $t+1$ signatures of a message M under different partial signing keys are required to construct a signature on M under the threshold signing key;
- generate partial verification keys for n parties such that each party can verify the partial signatures of every other entity.

At each stage during the key generation protocol, parties verify that every other party is behaving honestly, and that their outputs are valid. If, at any stage, a party is found to be cheating, appropriate steps in the protocol must be re-run. Informally, the RSA modulus generation protocol described by Yung et al. [15] comprises the following stages:

1. Each party P_i where $i \in \{1, \ldots, n\}$ randomly chooses p_i, q_i values (from a specified range).

[1] P is a safe prime if $P = 2p + 1$ where p is also prime.

2. Each party P_i distributes a share of their p_i, q_i values to every other party (using Shamir secret sharing techniques [26]).
3. Parties distributively compute $N = pq = (p_1 + \cdots + p_n)(q_1 + \cdots + q_n)$.
4. Parties participate in an interactive protocol to test whether N is a product of exactly two primes[2]. If the result is negative, the protocol is re-run from step 1 until a valid N is found.

The way in which the RSA modulus is generated offers no guarantees that the resulting N candidate is indeed a product of exactly two primes. Indeed, the RSA modulus protocol typically has to be run thousands of times on average (even for a 1024-bit RSA modulus) in order for a successful RSA modulus to be found. Note that in the schemes described by Damgård et al. [7] and Yung et al. [15], the private signing key for the threshold scheme is never constructed, since this would allow any entity to unilaterally sign a message under the secret signing key without the need to gain consensus on the message. Similarly, no entity should be able to learn p or q where $N = pq$ or $\phi(N) = (p-1)(q-1)$.

An alternative RSA modulus generation protocol is described in [8] (originally proposed in [5]), although it is not clear whether this protocol performs better in the average case, than the protocol described in [15]. Damgård et al. also propose a new RSA threshold signature scheme in [6] which removes some assumptions made on the RSA modulus in [7]. Several proactive RSA schemes, which support the refreshing of partial signing keys have been proposed in the literature [14, 13, 24, 2]. Whilst the scheme we consider in this work is not proactive, we hope that future work could consider implementing such schemes.

Whilst alternative threshold schemes exist in the literature (i.e. based on ElGamal, DSS, ECDSA etc.) [4, 10, 16, 18, 19], the majority of such schemes are discrete-log based and thus usually require an interactive signing protocol (as the schemes are randomized) [27], although more recent discrete log-based schemes (based on bilinear pairings) are non-interactive [21, 22]. Again, we stipulate that our work serves to demonstrate the advantages of recording communication in threshold signature schemes using a distributed ledger, and thus should not be dependent on the scheme chosen. As RSA threshold (signature) schemes without a trusted dealer typically require entities to run an extensive setup protocol requiring a lot of communication amongst parties (as the RSA modulus protocol must typically be run multiple times), we believe that a distributed ledger could particularly benefit such schemes.

2.2 Definitions

A *digital signature scheme* $(\mathsf{Keygen}, \mathsf{Sign}, \mathsf{Verify})$ comprises the following three algorithms [3]:

– $(pk, sk) \xleftarrow{\$} \mathsf{Keygen}(1^\lambda)$: a randomised algorithm that takes as input a security parameter 1^λ and outputs a public-private key pair (pk, sk) where sk is the secret signing key, and pk is the public verification key;

[2] Note that this protocol is designed in such a way as to not reveal p, q or $\phi(N)$ to any party.

- $\sigma \xleftarrow{\$} \mathsf{Sign}(sk, M)$: a randomised algorithm that takes as input sk and a message M to be signed and outputs a signature σ.
- $b \leftarrow \mathsf{Verify}(pk, M, \sigma)$: a deterministic algorithm that takes as input pk, a message M and a signature σ. It outputs a bit $b \in \{0, 1\}$.

A digital signature scheme is *correct* if, for every security parameter $\lambda \in \mathbb{N}$, for every key pair (pk, sk) output by Keygen, any message $M \in \{0,1\}^\star$:

$$\Pr[\mathsf{Verify}(pk, M, \mathsf{Sign}(sk, M)) = 1] = 1.$$

Informally, a digital signature scheme is considered *secure* if an adversary cannot forge a valid signature. That is, an adversary cannot find a (M, σ) (which he has not previously seen) such that $\mathsf{Verify}(pk, M, \sigma) = 1$. (A more formal definition of security is given in [3].)

A *(non-interactive) (t, n)-threshold (digital) signature scheme* (Keygen, $\mathsf{Share\text{-}Sign}$, $\mathsf{Share\text{-}Verify}$, $\mathsf{Combine}$, $\mathsf{Partial\text{-}Verify}$, Verify) comprises the following algorithms [21]:

- $(pk, \mathbf{VK}, \mathbf{SK}) \xleftarrow{\$} \mathsf{Keygen}(\mathsf{params}, 1^\lambda, t, n)$: this is an interactive protocol involving n parties $P_1, P_2, \ldots P_n$ that takes as input public parameters params, security parameter 1^λ, integers t, n such that $t \leq n$. It outputs a public verification key pk, a vector $\mathbf{SK} = (sk_1, sk_2, \ldots, sk_n)$ of secret partial signing keys, where each party P_i only knows sk_i, and a vector of public verification keys $\mathbf{VK} = (vk_1, vk_2, \ldots, vk_n)$.
- $\sigma_i \xleftarrow{\$} \mathsf{Share\text{-}Sign}(M, sk_i)$ takes as input a message M and private key share sk_i, and outputs a signature share σ_i.
- $b \leftarrow \mathsf{Share\text{-}Verify}(pk, \mathsf{VK}, M, (\sigma_i, i))$ takes as input the public verification key pk, veriication key vector $\mathbf{VK}$ and a signature share σ_i indexed by i. It outputs a bit $b \in \{0, 1\}$ (1 if the share is deemed valid, and 0 otherwise).
- $(\sigma \cup \ \perp) \leftarrow \mathsf{Combine}(pk, \mathbf{VK}, \{(\sigma_i, i)\}_{i \in S})$ takes as input the public verification key pk, veriication key vector $\mathbf{VK}$ and a set of indexed signature shares $\{(\sigma_i, i)\}_{i \in S}$ where $S \subset \{1, 2, \ldots, n\}$ and $|S| = t + 1$. It outputs a full signature σ or $\perp$ if any of the signature shares σ_i is ill-formed.
- $b' \leftarrow \mathsf{Verify}(pk, M, \sigma)$ takes as input the public verification key pk, message M and full signature σ and outputs a bit $b' \in \{0, 1\}$.

A (t, n)-threshold (digital) signature scheme is *non-interactive* if its signing protocol is non-interactive [21]. That is, entities can create their own partial signatures of a messages (via the $\mathsf{Share\text{-}Sign}$ algorithm) without the need to communicate online with other parties [21]. A partial signature $\sigma_i \xleftarrow{\$} \mathsf{Share\text{-}Sign}(M, sk_i)$ is *correct* if, for all $\lambda, t, n \in \mathbb{N}$ where $t \leqslant n$, all valid params, for all $(pk, \mathbf{VK}, \mathbf{SK})$ output by Keygen:

$$\Pr[\mathsf{Share\text{-}Verify}(pk, \mathsf{VK}, M, (\sigma_i, i)) = 1] = 1.$$

A (t, n)-threshold signature scheme is *correct* if, for all $\lambda, t, n \in \mathbb{N}$ where $t \leqslant n$, all valid params, for all $(pk, \mathbf{VK}, \mathbf{SK})$ output by Keygen:

$$\Pr[\mathsf{Verify}(pk, M, \mathsf{Combine}(pk, \mathbf{VK}, \{(\sigma_i, i)\}_{i \in S})) = 1] = 1.$$

Where $S \subset \{1, 2, \ldots, n\}$, $|S| = t + 1$ and $\{(\sigma_i, i)\}_{i \in S}$ is a set of *correct* partial signatures.

A (t, n)-threshold signature scheme is *robust* [17] if honest parties can still create genuine signatures even in the presence of up to t corrupted/malicious parties. A (t, n)-threshold signature scheme is *unforgeable* (against a chosen message attack) if it is computationally infeasible for an adversary (who can corrupt up to t parties) to compute a valid signature of a message (to which he has not already seen/requested a signature for). Informally, we say that a (t, n)-threshold signature scheme is *secure* if it is robust and unforgeable [27].

2.3 Security Models

A *passive* adversary (also known as an *honest but curious* adversary) is one who may corrupt parties and learn the data that the parties have access to, but behaves honestly in party protocols. Similarly to a passive adversary, a *malicious* (or *active*) adversary corrupts a set of parties to learn the data that the parties have access to. However, unlike a passive adversary, a malicious adversary may misbehave in party protocols (e.g. submits invalid signatures, impersonate other parties etc.), potentially in the hope that such protocols deviate from their expected behaviour or leaks information to the adversary.

A *static* adversary corrupts parties before the threshold scheme is initialised. Informally, a static adversary models an adversary who learns nothing about a threshold scheme (e.g. he does not see its public information, the communication between parties in the protocol etc. prior to corrupting parties), and thus chooses the parties that he corrupts at random. An *adaptive* adversary can corrupt parties at any time (e.g. during the construction of the RSA modulus, secret key setup or signing protocol). Such an adversary may take public information associated to the protocol (e.g. ciphertexts) and the internal states of previously corrupted parties into account when deciding which parties to corrupt (i.e. he may believe that it is more beneficial to corrupt specific parties). This is a stronger type of adversary to model than a static adversary, and thus one could argue that a scheme is more secure if it is secure against an adaptive adversary than just a static adversary. A scheme secure against an adaptive adversary is also secure against a static adversary.

Informally, in a security game featuring a static adversary, the adversary must select up to $t-1$ parties to corrupt before the scheme is setup, whereas an adaptive adversary may corrupt up to $t-1$ parties during the run of the protocol. Many threshold signature schemes in the literature are proven secure against a static adversary [7, 15, 27], though more recent schemes have been proven secure against an adaptive adversary [2, 22, 21]. The RSA threshold schemes we adopt in this paper [7, 15] are secure against *static, malicious* adversaries.

Optimistic Security As mentioned previously, we adopt the distributed RSA modulus generation protocol by Yung et al. [15], which enables parties to detect, at various stages, which party behaved maliciously (if any). Unfortunately,

the protocol features expensive computation and communication overheads at each stage, as parties have to produce proofs that they have behaved honestly, which have to be verified by every other party in an in-line manner. Furthermore, the protocol typically has to be re-run thousands of times before a valid RSA modulus N is generated. The cost of computing these in-line commitments and verifications dwarves the functional components of the protocol by a large margin, due to the repeated use of 'large-exponent' modular exponentiations throughout.

In our system, we thus slightly weaken the security of the RSA key generation protocol described in [15] in order to optimise the performance of our system. In particular we delay verification checks (to check whether parties have generating the RSA modulus correctly) until a successful RSA modulus N is found. That is, parties can check that N values which pass a double primality test are constructed honestly, but cannot detect whether parties behaved dishonestly in the previous runs of the protocol (i.e. whether unsuccessful N candidate values were constructed correctly). As a consequence, this means that we cannot guarantee termination of the protocol (similarly to the scheme proposed in [8]), but we argue that such checks can be re-introduced if the protocol has been run a sufficient number of times without success.

2.4 Tendermint - a consensus engine

As the communication and consensus platform connecting the protocol participants, we use Tendermint [20].

Tendermint is software for securely and consistently replicating an application on many machines. By *securely*, we mean that Tendermint works even if up to $\frac{1}{3}$ of machines fail in arbitrary ways. By *consistently*, we mean that every non-faulty machine sees the same transaction log and computes the same state. Secure and consistent replication is a fundamental problem in distributed systems; it plays a critical role in the fault tolerance of a broad range of applications, from currencies, to elections, to infrastructure orchestration, and beyond.

The ability to tolerate machines failing in arbitrary ways, including becoming malicious, is known as Byzantine fault tolerance (BFT). The theory of BFT is decades old, but software implementations have only became popular recently, due largely to the success of "blockchain technology" like Bitcoin and Ethereum. (Blockchain technology is just a reformalization of BFT in a more modern setting, with emphasis on peer-to-peer networking and cryptographic authentication.) The name 'blockchain' derives from the way transactions are batched in blocks, where each block contains a cryptographic hash of the previous one, forming a chain. In practice, the blockchain data structure actually optimizes BFT design.

Tendermint consists of two chief technical components: a blockchain consensus engine and a generic application interface. The consensus engine, called Tendermint Core, ensures that the same transactions are recorded on every machine in the same order. The application interface, called the Application BlockChain

Interface (ABCI), enables the transactions to be processed in any programming language.

The consensus model. A Tendermint application consists, at any given time, of a set of *validators* which take turns in constructing blocks of transactions, proposing them to other validators and voting on them. Each validator has an associated voting power, with at least $\frac{2}{3}$ of the network-wide voting power necessary to commit a particular block of transactions.

Private channels in Tendermint. In Tendermint, all protocol-level communication between nodes is secured by an authenticated encryption scheme based on the station-to-station protocol. In our scheme, while the majority of messages are intended for all participants, several stages of the protocol involve the secure delivery of (e.g. Shamir shares) to individual parties. At the time of writing, application-level communication between nodes does not benefit from node-specific encryption of messages, hence an additional authenticated encryption scheme was added to the Tendermint application to allow single-recipient encryption.

3 The Auditability Problem

In all such previous systems, verifying the correct execution of the protocol has, at least in theory, been possible, albeit fraught with difficulty in any realistic setting: the number and variety of messages and communication routes in the system greatly complicates the auditing of such a protocol implemented *ad-hoc*, with higher level concerns such as denial of service attacks coming into play.

In the case of a malicious party disrupting the protocol, maybe not by cryptographic machinations but by 'user-level' manipulation, detecting and attributing failures without ambiguity is likely to prove problematic.

In our scheme, by tying the communication, voting and protocol elements into a single distributed ledger, auditing the protocol after execution can be greatly simplified and the source of any divergences pinpointed with comparative ease.

Likewise, while the key generation may take a matter of ten minutes, the resulting blockchain and nodes may lie dormant for indefinite periods of time before resuming operation and consensus once a signing request is submitted by an authorized party. Allowing re-establishment of communication, synchronization, consensus, mutual authentication etc. in these cases are, in the traditional schemes, unaddressed 'higher-level' concerns, whereas in reality have concrete impacts on the security, resilience and usability of any such scheme.

4 The Scheme

As mentioned previously, our scheme combines the threshold key generation scheme of Yung, Frankel and Mackenzie [15] with the scheme of Damgård and

Koprowski [7], which do not assume the existence of a trusted dealer. RSA-based threshold signature schemes typically produce lengthy transcripts, particularly if its underlying RSA modulus N is constructed by a set of mutually distrusting parties. Typically in this situation, the RSA modulus protocol generation protocol must be run multiple times until a valid RSA modulus N is found - such a protocol seems inevitable given that no single entity is trusted to know p or q, let alone generate N. Thus, we argue that our scheme could be particularly useful for auditing RSA-based schemes due to their lengthy setup protocols. Several RSA threshold schemes require the RSA modulus N to be a product of two strong primes; this consequently may slow down the RSA modulus generation protocol significantly as more RSA candidate N values are likely to be rejected than when this assumption is not required. In the schemes considered, N need not be a product of strong prime. Hence for this reason, and because no trusted dealer is assumed, the schemes of Yung, Frankel and Mackenzie [15] with the scheme of Damgård and Koprowski [7] seem reasonable to consider for the purposes of demonstrating the advantages of our system. Future work could consider alternative schemes.

Our scheme operates as follows:

1. *Orchestration and Establishment:* Participating nodes are established by an orchestrator and a genesis file (corresponding to a single key generation) is shared between them, containing a list of public keys of authorized participants
2. *Threshold Key Generation:* On command (by an authorized user), nodes execute the threshold RSA key generation protocol, recording all transactions and commitments in a distributed ledger.
3. *Certificate Generation and Extraction:* The protocol concludes with nodes jointly creating and signing a Certificate Signing Request (CSR), then submitting it to a certificate authority for certificate creation. On successful conclusion of the protocol, the Certificate Authority (CA)-signed certificate is available (for extraction by the user) at the 'tail' of the blockchain, while the nodes hold private signing key fragments
4. *Signing:* On command (by an authorized user), nodes sign a PKCS#1 signing request, publishing their signature fragments to the ledger. A challenge-response protocol proceeds, establishing the validity of the signature fragments, highlighting any compromised nodes. A sufficient quorum of valid signing fragments is (publicly) combined into a valid PKCS#1 signature, made available to the requester on the 'tail' of the blockchain

In more detail:

4.1 Orchestration and Establishment

In our implementation, a system orchestrator creates and deploys the participating nodes to their respective environments, provisioning each with a Tendermint consensus ED25519 key. The nodes, on command, activate their Tendermint

core applications, contacting each other in a peer-to-peer (P2P) fashion, validating each other against the public ED25519 keys contained in the orchestrator-provided genesis file and establishing pairwise bulk communication keys. The orchestrator features a simple certificate authority, the public key of which is known to each participating node. A number of authorised users are created by the CA, each being issued with a user certificate. The authorised users are registered with the key generation nodes.

4.2 Threshold Key Generation

An authorised user creates and signs a 'generate' command, transmitting this to any one (or multiple) of the nodes. This message is validated using an authorised user list (and the corresponding CA-signed user certificate) and, if determined to be valid, key generation commences. The nodes collaborate to generate a workable RSA modulus, followed by generating corresponding signing key fragments.

4.3 Certificate Generation and Extraction

Once the nodes have achieved consensus on an RSA modulus and have verified the correct execution of the protocol, a certificate signing request is jointly created and signed (using the newly-established threshold key), then submitted to the certificate authority. The CA-signed X.509 certificate is then returned to the key requester.

4.4 Signing

With the certificate in hand, the authorised user(s) can proceed to submit PKCS#1 signing requests to the nodes (accompanied by user authorisation signatures), with the nodes using quorums of signing key fragments to produce valid RSA signatures. During each such signature, a node providing a proposed partial signature share is subjected to a challenge-response exercise by the other nodes, allowing the detection of deliberately-false signature shares and the exclusion of now-malicious parties from current future signatures. Each valid threshold signature, in a similar manner to the generated key and certificate, takes the form of a transaction appended to the blockchain, allowing extraction by the requester.

5 Building a Distributed, BFT Threshold RSA Key Generation and Signing System

Our implementation has taken the form of a number of Golang [1] applications, combined with Tendermint nodes. For ease of use, a web frontend application provides an user interface to the system, allowing generation of user keys, threshold key, signatures along with verification of blockchain content. Figure 5 illustrates the high-level architecture of our system.

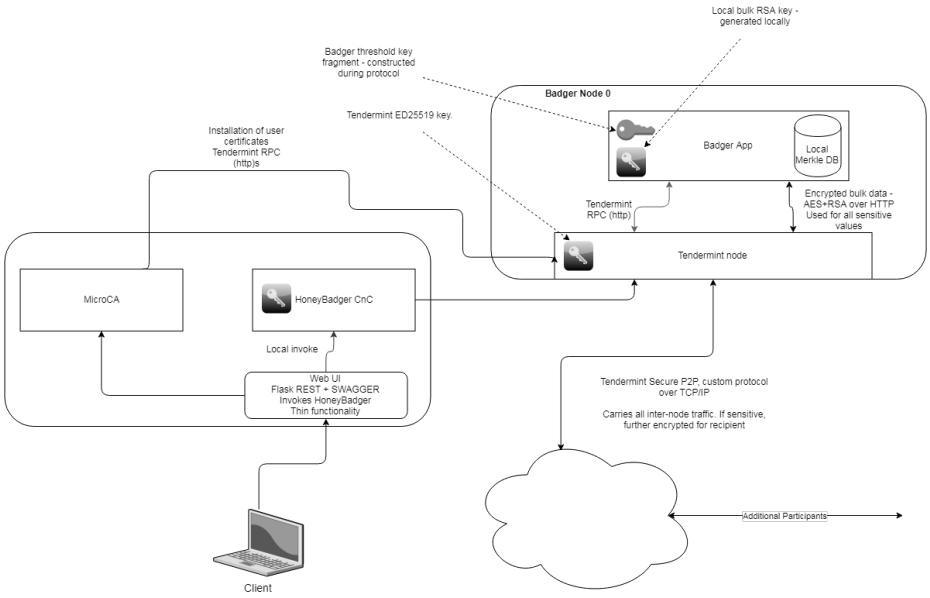

Fig. 1. Our System Architecture (only single threshold node shown)

For testing purposes, our prototype system was deployed on a set of AWS cloud servers (in the same geographical region) and was additionally integrated with a PKCS#11 adapter, allowing usage through Adobe Reader. Thus we were able to demonstrate 'real-world' usability of the system by initiating threshold signatures through the document signing feature of Adobe Reader, with the resulting threshold signatures being returned, verified and accepted by Adobe Reader.

6 Performance Results

Threshold RSA protocols, in trusted-dealer or dealerless form, are inherently slow in comparison to unilateral generation protocols. At the root of this lies a repeated generate-and-test process in which candidate modulii are generated jointly and then tested for compatibility. The vast majority of such candidate modulii are incompatible (i.e. are not the product of two primes), hence a large number of such attempts is required to generate a workable modulus. Several optimisation methods have been proposed [23]. Our implementation incorporates the small-prime test division technique suggested by Malkin et.al [23], which tests whether a candidiate N is divisible by the smallest 'X' primes before the double primality test is carried out on N; if so, it may be discarded before the double primality test is run. Figure 5 illustrates the percentage of bad RSA modulus candidates eliminated using this small-prime test division step, as well as the

number of small primes used for testing against the amortised time per test. The x axis displays the number of small primes (in thousands) tested. Each data point is the result of 10,000 executions for that parameter set. Of course, these timings are implementation and parameter-dependent, but it seems that taking a list of around 2000 small primes provides a reasonably-optimal number.

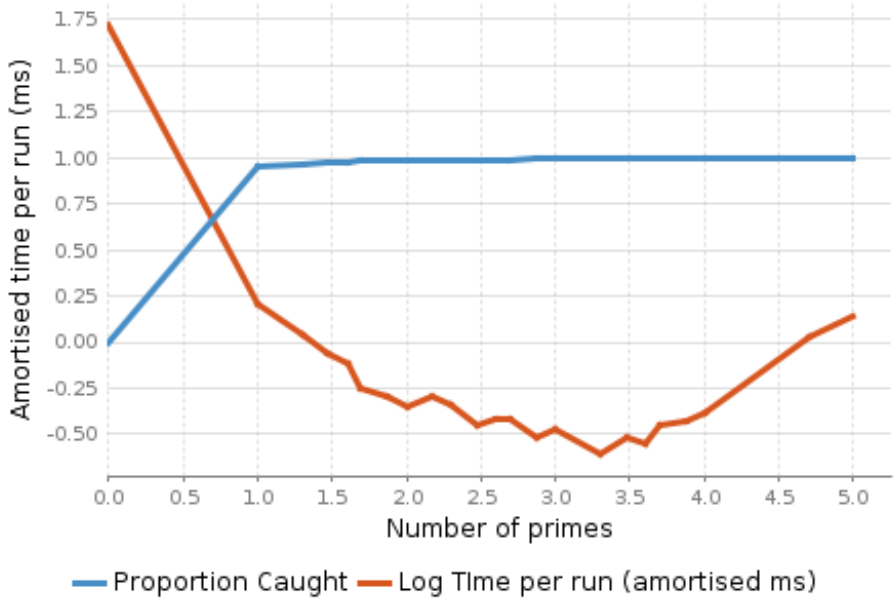

Fig. 2. Eliminating bad RSA modulus candidates using small prime division.

For practical efficiency and for minimizing the number of messages (and hence avoiding accumulated latency), our implementation also takes a batched approach, dealing with (typically) thousands of modulus generation attempts in parallel. In practice, batching modulus generation messages results in very substantial performance benefits, minimising the delays incurred through network and consensus latencies. Future work could also consider implementing additional optimisations, such as the sieving technique discussed in [23].

6.1 Key Generation

By the nature of the protocol, the distribution of key generation running times is dominated by a geometric distribution. Namely, the overwhelming majority of time in the protocol is consumed by nodes repeatedly choosing, sharing, computing on and checking values which allow a (probably) semiprime public modulus to be arrived at. In the case of 1024-bit RSA key generation, the median number of modulus generation attempts required in the $(t, n) = (2, 3)$ case is around

3,500, requiring between 1 and 5 minutes (in the majority of cases) to generate such a key on three separate cloud servers.

6.2 Signing and Partial Signature Combination

In contrast to key generation, signing is a real-time operation, with local signature share creation taking a few milliseconds. Distributed signing and combination, in the distributed setting, takes less than 10 seconds (largely due to Tendermint consensus operations and communication latency), allowing acceptable delays when used in a real-world setting, e.g. from Adobe Reader.

7 Security Properties

Through our use of Tendermint, we introduce additional constraints on the threshold RSA scheme, namely that our scheme, in the $(t, n) = (2, 5)$ case, while only a quorum of three nodes is required to construct a valid signature, the $> \frac{2}{3}$ consensus requirement for Tendermint implies that a minimum of 4 nodes must participate in the construction of a signature, even if one of these nodes is a 'silent bystander', approving transactions. Our implementation followed the 'Optimistic Security' model, namely allowing parties to postpone commitments and verification till a certain point in the protocol, followed by 'replaying' the successful round of the protocol with the same values, but now accompanied by the (expensive) commitments and verifications. In this model, a malicious party is able to disrupt the protocol (carry out a denial of service attack), but in the event that one round of the protocol did succeed, their inability to provide compatible commitments would expose their malicious past behaviour. Thus, if one can accept the possibility of a denial of service attack during key generation, the parties would be assured that proceeding would be safe (even in the presence of malicious and active adversaries) if satisfactory retrospective commitments were provided.

8 Conclusion

We have reported on our development of a threshold RSA key generation and signature system, the first such implementation (to the best of our knowledge) which exploits multiple features of modern distributed ledger and consensus techniques and demonstrates that such use largely solves the consensus and auditability problems which have posed obstacles in the past to the real-world use of threshold signature schemes. Indeed, it seemed interesting to consider modelling an RSA-based scheme due to its lengthy setup process and large communication overhead (mainly due to the way in which an RSA modulus is generated), though one could consider schemes based on alternative hardness assumptions (e.g. discrete log-based schemes). Future work could consider using our system to support proactive RSA schemes in which partial signing keys can be refreshed and/or

the threshold parameters t, n can be modified. Future work could also consider making further performance optimisations (such as those described in [23].) in order improve the performance of our system.

References

1. The Go programming language. https://golang.org. Accessed: 2018-04-09.
2. Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, pages 593–611, 2006.
3. Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. *Ucsd Cse*, 207:207, 2005.
4. Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography, Miami, FL, USA, January 6-8, 2003, Proceedings*, pages 31–46, 2003.
5. Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. *J. ACM*, 48(4):702–722, 2001.
6. Ivan Damgård and Kasper Dupont. Efficient threshold RSA signatures with general moduli and no extra assumptions. In *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, pages 346–361, 2005.
7. Ivan Damgård and Maciej Koprowski. Practical threshold RSA signatures without a trusted dealer. In *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, pages 152–165, 2001.
8. Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, pages 183–200, 2010.
9. Yvo Desmedt. Society and group oriented cryptography: A new concept. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 120–127, 1987.
10. Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 307–315, 1989.
11. Pierre-Alain Fouque and Jacques Stern. Fully distributed threshold RSA under standard assumptions. In *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, pages 310–330, 2001.
12. Yair Frankel and Yvo Desmedt. Parallel reliable threshold multisignature. *Dept. of Elect. Eng. and Computer Sci., Univ. of Wisconsin-Milwaukee, Tech. Rep. TR-92-04-02*, 1992.

13. Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Optimal resilience proactive public-key cryptosystems. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 384–393, 1997.
14. Yair Frankel, Peter Gemmell, Philip D. MacKenzie, and Moti Yung. Proactive RSA. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 440–454, 1997.
15. Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed rsa-key generation. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 663–672, 1998.
16. Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 156–174, 2016.
17. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust and efficient sharing of RSA functions. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, pages 157–172, 1996.
18. Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. *Inf. Comput.*, 164(1):54–84, 2001.
19. Lein Harn. Group-oriented (t, n) threshold digital signature scheme and digital multisignature. *IEE Proceedings-Computers and Digital Techniques*, 141(5):307–313, 1994.
20. Jae Kwon. Tendermint: Consensus without mining. *Retrieved May*, 18:2017, 2014.
21. Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theor. Comput. Sci.*, 645:1–24, 2016.
22. Benoît Libert and Moti Yung. Adaptively secure non-interactive threshold cryptosystems. *Theor. Comput. Sci.*, 478:76–100, 2013.
23. Michael Malkin, Thomas D. Wu, and Dan Boneh. Experimenting with shared generation of RSA keys. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 1999, San Diego, California, USA*, 1999.
24. Tal Rabin. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, pages 89–104, 1998.
25. Alfredo De Santis, Yvo Desmedt, Yair Frankel, and Moti Yung. How to share a function securely. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 522–533, 1994.
26. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
27. Victor Shoup. Practical threshold signatures. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 207–220, 2000.