Distributed SSH Key Management with Proactive RSA Threshold Signatures

Yotam Harchol[1], Ittai Abraham[2], Benny Pinkas[2,3]

[1] UC Berkeley
[2] VMware Research
[3] Bar-Ilan University

Abstract. SSH is a security network protocol that uses public key cryptography for client authentication. SSH connections are designed to be run between a client and a server and therefore in enterprise networks there is no centralized monitoring of all SSH connections. An attractive method for enforcing such centralized control, audit or even revocation is to require all clients to access a centralized service in order to obtain their SSH keys. The benefits of centralized control come with new challenges in security and availability.

In this paper we present ESKM - a *distributed enterprise SSH key manager*. ESKM is a secure and fault-tolerant logically-centralized SSH key manager. ESKM leverages k-out-of-n threshold security to provide a high level of security. SSH private keys are never stored *at any single node*, not even when they are used for signing. On a technical level, the system uses k-out-of-n threshold RSA signatures, which are enforced with new methods that refresh the shares in order to achieve proactive security and prevent many side-channel attacks. In addition, we support password-based user authentication with security against offline dictionary attacks, that is achieved using threshold oblivious pseudo-random evaluation.

ESKM does not require modification in the server side or of the SSH protocol. We implemented the ESKM system, and a patch for OpenSSL libcrypto for client side services. We show that the system is scalable and that the overhead in the client connection setup time is marginal.

1 Introduction

SSH (Secure Shell) is a cryptographic network protocol for establishing a secure and authenticated channel between a client and a server. SSH is extensively used for connecting to virtual machines, managing routers and virtualization infrastructure in data centers, providing remote support and maintenance, and also for automated machine-to-machine interactions.

This work describes a key manager for SSH. Client authentication in SSH is typically based on RSA signatures. We designed and implemented a system called ESKM – a distributed Enterprise SSH Key Manager, which implements and manages client authentication using threshold proactive RSA signatures

Our work focuses on SSH but has implications beyond SSH key management. Enterprise-level management of SSH connections is a known to be a critical problem which is hard to solve (see Sec. 1.1). The solution that we describe is based on threshold cryptography, and must be compliant with the SSH protocol. As such, it needs to compute RSA signatures. Unfortunately, existing constructions for threshold computation of RSA signatures with proactive security, such as [23,22,21], do not tolerate temporary unavailability of key servers (which is a common feature). We therefore designed a new threshold RSA signature protocol with proactive security, and implemented it in our system. This protocol should be of independent interest.

Technical contributions. In addition to designing and implementing a solution for SSH key management, this work introduces the following novel techniques:

- **Threshold proactive RSA signatures with graceful handling of non-cooperating servers:** Threshold cryptography divides a secret key between several servers, such that a threshold number of servers is required to compute cryptographic operations, and a smaller number of servers learns nothing about the key. Threshold RSA signatures are well known [28]. There are also known constructions of RSA threshold signatures with proactive security [23,22,21]. However, these constructions require all key servers to participate in each signature. If a key server does not participate in computing a signature then its key-share is reconstructed and exposed to all other servers. This constraint is a major liveness problem and is unacceptable in any large scale system.
  This feature of previous protocols is due to the fact that the shares of threshold RSA signatures must be refreshed modulo $\phi(N)$ (for a public modulus N), but individual key servers cannot know $\phi(N)$ since knowledge of this value is equivalent to learning the private signature key.
  ESKM solves this problem by refreshing the shares over the integers, rather than modulo $\phi(N)$. We show that, although secret sharing over the integers is generally insecure, it is secure for proactive share refresh of RSA keys.
- **Dynamic addition of servers:** ESKM can also securely add key servers or recover failed servers, without exposing to any key server any share except its own. (This was known for secret sharing, but not for threshold RSA signatures.)
- **Client authentication:** Clients identify themselves to the ESKM system using low-entropy secrets such as passwords. We enable authentication based on threshold oblivious pseudo-random function protocols [20] (as far as we know, we are the first to implement that construction). The authentication method is secure against offline dictionary attacks even if the attacker has access to the memory of the clients and of less than k of the key servers.

1.1 Current SSH Situation

SSH as a security risk. Multiple security auditing companies report that many large scale enterprises have challenges in managing the complexity of SSH keys.

SSH communication security [7] "analyzed 500 business applications, 15,000 servers, and found three million SSH keys that granted access to live production servers. Of those, 90% were no longer used. Root access was granted by 10% of the keys". Ponemon Institute study [6] in 2014 "of more than 2,100 systems administrators at Global 2000 companies found that three out of the four enterprises were vulnerable to root-level attacks against their systems because of failure to secure SSH keys, and more than half admitted to SSH-key-related compromises." It has even been suggested by security analysts at Venafi [8] that one of the ways Edward Snowden was able to access NSA files is by creating and manipulating SSH keys. Recent analysis [34] by Tatu Ylonen, one of the authors of the SSH protocol, based on Wikileaks reports, shows how the CIA used the BothanSpy and Gyrfalcon hacking tools to steal SSH private keys from client machines.

The risk of not having an enterprise level solution for managing SSH keys is staggering. In a typical kill chain the attacker begins by compromising one machine, from there she can start a devastating lateral movement attack. SSH private keys are either stored in the clear or protected by a pass-phrase that is typically no match for an offline dictionary attack. This allows an attacker to gain new SSH keys that enable elevating the breach and reaching more machines. Moreover, since many SSH keys provide root access, this allows the attacker to launch other attacks and to hide its tracks by deleting auditing controls. Finally, since SSH uses state-of-of-the-art cryptography it prevents the defender from having visibility to the attackers actions.

Motivation. A centralized system for storing and managing SSH secret keys has major advantages:

- A centralized security manager can observe, approve and log all SSH connections. This is in contrast to the peer-to-peer nature of plain SSH, which enables clients to connect to arbitrary servers without any control by a centralized authority. A centralized security manager can enforce policies and identify suspicious SSH connections that are typical of intrusions.
- Clients do not need to store keys, which otherwise can be compromised if a client is breached. Rather, in a centralized system clients store no secrets and instead only need to authenticate themselves to the system (in ESKM this is done using passwords and an authentication mechanism that is secure against offline dictionary attacks).

In contrast to the advantages of a central key server, it is also a single point of failure, in terms of both availability and security. In particular, it is obviously insecure to store all secret keys of an organization on a single server. We therefore deploy n servers (also known as "control cluster nodes" – CC nodes) and use k-out-of-n threshold security techniques to ensure that a client can obtain from any k CC nodes the information needed for computing signatures, while any subset of fewer than k CC nodes cannot learn anything useful about the keys. Even though computing signatures is possible with the cooperation of k CC nodes, the private key itself is never reconstructed. Security is enhanced by proactive

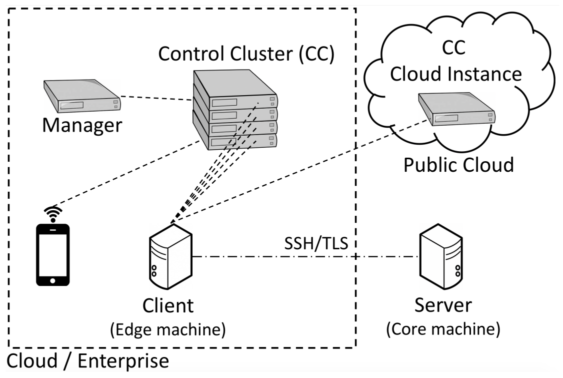

Fig. 1. General system architecture

refresh of the CC nodes: every few seconds the keys stored on the nodes are changed, while the signature keys remain the same. An attacker who wishes to learn a signature key needs to compromise at least k CC nodes in the short period before a key refresh is performed.

Secret key leakage. There are many side-channel attack vectors that can be used to steal keys from servers (e.g., [2,31,24]). Typically, side-channel attacks steal a key by repeatedly leaking little parts of the secret information. Such attacks are one of the main reasons for using HSMs (Hardware Secure Modules). Proactive security reduces the vulnerability to side-channel attacks by replacing the secret key used in each server after a very small number of invocations, or after a short timeout. It cab therefore be used as an alternative to HSMs. We discuss proactive security and our solutions is Sections 2.2 and 3.2. (It is also possible to use both threshold security and HSMs, by having some CC nodes use HSMs for their secret storage.)

Securing SSH. The focus of this work is on securing client keys that are used in SSH connections. Sec. 2.1 describes the basics of the handshake protocol used by SSH. We use Shamir's secret sharing to secure the storage of keys. The secret sharing scheme of Shamir is described in Sec. 2.2. We also ensure security in the face of actively corrupt servers which send incorrect secret shares to other servers. This is done using verifiable secret sharing which is described in Sec. 2.2. The main technical difficulty is in computing signatures using shared keys, so that no server has access to a key neither in computation nor in storage. This is achieved by using Shoup's threshold RSA signatures (Sect. 2.2). We also achieve proactive security, meaning that an attacker needs to break into a large subset of the servers in a single time frame. This is enabled by a new cryptographic construction that is described in Sec. 3.

1.2 ESKM

ESKM (Enterprise SSH Key Manager) is a system for secure and fault-tolerant management of SSH private keys. ESKM provides a separation between the security control plane, and the data plane. The logically-centralized control plane is in charge of managing and storing private keys in a secure and fault-tolerant manner, so that keys are never stored in any single node at any given time. The control plane also provides centralized management services, such as auditing and logging for network-wide usage of secrets, and key revocation.

The general architecture of ESKM is presented in Fig. 1. The control plane is composed of a *security manager* (SM) and a *control cluster* (CC). The ESKM CC is a set of servers that provide the actual cryptographic services to data plane clients. These servers can be located in the same physical site (e.g., a datacenter), in multiple sites, or even in multiple public clouds. These servers can be run in a separate hardened machine or as VMs or a container. They do not require any specialized hardware but can be configured to utilize secure hardware as a secondary security layer.

Threshold cryptography. The ESKM control plane leverages k-out-of-n *threshold* security techniques to provide guarantees for both a high level of security and for strong liveliness. Secrets are split into n shares, where each share is stored on a different control plane node. In order to retrieve a secret or to use it, at least k shares are required ($k < n$). Specifically, in order to sign using a private key, k out of n shares of the private key are used, but the private key itself is never reconstructed, not even in memory, in cache, or in the CPU of any machine. Instead, we use a threshold signature scheme where each node uses its share of the private key to provide a signature fragment to the client. Any k of these fragments are then transformed by the client to a standard RSA signature. Any smaller number of these fragments is useless for an attacker, and in any case, the shares, or the private key, cannot be derived from these fragments.

To protect against rogue admins, each CC node should be managed by a different administrator. Ideally, each CC node can have a different implementation, to reduce the threat of system-wide zero-day exploits. As the communication protocol between ESKM instances is open, ESKM is vendor-neutral with regards to the implementation of each entity in the system. Specifically, different CC nodes can be implemented by different vendors. For enhanced security, some CC nodes can be placed in public clouds, such that at least one (or less than k) of them is required in order to use a key. This way, even if the entire *internal* network is compromised, keys can be immediately revoked by the administrator of the remote machines. As each public cloud hosts less than k CC nodes, the keys are secured even if these remote services are compromised.

Proactive security. ESKM also provides a novel proactive security protocol that refreshes the shares stored on each CC node, such that the shares are randomly changed, but the secret they hide remains the same. This protects against a mobile adversary and side-channel attacks, since keys are refreshed very frequently while on the other hand any successful attack must compromise at least

k servers *before* the key is refreshed. Known constructions of proactive refreshing of threshold RSA signatures are inadequate for our application:

- In principle, proactive refreshing can be computed using generic secure multi-party computation (MPC) protocols. However, this requires quite heavy machinery (since operations over a secret modulus need to be computed in the MPC by a circuit).
- There are known constructions of RSA threshold signatures with proactive security [23,22,21], but these constructions require all key servers to participate in each signature. If a key server does not participate in computing a signature then its key-share is reconstructed by the other servers and is exposed, and therefore this key server is essentially removed from the system. This constraint is a major liveness problem and is unacceptable in any large scale system.

Given these constraints of the existing solutions for proactively secure threshold RSA, we use a novel, simple and lightweight multi-party computation protocol for share refresh, which is based on secret sharing over the integers.

While secret sharing over the integers is generally insecure, we show that under certain conditions, when the secret is a random integer in the range $[0 \ldots R)$ and the number n of servers is small ($n^n \ll R$), then such a scheme is statistically hiding in the sense that it leaks very little information about the secret key. In our application $|R|$ is the length of an RSA key, and the number n of servers is at most a double-digit number. We provide a proof of security for the case where the threshold is 2, and a conjecture and a proof sketch for the general case. Our implementation of proactive secret sharing between all or part of the CC nodes, takes less than a second, and can be performed every few seconds.

Provisioning new servers. Using a similar mechanism, ESKM also allows distributed provisioning of new CC nodes, and recovery of failed CC nodes, without ever reconstructing or revealing the key share of one node.

Minimal modifications to the SSH infrastructure. As with many new solutions, there is always the tension between clean-slate and evolution. With so much legacy systems running SSH servers, it is quite clear that a clean-slate solution is problematic. In our solution there is *no modification* to the server or to the SSH protocol. The only change is in a very small and restricted part of the client implementation. The ESKM system can be viewed as a virtual security layer on top of client machines (whether these are workstations, laptops, or servers). This security layer manages secret keys on behalf of the client and releases the client from the liability of holding, storing, and using multiple unmanaged secret keys. In fact, even if an attacker takes full control over a client machine, it will not be able to obtain the secret keys that are associated with this client.

Abstractly, our solution implements the concept of *algorithmic virtualization*: The server believes that a common legacy single-client is signing the authentication message while in fact the RSA signature is generated via a threshold mechanism involving the client and multiple servers.

Cryptographic mechanisms. The cryptographic mechanisms of ESKM are based on Shamir's secret sharing [27], and its application for RSA threshold signatures by Shoup [28]. In this paper we suggest a novel approach to combine the practical RSA signature scheme of Shoup with a proactive scheme [25] that redistributes the secret key shares every few seconds. Our scheme also redistributes the verification information needed for the non-interactive public signature verification of Shoup.

Implementation and experiments. We fully implemented the ESKM system: a security manager and a CC node, and a patch for the OpenSSL libcrypto for client side services. Applying this patch makes the OpenSSH client, as well as other software that uses it such as `scp`, `rsync`, and `git`, use our service where the private key is not supplied directly but is rather shared between CC nodes. We also implemented a sample phone application for two-factor human authentication, as discussed in Sec. 4.2.

We deployed our implementation of the ESKM system in a private cloud and on Amazon AWS. We show by experiments that the system is scalable and that the overhead in the client connection setup time is up to 100ms. We show that the control cluster is able to perform proactive share refresh in less than 500ms, between the 12 nodes we tested.

Summary of contributions:

1. A system for secure and fault-tolerant management of secrets and private keys of an organization. ESKM provides a distributed, yet logically-centralized control plane that is in charge of managing and storing the secrets in a secure and fault-tolerant manner using k-out-of-n threshold signatures.
2. Our main technical contribution is a lightweight proactive secret sharing protocol for threshold RSA signatures. Our solution is based on a novel utilization of secret sharing over the integers.
3. The system also supports password-based user authentication with security against offline dictionary attacks, which is achieved by using threshold oblivious pseudo-random evaluation (as is described in Sec. 3.4).
4. We implemented the ESKM system to manage SSH client authentication using the standard OpenSSH client, with no modification to the SSH protocol or the SSH server.
5. Our experiments show that ESKM has good performance and that the system is scalable. A single ESKM CC node running on a small AWS VM instance can handle up to 10K requests per second, and the latency overhead for the SSH connection time is marginal.

2 Background

2.1 SSH Cryptography

The SSH key exchange protocol is run at the beginning of a new SSH connection, and lets the parties agree on the keys that are used in the later stages of the SSH protocol. The key exchange protocol is specified in [33] and analyzed in [29,9]. The session key is decided by having the two parties run a Diffie-Hellman key exchange. Since a plain Diffie-Hellman key exchange is insecure against active man-in-the-middle attacks the parties must authenticate themselves to each other. The server confirms its identity to the client by sending its public key, verified by a certificate authority, and using the corresponding private key to sign and send a signature of a hash computed over all messages sent in the key exchange, as well as over the exchanged key. This hash value is denoted as the "session identifier".[4]

Client authentication to the server is described in [32]. The methods that are supported are password based authentication, host based authentication, and authentication based on a public key signature. We focus on public key authentication since it is the most secure authentication method. In this method the client uses its private key to sign the session identifier (the same hash value signed by the server). If the client private key is compromised, then an adversary with knowledge of that key is able to connect to the server while impersonating as the client. Since the client key is the only long-lived secret that the client must keep, we focus on securing this key.

2.2 Cryptographic Background

Shamir's Secret Sharing. The basic service provided by ESKM is a secure storage service. This is done by applying Shamir's polynomial secret sharing [27] on secrets and storing each share on a different nodes. Specifically, given a secret d in some finite field, the system chooses a random polynomial s of degree $k-1$ in that field, such that $s(0) = d$. Each node $1 \leq i \leq n$ stores the share $s(i)$. k shares are sufficient and necessary in order to reconstruct the secret d.

Proactive Secret Sharing. One disadvantage of secret sharing is that the secret values stored at each node are fixed. This creates two vulnerabilities: (1) an attacker may, over a long period of time, compromise more than $k-1$ nodes, (2) since the same shares are used over and over, an attacker might be able to retrieve them by exploiting even a side channel that leaks very little information by using de-noising and signal amplification techniques.

The first vulnerability is captured by the *mobile adversary model*, in this model the adversary is allowed to move from one node to another as long as at most $k-1$ nodes are compromised at any given two-round period [25]. For example, for $k = 2$, the adversary can compromise any single node and in order to move from this node to another node the adversary must have one round in between where no node is compromised.

[4] Security cannot be proved under the sole assumption that the hash function is collision-resistant, since the input to the function contains the exchanged key. In [29] the security of SSH is analyzed under the assumption that the hash function is a random oracle. In [9] it was analyzed under the assumption that the function essentially implements a PRF.

Secret sharing solutions that are resilient to mobile adversaries are called *proactive secret sharing* schemes [19,35]. The core idea is to constantly replace the polynomial that is used for sharing a secret with a new polynomial which shared the same secret. This way, knowing $k-1$ values from each of two different polynomials does not give the mobile attacker any advantage in learning the secret that is shared by these polynomials.

Proactive secret sharing is particularly effective against side-channel attacks: Many side-channel attacks are based on observing multiple instances in which the same secret key is used in order to de-noise the data from the side channel. By employing proactive secret sharing one can limit the number of times each single key is used, as well as limit the duration of time in which the key is used (for example, our system is configured to refresh each key every 5 seconds or after the key is used 10 times).

Feldman's Verifiable Secret Sharing. Shamir's secret sharing is not resilient to a misbehaving dealer. Feldman [13] provides a non-interactive way for the dealer to prove that the shares that are delivered are induced by a degree k polynomial. In this scheme, all arithmetic is done in a group in which the discrete logarithm problem is hard, for example in Z_p^* where p is a large prime.

To share a random secret d the dealer creates a random degree k polynomial $s(x) = \sum_{0 \leq i \leq k} a_i x^i$ where $a_0 = d$ is the secret. In addition, a public generator g is provided. The dealer broadcasts the values $g^{a_0}, \ldots, g^{a_k}$ and in addition sends to each node i the share $s(i)$. Upon receiving $s(i), g^{a_0}, \ldots, g^{a_k}$, node i can verify that $g^{s(i)} = \prod_{0 \leq j \leq k} (g^{a_j})^{i^j}$. If this does not hold then node i publicly complains and the dealer announces $s(i)$. If more than k nodes complain, or if the public shares are not verified, the dealer is disqualified.

Shoup's Threshold RSA Signatures. The core idea of threshold RSA signature schemes is to spread the private RSA key among multiple servers [10,14]. The private key is never revealed, and instead the servers collectively sign the requested messages, essentially implementing a secure multi-party computation of RSA signatures.

Recall that an RSA signature scheme has a public key (N, e) and a private key d, such that $e \cdot d = 1 \mod \phi(N)$. A signature of a message m is computed as $(H(m))^d \mod N$, where $H()$ is an appropriate hash function.

An n-out-of-n threshold RSA scheme can be easily implemented by giving each server a key-share, such that the sum of all shares (over the integers) is equal to d [10,14]. Such schemes, however, require all parties to participate in each signature. This issue can be handled using interactive protocols [15], some with potentially exponential worst case costs [35,26]. These protocols essentially recover the shares of non-cooperating servers and reveal them to all other servers, and are therefore not suitable for a system that needs to operate even if some servers might be periodically offline.

To overcome these availability drawbacks, Shoup [28] suggested a threshold RSA signing protocol based on secret sharing, which provides k-out-of-n reconstruction (and can therefore handle $n-k$ servers being offline). Shoup's scheme is highly practical, does not have any exponential costs, is non-interactive, and provides a public signature verification. (However, it does not provide proactive security.)

We elaborate more on the details of the threshold RSA signature scheme suggested by Shoup: The main technical difficulty in computing threshold RSA is that polynomial interpolation is essentially done in the exponent, namely modulo $\phi(N)$. Polynomial interpolation requires multiplying points of the polynomial by Lagrange coefficients: given the pairs $\{(x_i, s(x_i))\}_{i=1,\ldots,k}$ for a polynomial $s()$ of degree $k-1$, there are known Lagrange coefficients $\lambda_1, \ldots, \lambda_k$ such that $s(0) = \sum_{i=1,\ldots,k} \lambda_i s(x_i)$. The problem is that computing these Lagrange coefficients requires the computation of an inverse modulo $\phi(N)$. However, the value $\phi(N)$ must be kept hidden (since knowledge of $\phi(N)$ discloses the secret key d). Shoup overcomes this difficulty by observing that all inverses used in the computation of the Lagrange coefficients are of integers in the range $[1, n]$, where n is the range from which the indexes x_i are taken. Therefore, replacing each Lagrange coefficient λ_i with $\Delta \cdot \lambda_i$, where $\Delta = n!$, converts each coefficient to an integer number, and thus no division is required.

We follow Shoup's scheme [28] to provide a distributed non-interactive verifiable RSA threshold signature scheme. Each private key d is split by the system manager into n shares using a random polynomial s of degree $k-1$, such that $s(0) = d$. Each node i of the system receives $s(i)$.

Given some client message m to be signed (e.g., a SSH authentication string), node i returns to the client the value

$$x_i = H(m)^{2 \cdot \Delta \cdot s(i)} \mod N,$$

where H is a hash function, $\Delta = n!$, and N is the public key modulus.

The client waits for responses from a set S of at least k servers, and performs a Lagrange interpolation on the exponents as defined in [28], computing

$$w = \prod_i x_i^{2 \cdot \lambda_i^S}$$

where λ_i^S is defined as the Lagrange interpolation coefficient applied to index i in the set S in order to compute the free coefficient of $s()$, multiplied by Δ to keep the value an integer. Namely,

$$\lambda_i^S = \Delta \cdot \frac{\prod_{j \in S \setminus \{i\}} j}{\prod_{j \in S \setminus \{i\}} (j-i)} \in Z$$

The multiplication by Δ is performed in order to cancel out all items in the denominator, so that the computation of λ_i^S involves only multiplications and no divisions.

The result of the interpolation is $w = (H(m))^{4\Delta^2 \cdot d}$. Then, since e is relatively prime to Δ, the client uses the extended Euclidean algorithm to find integers a, b such that $4\Delta^2 a + eb = 1$. The final signature $(H(m))^d$ is computed as

$y = w^a \cdot H(m)^b = (H(m)^d)^{4\Delta^2 a} \cdot (H(m)^{de})^b = (H(m)^d)^{4\Delta^2 a+eb} = (H(m))^d$. The client then verifies the signature by verifying that $H(m) = y^e$ (where e is the public key).

Share verification: Shoup's scheme also includes an elegant non-interactive verification algorithm for each share. This means that the client can quickly detect invalid shares that might be sent by a malicious adversary which controls a minority of the nodes, and use the remaining honest majority of shares to interpolate the required signature. We only describe the highlights of the verification procedure. Recall that an honest server must return $x_i = H(m)^{2\cdot\Delta\cdot s(i)}$, where only $s(i)$ is unknown to the client. The protocol requires the server to initially publish a value $v_i = v^{s(i)}$, where v is a publicly known value. The verification is based on well known techniques for proving the equality of discrete logarithms: The server proves that the discrete log of $(x_i)^2$ to the base $(H(m))^{4\Delta}$, is equal to the discrete log of v_i to the base v. (The discrete log of $(x_i)^2$ is used due to technicalities of the group Z_N^*.) The proof is done using a known protocol of of Chaum and Pedersen [11], see Shoup's paper [28] for details. The important issue for our system is that whenever the shares $s(i)$ are changed by the proactive refresh procedure, the servers' verification values, $v^{s(i)}$, must be updated as well.

Using polynomial secret sharing for RSA threshold signatures gives very good liveliness and performance guarantees that are often not obtainable using comparable n-out-of-n RSA threshold signatures. The main drawback of Shoup's scheme, as well as of all other known polynomial secret sharing schemes for RSA, is that they do not provide an obvious way to implement *proactive security*, which will redistribute the servers shares such that (1) the new shares still reconstruct the original signature (2) the old shares of the servers contain no information that can help in recovering the secret key from the new shares. Proactive security gives guarantees against a mobile adversary and against side channel attacks as discussed in the introduction. We address this drawback and provide a novel proactive scheme for Shoup's threshold signatures in Sec. 3.

3 ESKM Cryptography

In this section we describe the cryptographic novelties behind the ESKM system, for cryptographic signing and for storage services for secret keys. We focus on RSA private keys as secrets, as they are the most interesting use case of ESKM. However, the same techniques can be applied to other secrets as well. ESKM uses Shamir's secret sharing in order to securely split secrets, such that each share is stored on a different CC node. Given a secret d, the ESKM manager creates a random polynomial s over $\phi(N)$ such that $s(0) = d$. It then provides each node i with the value of $s(i)$.

Threshold signatures are computed according to Shoup's protocol. We focus in this section on the new cryptographic components of our construction, which support three new features:

1. Proactive refresh of the shares of the secret key.

2. Recovery and provisioning of new servers (this is done by the existing servers, without the help of any trusted manager).
3. Support for password-based user authentication (with security against offline dictionary attacks).

3.1 Security Model

The only entity in the ESKM system that is assumed to be fully trusted is the system manager, which is the root of trust for the system. However, this manager has no active role in the system other than initializing secrets and providing centralized administrative features. In particular, the manager does not store any secrets.

For the ESKM control cluster nodes (CC) we consider both the semi-honest and malicious models and we provide algorithms for both. In the semi-honest model, up to $f = k - 1$ CC nodes can be subject to offline attacks, side-channel attacks, or to simply fail, and the system will continue to operate securely. In the malicious model we also consider the case of malicious CC nodes that intentionally lie or do not follow the protocol. Note that our semi-honest model is also malicious-abortable. That is, a node which deviates from the protocol (i.e., behaves maliciously) will be detected and the refresh and recovery processes will be aborted, so the system can continue to operate, although without share refreshing and node recovery.

Clients are considered trusted to access the ESKM service, based on the policy associated with their identity. Clients have to authenticate with the ESKM CC nodes. Each authenticated client has a policy associated with its identity. This policy defines what keys this client can use and what secrets it may access. Policies may also include other restrictions such as specific times when a client may or may not use a key, frequency of key usages, etc. We discuss the client authentication issue in Sec. 3.4.

3.2 Proactive Threshold Signatures

In order to protect CC nodes against side-channel and offline attacks, we use a proactive security approach to refresh the shares stored on each CC node. The basic common approach to proactive security is to add, at each refresh round, a set of random *zero-polynomials.* A zero-polynomial is a polynomial z of degree $k - 1$ such that $z(0) = 0$ and all other coefficients are random. Ideally, each CC node chooses a uniformly random zero-polynomial, and sends the shares of this polynomial to all other participating nodes. If only a subset of the nodes participate in the refresh protocol, the value that the zero-polynomial assigns for the indexes of non-participating nodes must be zero. All participating nodes verify the shares they receive and add them, along with the share they produce for themselves, to their original shares. The secret is therefore now shared by a new polynomial which is the sum of the original polynomial $s()$ and the $z()$ polynomials that were sent by the servers. The value of this new polynomial at 0 is equal to $s(0) + z(0) = s(0) + 0 = d$, which is the original secret. This way, while the shares change randomly, the secret does not change as we always add zero to it.

As is common in the threshold cryptography literature, a mobile adversary which controls $k-1$ nodes at a specific round and then moves to controlling $\ell > 0$ new nodes (as well $k-\ell-1$ of nodes that it previously controlled), must have a transition round, between leaving the current nodes and controlling the new nodes, where she compromises at most $k-\ell-1$ nodes. Even for $\ell = 1$ this means that the adversary has at most $k-2$ linear equations of the $k-1$ non-zero coefficients of z. This observation is used to prove security against a mobile adversary.

The difficulty in proactive refresh for RSA: The proactive refresh algorithm is typically used with polynomials that are defined over a finite field. The challenge in our setting is that the obvious way of defining the polynomial z is over the secret modulus $\phi(N) = (p-1)(q-1)$. On the other hand, security demands that $\phi(N)$ must not be known to the CC nodes, and therefore they cannot create a z polynomial modulo $\phi(N)$. In order not to expose $\phi(N)$ we take an alternative approach: Each server chooses a zero polynomial z over the *integers* with very large random positive coefficients (specifically, the coefficients are chosen in the range $[0, N-1]$). We show that the result is correct, and that the usage of polynomials over the integers does not reduce security.

With respect to correctness, recall that for all integers x, s, j it holds that $x^s = x^{s+j\cdot\phi(N)} \mod N$. The secret polynomial $s()$ satisfies $s(0) = d \mod \phi(N)$. In other words, interpolation of this polynomial over the *integers* results in a value $s(0) = d + j\phi(N)$ for some integer j. The polynomial $z()$ is interpolated over the integers to $z(0) = 0$. Therefore, $x^{s(0)+z(0)} = x^{d+j\cdot\phi(N)+0} = x^d \mod N$.

With regards to security, while polynomial secret sharing over a finite field is perfectly hiding, this is not the case over the integers. For example, if a polynomial $p()$ is defined over the positive integers then we know that $p(0) < p(1)$, and therefore if $p(1)$ happens to be very small (smaller than N) than we gain information about the secret $p(0)$. Nevertheless, since the coefficients of the polynomial are chosen at random, we show in Appendix B that with all but negligible probability, the secret will have very high entropy. To the best of our knowledge, this is the first such analysis for polynomial secret sharing over the integers.

The refresh protocol for proactive security. Algorithm 1 presents our share refresh algorithm for the malicious model. This is a synchronous distributed algorithm for n nodes, with up to $f = k-1$ malicious or faulty nodes, where $n = 2f+1$. The dealer provides the initial input parameters to all nodes. Note that verification is done over some random prime field v_p and not over the RSA modulus N ($v_p > N$).

For the semi-honest-malicious-abortable CC nodes model, Round 3 of the algorithm is not necessary anymore, as well as signature validation for verification values (lines 4, 9) and the completion of missing information in line 21.

Proactive refresh of verification information: Secret sharing over the integers allows to refresh the secret shares, but this is not enough. To obtain

Algorithm 1 Malicious Model Share Refresh Algorithm for Node i

```
    Input parameters:
    $s_i$ - current share of node $i$; $N$ - public key modulus;
    $p$ - an upper bound on the coefficients of the zero polynomial(typically, $p = N$);
    $n$ - number of nodes; $f$ - maximal number of faulty nodes ($f = k - 1$)
    $v$ - used as the base for verification of exponents; $v^{s_1}, \ldots, v^{s_n}$ - verification values;
    $v_p$ - verification field; $\mathcal{H}$ - hash function for message signing
    (Note: computations mod $N$, unless noted otherwise)

    Round 1:
 1: Choose $\alpha^i_1, \ldots, \alpha^i_{k-1} \sim U([0,p))$ to create a zero-polynomial $z^i(x) = \sum_{q=1}^{k-1} \alpha^i_q \cdot x^q$
    over the integers.
 2: Compute shares $z^i_1 = z^i(1), \ldots, z^i_n = z^i(n)$
 3: Compute $v^{\alpha^i_1}, \ldots, v^{\alpha^i_{k-1}}$ over $v_p$
 4: Compute $Sig_i \leftarrow \mathcal{H}(v^{\alpha^i_1}, \ldots, v^{\alpha^i_{k-1}})$
 5: for each node $\ell \neq i$ Send $z^i_\ell, (v^{\alpha^i_1}, \ldots, v^{\alpha^i_{k-1}}), Sig_i$ to node $\ell$
    Round 2:
 6: for each received share $z^\ell_i$ from node $\ell$ do
 7:     Verify that $\mathcal{H}(v^{\alpha^\ell_1}, \ldots, v^{\alpha^\ell_{k-1}}) = Sig_\ell$
 8:     Verify that $v^{z^\ell_i} = \prod_{q=1}^{k-1} \left(v^{\alpha^\ell_q}\right)^{i^q} \mod v_p$
 9:     If verification failed then Report node $\ell$
10: end for
11: if verified at least $f + 1$ shares then
12:     Let $s^*_i = s_i + \left(\sum_{\text{verified shares } \ell} z^\ell_i\right)$. (Summation is over the integers)
13:     For each $j$, compute $v^{s^*_j} = v^{s_j} \prod_{\ell=1}^{n} \prod_{q=1}^{k-1} v^{\alpha^\ell_q \cdot j^q} = v^{s_j} \prod_\ell v^{z^\ell_j} \mod v_p$.
14:     Send OK messages to everyone with $Sig_\ell$ of each verified sender $\ell$,
        $(v^{s^*_1}, \ldots, v^{s^*_n})$, and with a report of missing or invalid shares.
15: else Abort
16: end if
    Round 3:
17: Compare signatures in all received OKs
18: Publicly announce everything known by node $i$ on disputed and missing shares to
    everyone (there are up to $f$ such shares)
    Round 4:
19: Complete missing information using information sent in Round 3: Update $s^*_i$, Up-
    date $v^{s^*_1}, \ldots, v^{s^*_n}$, Ignore OKs and shares of identified malicious nodes.
20: if received at least $f + 1$ valid OKs then
21:     Commit new share: $s_i \leftarrow s^*_i$
22:     Commit $v^{s^*_1}, \ldots, v^{s^*_n}$
23: else Abort
24: end if
```

verifiable RSA threshold signatures we also need to refresh the verification information to work with the new shares, as is done in line 19 of the protocol.

Security: The security analysis of the proactive share refresh appears in Appendix B.

3.3 Recovery and Provisioning of CC Nodes

Using a slight variation of the refresh protocol, ESKM is also able to securely recover CC nodes that have failed, or to provision new CC nodes that are added to the system (and by that increase reliability). The process is done without exposing existing shares to the new or recovered nodes, and without any existing node knowing the share of the newly provisioned node.

The basic idea behind this mechanism is as follows: A new node r starts without any shares in its memory. It contacts at least k existing CC nodes. Each one of these existing nodes creates a random polynomial $z()$ such that $z(r) = 0$ and sends to each node i the value $z(i)$ (we highlight again that these polynomials evaluate to 0 for an input r). If all nodes are honest, each node should simply add its original share $s(i)$ to the sum of all $z(i)$ shares it received, and compute $s^*(i) = s(i) + \sum z(i)$. The result of this computation, $s^*()$, is a polynomial which is random except for the constraint $s^*(r) = s(r)$. Node i then sends $s^*(i)$ to the new node r, which then interpolates the values it received and finds $s^*(r) = s(r)$. Since we assume that nodes may be malicious, the algorithm uses verifiable secret sharing to verify the behavior of each node, and it is recommended that the new or recovered node contacts more than k existing nodes.

Algorithm 2 presents the pseudo-code for each existing CC node participating in the recovery process. Algorithm 3 presents the logic of the recovered node.

We note that if this mechanism is used to provision an additional node (as opposed to recovery of a failed node), it changes the threshold to k-out-of-$n+1$. The security implication of this should be taken into account when doing so.

3.4 Threshold-Based Client Authentication

ESKM CC nodes need to verify their clients' identity in order to securely serve them and associate their corresponding policies and keys. However, in order to be authenticated clients must hold some secret that represents their identity, and hence we have a chicken-and-egg problem: Where would this secret be stored?

The adversary model assumes that an adversary might control some CC nodes (but less than k CC nodes), and might have access to the client machine. The adversary must also be prevented from launching an offline dictionary attack against the password.

Human authentication: For human-operated client machines, a two-factor authentication mechanism that uses both a password (as the *something-you-know* factor) and a private key (as the *something-you-have* factor) can be used. However, recall that small coalitions of CC nodes are not trusted, and therefore the password may not be given as-is to each CC node separately. Another approach could be to encrypt the private key using a password and store it at the client or in the CC nodes, but this approach is insecure against offline dictionary attacks on the encrypted file.

A straightforward authentication solution could be to encrypt the private key using a password and store it at the client or in the CC nodes, but since the password might have low entropy this approach is insecure against offline dictionary attacks on the encrypted file. In addition, passwords or hashes of passwords must not be recoverable by small server coalitions.

A much preferable option for password-based authentication is to use a threshold oblivious pseudo-random function protocol (T-OPRF), as suggested in [20]. A T-OPRF is a threshold modification to the concept of an OPRF. An OPRF is a two-party protocol for obliviously computing a pseudo-random function $F_K(x)$, where one party knows the key K and the second party knows x. At the end the protocol the second party learns $F_K(x)$ and the first party learns nothing. (At an intuitive level, one can think of the pseudo-random function as the equivalent of AES encryption. The protocol enables to compute the encryption using a key known to one party and a plaintext known to the other party.) A T-OPRF is an OPRF where the key is distributed between multiple servers. Namely K is shared between these servers using a polynomial p such that $p(0) = K$. The client runs a secure protocol with each of the members of a threshold subset of the servers, where it learns $F_{p(i)}(x)$ from each participating server i. The protocol enables the client to use this data to compute $F_K(x)$. The details of the T-OPRF protocol, as well as its security proof and its usage for password-based threshold authentication, are detailed in [20]. (In terms of availability, the protocol enables the client to authenticate itself after successfully communicating with any subset of the servers whose size is equal to the threshold.)

The T-OPRF protocol is used for secure human authentication as follows: The T-OPRF protocol is run with the client providing a password pwd and the CC nodes holding shares of a master key K. The client uses the protocol to compute $F_K(pwd)$. Note that the password is not disclosed to any node, and the client must run an online protocol, rather than an offline process, to compute $F_K(pwd)$. The value of $F_K(pwd)$ can then be used as the private key of the client (or for generating a private key), and support strong authentication in a standard way. For example, the client can derive a public key from this private key and provide it to the ESKM system (this process can be done automatically upon initialization or password reset). Thus, using this scheme, the client does not store any private information, and solely relies on the password, as memorized by the human user. Any attempt to guess the password requires running an online protocol with the CC nodes. This approach can be further combined with a private key that is stored locally on the client machine or on a different device such as a USB drive, in order to reduce the risk from password theft.

Machine authentication: For automated systems (e.g., scripts running on servers), a client machine must locally store a single private key which authenticates it to the ESKM system. This key can be stored either in main memory or on secure hardware (e.g., Amazon KMS). In terms of costs, this is of course better than storing a massive number of client-server keys in such costly services. In addition, any usage of this single private key is fully and securely audited by the ESKM CC nodes. In an unfortunate case of theft, the key can be immediately revoked without having to log into multiple destination server machines and revoke the key separately on each one of them.

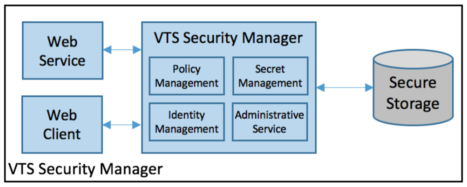

Fig. 2. ESKM Security Manager Architecture

4 ESKM System Design

In this section we describe the design details of the ESKM system, which is presented in Fig. 1. The system includes a logically-centralized control plane, which provides security services, and a data plane, which consumes these services.

4.1 ESKM Control Plane

The ESKM control plane provides security services for network users, whether these are humans or machines. It manages identities, access policies, private keys and secret storage. It also provides centralized auditing and logging capabilities. The control plane is divided into two key parts: the *security manager* (SM) and the *control cluster* (CC).

ESKM Security Manager. The ESKM security manager (SM) is a single (possibly replicated) node that serves as the entry point for all administrative and configuration requests from the system. It manages CC nodes with regards to policy enforcement, storage of secrets, revocation of keys and policies, etc. It is also a central access point for administrators for the purpose of auditing and logging. The SM gives privileged admins the right to read audit logs, but not to delete or prune them (this can be done at each CC node separately).

The SM provides a service for key generation.[5] Upon request, given some key specification, the SM can generate a private key for an identity, and immediately share it with the CC nodes. It then returns the *public key* to the user who requested the generation of the key, but the private key and its shares are deleted from the SM memory. The private key is never revealed or stored on disk.

The only point where the SM actually sees secrets is when secrets are initially stored in the system. In this case, the secret is sent over a secure channel to the SM. The SM immediately splits the secret into shares and sends these shares to the CC nodes. Then, the SM deletes the secret and the shares from its memory. It is important to verify that the data is actually being deleted and overwritten to prevent side-channel attacks on the SM.

[5] The only way to prevent key generation by a single entity is by running a secure multi-party protocol for RSA key generation. However, such protocols, e.g., [18], are too slow to be practical, especially when run between more than two servers, and therefore we did not implement them.

The architecture of the SM is shown in Fig. 2. This is essentially an application server with web interface (REST) and secure (encrypted) storage It has several modules for policy management, identity management, secret handling, and administrative services. It is important to note that the secure storage is not used for secrets. The main usage of this storage in the SM is for public keys of other entities in the network (users, machines, CC nodes) and for the SM's own private keys.

ESKM Control Cluster. The ESKM control cluster (CC) is a set of servers, referred to as "CC nodes". These servers are not replicas. Each CC node implements the CC node specification with regards to the communication protocol. However, each CC node stores different shares of the secrets they protect. In order to add robustness, each CC node can be implemented by a different vendor, run on a different operating system, or a different cryptography library.

A CC node provides two main services: signing, and secret storage and retrieval. The signing service is based on the threshold signatures discussed in Sec. 2 and 3. The storage and retrieval service is based on secret sharing as discussed in Sec. 2. We note that the secure storage shown in the figure is not used for storing shares of clients' secrets. These are only stored in memory, and in case of a CC node failure, the recovery process described in Sec. 3.3 is used in order to recover these shares. It is possible to use, in addition, a hardware secure module (HSM) to store the shares, instead of in memory.

Proactive Share Refresh. The CC nodes have a module that is responsible for executing the share refresh algorithm presented in Sec. 3.2. Specifically, once a refresh policy is set to the SM, the SM sets the corresponding policy arguments to the CC nodes that are included in this policy. A refresh policy has a future start date, duration of a single refresh round, and an interval between each two successive rounds. On the designated start date, each CC node executes Algorithm 1. This repeats every interval as defined in the policy.

A refresh policy also specifies what to do in case of a failure on a refresh round. A failure can be caused by a malicious or faulty node, or by some misconfiguration such as unsynchronized clocks. The available options are to ignore the failure as possible, report the failure and try to continue, report and abort the ongoing round, report and abort all future refresh rounds of this policy, or report and abort the CC node completely.

Secure Recovery and Provisioning. The CC nodes also have a module that is responsible for receiving and responding to recovery requests. Upon receiving such a request, the CC node waits for the next round 5-seconds time in order to make sure all other nodes are also ready, and then it executes the recovery algorithm described in Sec. 3.3. The wait time is configurable and can be much shorter if the network is fast enough. In addition, each CC node web server can initialize a recovery request and send it to the active CC nodes. This is the starting point for the recovery process for that server. It then executes Algorithm 3.

Multi Cloud Security. Putting all of an enterprise compute resource under a single cloud provider, or in a single private cloud, raises the risk that a systemic vulnerability of this cloud provider can compromise the whole compute resource. With threshold signatures it is possible to spread the keys onto multiple cloud providers (e.g., Amazon AWS, Microsoft Azure, Google Cloud Compute). Since each provider holds less than the threshold, then this is information theoretically secure. Moreover, by storing keys on multiple cloud providers we can dramatically decrease the probability of suffering from a systemic security vulnerability.

Auditing. One important feature of ESKM is the ability to provide fault-tolerant network-wide auditing of private key usage. Each CC node keeps track of the requests it handles and the signatures it produces, in a local log system. This log does not contain sensitive information such as shares of private keys.

In order to provide fault-tolerance of up to $f = k - 1$ failures, the SM is allowed to query CC nodes for log entries in order to compose audit reports for administrators and security auditors. Deletion or pruning of CC node logs can only be done by the administrator of a CC node. Thus, even if f nodes are compromised, an attacker cannot wipe their traces by deleting the logs, as each request is served by at least k CC nodes.

This centralized and robust auditing service provides two powerful features. The first feature is the ability to have a system wide view of all SSH sessions, and thus a centralized control and option of activating immediate system-wide user revocation. The second feature is fault-tolerance and threshold security that are provided by implementing the distributed auditing over the CC nodes. An attacker which controls at most $k - 1$ CC servers cannot hide its tracks by manipulating the audit logs in any way.

4.2 ESKM Data Plane

The only modification in the data plane that is required in order to incorporate ESKM is in the SSH client. In most cases, clients use the OpenSSH implementation [4], which uses OpenSSL [5], and specifically its libcrypto library, for RSA signatures. As part of our implementation, discussed in Appendix A, we present a patch for this library that enables ESKM in the client.

Authentication to ESKM CC Nodes. In order to connect to a CC node, a client creates a secure channel on top of which both client and CC node authenticate to each other. The CC node authenticates itself using a certificate, or a private key, whose corresponding public key is known to the client. Client authentication depends on the type of the client: a human or an automated machine.

Client edge machines are operated by humans, while *client core machines* are automated. When using ESKM, a human infiltrator must authenticate to ESKM from an edge machine in order to log into a core machine, and by that to perform a lateral movement to other machines. Thus, by hardening the authentication for edge machines we protect the entire network, allowing core machines to continue operating automatically using public key authentication.

Machine-to-Machine Authentication. Automated clients (core machines) use SSH client private key authentication in order to authenticate with CC nodes. Client certificates are generated by an administrator using an administrative request to the SM. This process can also be automated. These certificates are stored locally in the file system of each client (core machine).

Human Authentication. We employ two-factor authentication for human clients to authenticate with CC nodes. We use password authentication as *something-you-know*, and a private key as *something-you-have.* There are several ways to implement this authentication mechanism and we list the most practical ones here as the decision is based on the pros and cons of each way.

Something You Know. We allow password authentication using the following methods:

• *SSH/HTTPS password Authorization:* Considered less secure as it requires CC nodes to handle password hashing and storage. An attacker can steal passwords by only infiltrating a single CC node.

• *Encrypted Private Key:* The client SSH private key is password-protected so a human user should enter a password or a passphrase in order to load it. This method is also weak as the certificate file can be subject to an offline attack, and it does not allow easy user mobility.

• *Authentication using threshold OPRF:* As discussed in Sec. 3.4, threshold OPRF, as suggested in [20], is a powerful tool for achieving password authentication without exposing the password to the CC nodes. However, since this is a more complex and non-standard authentication protocol, we decided to also support the previous two methods, and give users the ability to configure their installation of ESKM with their preferred method.

Something You Have. For the second factor of authentication, we use RSA private keys. The private key can be installed on the client machine or on a secure USB. Another option is to keep the key on the user's smartphone. When a request from a user arrives, the phone is notified and the user is asked to enter a password, or use their thumbprint, in order to enable the smartphone to perform the RSA signing. The signature is tunneled through a special CC node back to the client machine to complete the authentication. Finally, we have the option to do *multi-device authentication.* We use our threshold mechanism for various k-out-of-n schemes. For example, one share of the key is stored at the client machine and the other is stored at the client smartphone and both devices are needed in order to enable this second factor authentication.

5 Experimental Results

The implementation of the ESKM system is described in Appendix A .

We evaluated our implementation of the ESKM system by deploying it in VMs in a private cloud. Our setup includes 14 VMs: One VM runs the ESKM security manager, twelve VMs serve as ESKM CC nodes, and one VM serves as a local client. Each VM on the private cloud is allocated a single CPU core of type Intel Xeon E5-2680, with clock speed of 2.70 GHz. Most VMs do not share their physical host. We also deploy one CC node on an AWS t2.micro VM.

The client agent performance experiment tests the latency overhead introduced by our client agent, for the execution of the `RSA_sign` function in libcrypto, compared to a standard execution of this function using a locally stored private key. Another measurement we provide is the throughput of the client agent.

ESKM client performance in a private cloud We first use the twelve CC nodes that are deployed in our private cloud. We measure client agent performance as a function of k - the minimal number of CC nodes replies required to construct the signed authentication message. Figure 3 shows the results of this experiment. Even when k is high, the latency overhead does not exceed 100 ms, and the throughput of the client agent does not drop below 19 requests per second. We note that the throughput can be greatly improved using batching techniques, when request frequency is high.

Client performance with a public cloud CC node As mentioned in Sec. 4.1, for enhanced security, CC nodes may also be placed in a public cloud, and one share from these remote CC nodes must be used in order to make a progress. We repeated the previous experiments with a CC node deployed in AWS (t2.micro instance). The additional latency was 103 ms on average.

Client performance with failing CC nodes Figure 4 shows the throughput and latency of the client agent every second over time, when during this time more and more CC nodes fail. After each failure there is a slight degradation in performance. However, these changes are insignificant and the performance remains similar even when most CC nodes fail.

ESKM CC node performance We evaluated the performance of an AWS CC node by measuring the CPU utilization and memory usage of the process, as a function of the number of sign requests it processed per second. Figure 5 presents the results of these measurements: our CC node is deployed on a single-core low-end VM, and is able to handle thousands of sign requests per second without saturating the CPU.

Proactive share refresh We tested our proactive share refresh algorithm implementation to find how fast all 12 CC nodes can be refreshed. Usually, the algorithm requires less than 500 ms to complete successfully. However, in some rare cases this is not enough due to message delays. We set the refresh to be done at least every two seconds, and to limit the length of a single refresh round to at least one second.

CC node recovery We also tested our node recovery algorithm implementation and found that it provides similar performance as the refresh algorithm (this is not surprising as they are very similar). In all our tests, the recovery process required less than 500 ms in order to complete successfully. As for the refresh algorithm, we recommend to use a duration of at least one second to avoid failures that may occur due to message delays.

6 Related Work

Polynomial secret sharing was first suggested by Shamir [27]. Linear k-out-of-k sharing of RSA signatures was suggested by Boyd [10], Frankel [14]. Desmedt and Frankel [12] observed that RSA k-out-of-n threshold signatures is challenging because the interpolation of the shares is over $\phi(n)$. Frankel et al. [15] provided methods to move from polynomial to linear sharing and back. This technique is interactive and not practical.

Rabin [26] provided a simpler proactive RSA signature scheme, using a two layer approach (top is linear, bottom uses secret sharing). This protocol is used in Zhou et al. [35] use in COCA. The scheme leaks information publicly when there is a failure and hence does not seem suitable against a mobile adversary. It also can incur exponential costs in the worst case.

Wu et al. [30] proposed a library for threshold security that provides encryption, decryption, signing, and key generation services. Their scheme is based on additive RSA signatures, and in order to provide threshold properties they use exponential number of shares as in previous additive schemes.

Shoup [28] suggested a scheme that overcomes the interpolation problem, and provides non-interactive verification, that is resilient to an adversary controlling a minority. Gennaro et al. [16] improve Shoup's scheme to deal with large dynamic groups. Gennaro et al. [17] provide constructions for verifiable RSA signatures that are secure in standard models, but require interaction.

Centralized management of SSH keys has recently been the focus of several open source projects: BLESS by Netflix [3], and Vault by Hashicorp [1]. They do not provide threshold signature functionality, but instead resort to the more traditional single node approach.

7 Conclusion

We presented *ESKM*: an Enterprise SSH Key Manager. ESKM advocates a logically-centralized and software-defined security plane that is decoupled from the data plane. By separating the security functionality we can incorporate cutting-edge cryptography in a software defined manner. In particular, the ESKM control plane employs k-out-of-n RSA threshold signatures, verifiable secret sharing against malicious adversaries, and a novel proactive secret sharing method against mobile adversaries and side channel attacks.

Our implementation shows that with minimal changes to the OpenSSL library in the client, one can significantly increase the security of enterprise SSH key management without making any changes to the server SSH deployment. In this sense, ESKM provides a virtual layer of security on top of any existing legacy SSH server implementation. Our experiments show that ESKM incurs a modest performance overhead on the client side. Our implementation of the ESKM control plane is scalable and fault-tolerant, and is able to proactively refresh the shares of CC nodes in a distributed way every few seconds.

References

1. "Hashicorp Vault," https://github.com/hashicorp/vault.
2. "Heartbleed bug," http://heartbleed.com.
3. "Netflix Bless," https://github.com/Netflix/bless.
4. "OpenSSH," https://www.openssh.com/.
5. "OpenSSL," https://www.openssl.org/.
6. "Ponemon report," https://www.venafi.com/assets/pdf/Ponemon_2014_SSH_Security_Vulnerability_Report.pdf.
7. "SSH report," https://www.ssh.com/iam/ssh-key-management/.
8. "Venafi report," https://www.venafi.com/blog/deciphering-how-edward-snowden-breached-the-nsa.
9. F. Bergsma, B. Dowling, F. Kohlar, J. Schwenk, and D. Stebila, "Multi-ciphersuite security of the secure shell (SSH) protocol," in *Proc. of the 2014 ACM Conference on Computer and Communications Security*, 2014, pp. 369–381.
10. C. Boyd, "Digital multisignatures," *Cryptography and coding*, 1986.
11. D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *CRYPTO '92*, ser. LNCS, vol. 740. Springer, 1992, pp. 89–105.
12. Y. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO '89*. Springer-Verlag, 1990, pp. 307–315.
13. P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *FOCS '87*, 1987, pp. 427–438.
14. Y. Frankel, "A practical protocol for large group oriented networks," in *EUROCRYPT '89*. Springer-Verlag New York, Inc., 1990, pp. 56–61.
15. Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung, "Optimal resilience proactive public-key cryptosystems," in *FOCS '97*, 1997, pp. 384–393.
16. R. Gennaro, S. Halevi, H. Krawczyk, and T. Rabin, "Threshold rsa for dynamic and ad-hoc groups," in *EUROCRYPT'08*. Springer-Verlag, 2008, pp. 88–107.
17. R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk, "Robust and efficient sharing of rsa functions," *J. Cryptol.*, vol. 20, no. 3, pp. 393–393, Jul. 2007.
18. C. Hazay, G. L. Mikkelsen, T. Rabin, and T. Toft, "Efficient rsa key generation and threshold paillier in the two-party setting." in *CT-RSA*. Springer, 2012, pp. 313–331.
19. A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *CRYPTO '95*. London, UK, UK: Springer-Verlag, 1995, pp. 339–352.
20. S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu, "Toppss: Cost-minimal password-protected secret sharing based on threshold oprf," Cryptology ePrint Archive, Report 2017/363, 2017, http://eprint.iacr.org/2017/363.
21. S. Jarecki and N. Saxena, "Further simplifications in proactive RSA signatures," in *TCC 2005*, ser. Lecture Notes in Computer Science, J. Kilian, Ed., vol. 3378. Springer, 2005, pp. 510–528.
22. S. Jarecki, N. Saxena, and J. H. Yi, "An attack on the proactive RSA signature scheme in the URSA ad hoc network access control protocol," in *Proc. of the 2nd ACM Workshop on Security of ad hoc and Sensor Networks, SASN*, 2004, pp. 1–9.

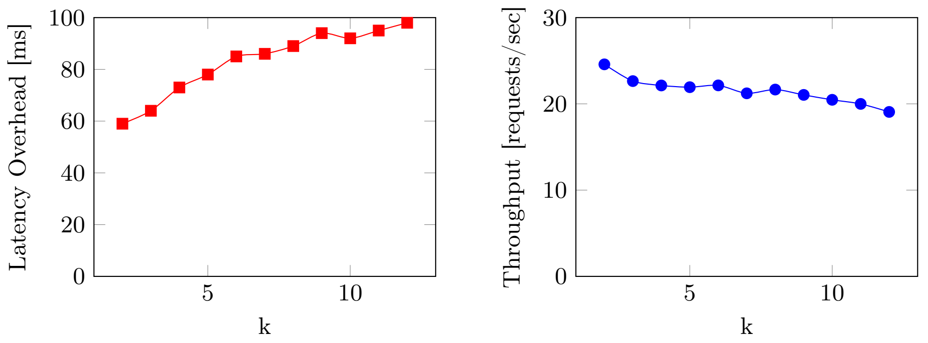

Fig. 3. Client agent performance with $n = 12$ CC nodes, as a function of the threshold parameter k.

23. J. Kong, P. Zerfos, H. Luo, S. Lu, and L. Zhang, "Providing robust and ubiquitous security support for manet," in *ICNP*, 2001.
24. F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy, SP 2015*. IEEE Computer Society, 2015, pp. 605–622.
25. R. Ostrovsky and M. Yung, "How to withstand mobile virus attacks (extended abstract)," in *PODC '91*. New York, NY, USA: ACM, 1991, pp. 51–59.
26. T. Rabin, "A simplified approach to threshold and proactive rsa," in *CRYPTO '98*. Springer-Verlag, 1998, pp. 89–104.
27. A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979.
28. V. Shoup, "Practical threshold signatures," in *EUROCRYPT'00*. Berlin, Heidelberg: Springer-Verlag, 2000, pp. 207–220.
29. S. C. Williams, *Analysis of the SSH Key Exchange Protocol*, 2011, pp. 356–374.
30. T. D. Wu, M. Malkin, and D. Boneh, "Building intrusion-tolerant applications," in *USENIX Security*, 1999.
31. Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 719–732.
32. T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Authentication Protocol," Internet Requests for Comments, RFC 4252, 2004.
33. ——, "The Secure Shell (SSH) Transport Layer Protocol," Internet Requests for Comments, RFC 4253, 2004.
34. T. Ylonen, "Bothanspy & Gyrfalcon - analysis of CIA hacking tools for SSH," August 2017, https://www.ssh.com/ssh/cia-bothanspy-gyrfalcon.
35. L. Zhou, F. B. Schneider, and R. Van Renesse, "Coca: A secure distributed online certification authority," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 329–368, Nov. 2002.

A Implementation

We implemented the ESKM system. In this section we describe the different components of our implementation.

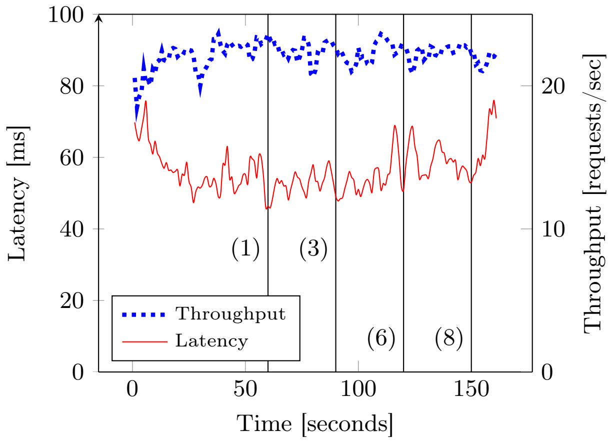

Fig. 4. Client performance with $k = 4, n = 12$. CC node failures are marked at the black vertical lines. Numbers in () are the total number of failed CC nodes.

ESKM Library The ESKM library is a Java library that provides the core cryptographic mechanisms discussed in this paper, such as secret sharing, verifiable secret sharing, threshold signatures, and proactive refresh of shares. The core logic of the security manager, CC nodes, and clients, is implemented in this library, as well as unit tests for the cryptography logic. The library consists of about 5100 lines of code.

ESKM CC Node Our CC nodes are implemented using the ESKM library for the core logic. The CC node implementation is mainly a web server and storage (using Java KeyStore). The web server exposes a REST API over HTTPS. Our CC node code implements signing and storage services, as well as proactive refresh of the shares in the semi-honest malicious-abortable model, and dynamic provisioning and recovery of nodes. This code is about 600 code lines on top of the ESKM library code.

ESKM Security Manager The Security Manager is a relatively thin server with KeyStore storage and a web server for REST API. The SM is about 420 lines of code on top of the ESKM library code.

ESKM Client Agent The client agent is a daemon process running in the background on client machines, maintains persistent HTTPS connections to CC nodes and listens to requests from local processes. The agent distributes requests to CC nodes, then integrates their responses and returns the results to requesting process over the TLS channel. We evaluate the performance of this agent in Sec. 5. The code is about 800 lines long.

ESKM Patch for OpenSSL libcrypto We introduce a patch of about 40 lines of code for the OpenSSL libcrypto library that provides RSA signing logic to various applications, among them OpenSSH. This patch identifies the case where an application tries to sign using a private key, but the private key is

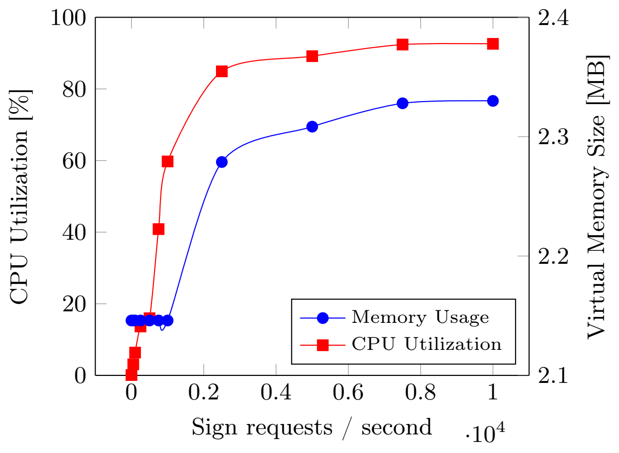

Fig. 5. CC node utilization under increasing load.

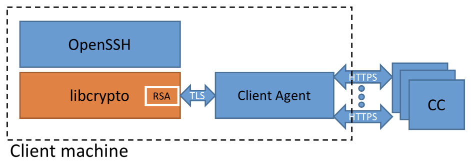

Fig. 6. Client side architecture of ESKM: We provide a patch to OpenSSL's libcrypto that outsources the RSA signature process to our client agent, which communicates with ESKM CC.

not present (for SSH, we create an identity file with only the public information in it). In this case the code calls our client agent (using a TLS/TCP socket), providing it with the message to sign and information about the session. Then it receives back the RSA signature and continues as before.

ESKM Smartphone Application for Two-Factor Authentication We implemented a sample web-based smartphone application for human two-factor authentication. The server-side of this application is provided by an extension to our CC node implementation. This designated CC node has the same API as before, as well as some additional methods for the phone to retrieve a private key, retrieve updates on authentication requests, and send responses to such requests. API calls that require the private key (e.g., `sign`), hang until the user approves them on her phone.

The phone application is initialized with a private key. It periodically polls the designated CC node for any pending request. If a request exists, its details are displayed on the phone screen, and the user is asked for approval. If the user approves, the application sends to the CC node a response for the corresponding request. This response is returned to the client, which then uses it in order to authenticate to other CC nodes.

Smartphone authentication can also be applied on the threshold signature process. To do that, the SM randomly selects d_1 such that $d_2 = d - d_1$. d_1 is shared and distributed to the regular CC nodes. d_2 is sent to the designated CC node. Each request for signature is now also sent to that CC node, which delegates it to the corresponding user's phone. The client agent constructs $H(m)^{d_1}$ as it does already and then multiply this value by $H(m)^{d_2}$, which is returned from the phone, to obtain $H(m)^d$.

In the future, we plan to create a standalone application where the private key is stored in the phone's secure storage (e.g., iOS KeyChain) and users approve requests using their thumbprint.

B Security of Secret Sharing Over the Integers

The proactive share refresh is implemented using polynomial-based secret sharing over the integers. Unlike secret sharing over a finite field, secret sharing over the integers does not provide perfect security. Yet, since in our application the shares are used to hide long keys (at least 4096 bits long), then revealing a small number of bits about the key might be harmless.

The effect of leaking a small number of bits: It is unknown how to efficiently use the knowledge of a small number of bits of an RSA private key in order to break RSA. However, even in the worst case, leaking σ bits about the secret key can only speed up attacks on the system by a factor of 2^σ: Namely, any algorithm A that breaks the system in time T given σ bits about the secret key can be replaced by an algorithm A' that breaks the system in time $2^\sigma \cdot T$ given only the public information about the system and no information about the key. The new algorithm A' simply goes over all options for the leaked bits and runs A for each option.

The degradation of security that is caused by leaking σ bits can therefore be mitigated by replacing the key length ($|N|$) that was used in the original system (with no leakage), by a slightly longer key length which is sufficiently long so that the best known attacks against the new key length are at least 2^σ times slower than the attacks against the original shorter key length. (Therefore, the effect of the leakage, which is at most speeding up the new attack time by 2^σ, is equivalent to using the original key length.)

Analyzing the amount of information that is leaked: In the rest of this section we state a theorem about the amount of information that is leaked about the secret key with proactive sharing of 2-out-of-n proactive secret sharing, and also state a conjecture about the case of k-out-of-n proactive secret sharing, for $k > 2$. (The exact analysis of the latter case seems rather technical, and we leave it as an open question.)

B.1 Leakage in 2-out-of-n Proactive Secret Sharing

We consider the case of 2-out-of-n sharing. The dealer picks two integer coefficients $A, B \in [0, R-1]$, where R is a parameter which defines the size of the range of possible coefficients. Party i receives the share $Q(i) = Bi + A$, and the arithmetic operations are done over the integers. The secret is the value of A.

Obviously, the share $Q(i)$ might leak some information. For example, if $Q(1) = 10$ then the secret can only be in the range $[0, 10]$. We will show that, except with negligible probability, very little is leaked about the secret.

Theorem 1. *Let Q be a linear polynomial with coefficients that are randomly chosen integers from the range $[0, R)$. Party i receives the value $Q(i)$, where the polynomial is computed over the integers. Let δ be a parameter (e.g., $\delta = 40$). It holds, except with probability $2^{-\delta}$, that for all $a \in [0, R)$ the a-posteriori probability of $Q(0) = a$ given the share known to any specific party, is at most $\frac{2^{\delta/2}\sqrt{i}}{R} \leq \frac{2^{\delta/2}\sqrt{n}}{R}$ (where n is the number of parties).*

The theorem is proved below. Keeping in mind that $R \approx 2^{4096}$, this probability is very close to the a-priori event of choosing $Q(0)$ with probability $\frac{1}{R}$.

Implication for typical parameter setups for RSA: A reasonable setting is where the number of servers is, say, $n = 16$, and we wish to ensure that the theorem holds with probability of at least $1 - 2^{-\delta} = 1 - 2^{-40}$. (This corresponds to setting a *statistical* security parameter to 40, as is common in the cryptographic literature.) In this case no secret $a \in [0, R)$ is chosen with probability greater than $\frac{2^{20} \cdot 4}{R} = \frac{2^{22}}{R}$. This means that an adversary learns at most 22 bits about the value of the secret. In the context of RSA, this means that the adversary is given at most 22 bits of knowledge about the secret key (which we set to be at least 4096 bits long).

Implication to learning the key refresh information in ESKM: The ESKM system uses proactive sharing of a polynomial z for which $z(0) = 0$. We are therefore not worried about hiding $z(0)$, but rather about the information that is leaked by the refresh information $z(i)$ received by one CC node the refresh information $z(j)$ of other nodes. Namely, the attack scenario is that an adversary which controls a corrupt CC node i might use the refresh value that it receives to learn about the refresh value of CC node j. In the future the adversary might move to control node j and learn its new share. It can then use its knowledge of the refresh information of node j to recover the previous share of that node. Now, since it also knows the share of node i from the earlier epoch, it is able to recover the secret.

We note that our analysis in Theorem 1 is directly applicable to limiting the information that is learned from the refresh received by node i, about the refresh received by node j: For every integer j it is possible to define $Q(x) = z(x + j)$, in which case $Q(0) = z(j)$. Define $\Delta = i - j$. The analysis of Theorem 1, of the leakage about $Q(0)$ from knowledge of a share $Q(\Delta)$, applies also to the information that party $j + \Delta$ $(= i)$ learns from the refresh share $z(i)$ about the share $z(j)$ of party j.

Implication to hiding the RSA secret key: Let $s_t()$ be the polynomial that hides the RSA secret key in time t, namely $s_t(0) = d$. Let $z_t()$ be the polynomial that is used for the proactive share refresh in time t. Therefore for all i it holds that $s_{t+1}(i) = s_t(i) + z_t(i)$.

We consider the case where the polynomials $s_t()$ are linear. In this case an adversary that learns any two values of any polynomial $s_t()$ can recover the secret key. We are interested in the power of an adversary that can control at most a single server at each time. Denote the server that is compromised by the adversary at time t as $c(t)$. The adversary therefore knows the set of values $\langle s_1(c(1)), z_1(c(1))\rangle, \langle s_2(c(2)), z_2(c(2))\rangle, \ldots = \{\langle s_t(c(t)), z_t(c(t))\rangle\}_{t=1,2,\ldots}$. We would like to claim that the adversary cannot learn $s_t(0)$ is any time t.

Consider first the simpler case where the adversary controls the same server i in all times. Therefore the adversary only knows $\{\langle s_t(i), z_t(i)\rangle\}_{t=1,2,\ldots}$, and the secret key $s_t(0)$ is independent of the information that the adversary has.

Suppose now that the adversary controls (over time) more than one server, and focus on a time t such that the adversary controls two different servers in times t and $t+1$. The adversary therefore knows $\langle s_t(i), z_t(i)\rangle, \langle s_{t+1}(j), z_{t+1}(j)\rangle$ for $i \neq j$. If the adversary knows $s_t(0)$ it can interpolate $s_t()$, compute $s_t(j)$, and then compute $z_t(j) = s_{t+1}(j) - s_t(j)$. But this contradicts Theorem 1. (More accurately, the contradiction to the theorem occurs even if the adversary learns more than $\frac{2^{\delta/2}\sqrt{n}}{R}$ bits of information about the secret $s_t(0)$, except with probability $2^{-\delta}$.) The same argument holds with respect to the adversary knowing $s_{t+1}(0)$.

Linear polynomials and $i = 1$

To start arguing about the proof of the theorem it is instructive to examine the simplest case, of a linear polynomial and User 1. This user receives the share $Q(1) = A + B$. It is required to analyze what can be learned about A given the value $Q(1)$. Namely, analyze the conditional probability $\Pr(A = a|A+B = Q(1))$.

$$\Pr(A = a \mid A + B = Q(1)) = \frac{\Pr(A + B = Q(1) \mid A = a)\Pr(A = a)}{\Pr(A + B = Q(1))}$$

It is easy to observe that

1. $\Pr(A + B = Q(1) \mid A = a)$ equals $\frac{1}{R}$ if $Q(1) - a \in [0, R-1]$, and is equal to 0 otherwise.
2. $\Pr(A = a) = \frac{1}{R}$ always.
3. Regarding $\Pr(A + B = Q(1))$,
   (a) If $0 \leq Q(1) < R$ then a can be any value in the range $[0, Q(1)]$, and therefore $\Pr(A + B = Q(1)) = (Q(1) + 1)/R^2$.
   (b) If $R \leq Q(1) \leq 2R - 2$ then a can be any value in the range $[Q(1) - R + 1, R)$, and therefore $\Pr(A + B = Q(1)) = (R - (Q(1) - R + 1))/R^2$.
   Combining these two cases, we get that $\Pr(A + B = Q(1)) = (R - |Q(1) - R + 1|)/R^2$ for all values of a.

Therefore, for each a such that $Q(1) - a \in [0, R-1]$, it holds that

$$\Pr(A = a \mid A + B = Q(1)) = \frac{1/R \cdot 1/R}{\frac{R-|Q(1)-R+1|}{R^2}} = \frac{1}{R - |Q(1) - R + 1|}$$

In other words, given $Q(1)$ there are exactly $R - |Q(1) - R + 1|$ values of A for which $Q(1) - a \in [0, R - 1]$, and they all have the same probability of being chosen by the dealer. (Note that the range of possible values for A becomes greater as the value of $Q(1)$ becomes closer to R.)

We would like to show that $|Q(1) - R + 1|$ is typically small. Based on Item 3 above, it holds that

$$\begin{aligned}
\Pr(|Q(1) - R| < R - \mathit{\Delta}) &= \Pr(\mathit{\Delta} < Q(1) < 2R - \mathit{\Delta}) \\
&= \sum_{s=\mathit{\Delta}}^{2R-\mathit{\Delta}} \frac{R - |s - R + 1|}{R} \cdot \frac{1}{R} = 2 \sum_{s=\mathit{\Delta}}^{R} \frac{s+1}{R} \cdot \frac{1}{R} \\
&= \frac{2}{R^2} \frac{(R + \mathit{\Delta})(R - \mathit{\Delta})}{2} = \frac{R^2 - \mathit{\Delta}^2}{R^2} \\
&= 1 - \frac{\mathit{\Delta}^2}{R^2}
\end{aligned}$$

Therefore this event happens with probability $1 - \frac{\mathit{\Delta}^2}{R^2}$. In this case, $|Q(1) - R| < R - \mathit{\Delta}$ and therefore

$$\Pr(A = a \mid A + B = Q(1)) = \frac{1}{R - |Q(1) - R|} < \frac{1}{\mathit{\Delta}}$$

For example, when working with a 4096 bit RSA moduli, and setting $R = 2^{4096}$, then for $\mathit{\Delta} = 2^{4000}$ we get that, given $Q(1)$, it holds, except with probability $\frac{\mathit{\Delta}^2}{R^2} = 2^{-192}$, that there are at least 2^{4000} possible values of the secret A, and each of these values is chosen with probability of at most $1/\mathit{\Delta} = 2^{-4000}$.

Note that the ESKM application uses proactive sharing of a polynomial z for which $z(0) = 0$. We are therefore not worried about hiding $z(0)$, but rather about the information that is leaked by one share about other shares. Note that for any integer e it is possible to define $Q(x) = z(x + e)$, in which case $Q(0) = z(e)$. Therefore, the analysis given here, for the leakage about $Q(0)$ from knowledge of $Q(1)$, applies to the information that User $e + 1$ learns about the share of user e.

The general case of linear polynomials

Party i receives the share $Q(i) = A + iB$, and we are interested in the value of $\Pr(A = a | A + Bi = Q(i))$.

Note that $Q(i)$ is the range $[0, (i + 1)(R - 1)]$ which has almost $(i + 1)R$ elements, but it can only be equal to values which are equal to A modulo i. Obviously, $Q(i)$ reveals the value of A modulo i. In addition, it holds that

$$\Pr(A = a \mid A + iB = Q(i)) = \frac{\Pr(A + iB = Q(i) \mid A = a) \Pr(A = a)}{\Pr(A + iB = Q(i))}$$

The easy parts of this expression are $\Pr(A = a) = \frac{1}{R}$, and $\Pr(A + iB = Q(i) \mid A = a)$ which is equal to $\frac{1}{R}$ when a is equal to $Q(i)$ modulo i and $(Q(i) - a)/i \in [0, R - 1]$, and is equal to 0 otherwise.

A careful analysis shows that when $R \le Q(i) \le iR$ it holds that $\Pr(A+iB = Q(i)) = \frac{R/i}{R}\frac{1}{R} = \frac{1}{iR}$. This holds since a can be any value in $[0, R)$ which is equal to $Q(i)$ modulo i, and given the choice of a there is only one option for a b value resulting in the right value of $Q(i)$.

For values of $Q(i)$ in the external areas of the possible range, namely in $[0, R)$ or $[iR, (i+1)R - i)$, $\Pr(A + iB = Q(i))$ decreases linearly with the distance of $Q(i)$ from the boundary of the range. For example, when $Q(i) < R$ then $\Pr(A + iB = Q(i)) = \frac{Q(i)/i}{R}\frac{1}{R} = \frac{Q(i)}{iR^2}$. (When $i = 1$ these probabilities match our previous special case analysis for $Q(1)$.)

Therefore, when $R \le Q(i) \le iR$ it holds for all a such that $a = Q(i) \mod i$, that $\Pr(A = a|A+iB = Q(i)) = i/R$. In other words, the value of A is uniformly distributed among all values that are equal to $Q(i)$ modulo i. Beyond this range, when we know that $Q(i)$ is in the range $[\Delta, (i+1)R - \Delta]$, it holds for all a that $\Pr(A = a|A + iB = Q(i)) < i/\Delta$.

As before, the value of $Q(i)$ is very unlikely to be close to the boundaries of its possible range, i.e. to 0 or to $(i + 1)R$. Therefore, it is likely that Δ is large. For example,

$$\Pr(Q(i) \le \Delta) = \sum_{s=0}^{\Delta} \frac{s}{iR^2} = \frac{\Delta(\Delta+1)}{2iR^2} \approx \frac{\Delta^2}{2iR^2}$$

The overall behavior is that the range of possible values of $Q(i)$ is of size $(i + 1)R$ (compared to $2R$ in the case of $i = 1$); getting to a distance smaller than Δ from the boundaries of this range happens with probability smaller than Δ^2/iR^2 (compared to Δ^2/R^2 in the case of $i = 1$); if this event does not happen, and $\Delta \le Q(i) \le (i+1)R - \Delta$, then $\Pr(A = a|A + iB = Q(i)) < i/\Delta$.

The theorem follows by setting $\delta = -\log(\Delta^2/iR^2)$ (in this case $Q(i)$ is in the required range with probability $1 - 2^{-\delta}$, and $\Pr(A = a|A+iB = Q(i)) < i/\Delta = \frac{2^{\delta/2}\sqrt{i}}{R}$).

B.2 Secret Sharing with a Threshold Greater than 2

We do not know how to prove a bound on the leakage for the case of polynomial secret sharing with a degree greater than 2. We conjecture that this is a technical issue of bounding the relevant probabilities, and state a conjecture about the probability bound.

Conjetcture 1 *Let Q be a polynomial of degree k with integer coefficients that are randomly chosen in the range $[0, R)$. Party i receives the value $Q(i)$, where the polynomial is computed over the integers. Then it holds, except with small probability, that for all $a \in [0, R)$ the a-posteriori probability of $Q(0) = a$ given any k shares is at most $\frac{2^{c\cdot k^2}}{R}$ for some constant c.*

(Note that $R = 2^{4096}$ and k is small, e.g., $k = 8$. We also expect the constant c to be very small. The resulting bound is thus very small.)

We describe here a *sketch* of the proof for the general case of k-out-of-n secret sharing. To simplify the notation, consider a polynomial $Q(x) = \sum_{j=0}^{k-1} A_j x^j$, with coefficients chosen randomly in $[0, R)$, and parties with identities $1, \ldots, k-1$. In this case we are interested in

$$\Pr(A_0 = a \mid Q(1), \ldots, Q(k-1)) = \frac{\Pr(Q(1), \ldots, Q(k-1) \mid A_0 = a) \Pr(A_0 = a)}{\Pr(Q(1), \ldots, Q(k-1))}$$

The probability that $Q(1), \ldots, Q(k-1)$ attain specific values given that $A_0 = a$, namely $\Pr(Q(1), \ldots, Q(k-1) \mid A_0 = a)$, is equal to $1/R^{k-1}$ for some choices of the values of $Q(1), \ldots, Q(k-1)$, and is equal to 0 otherwise. (This depends on whether the resulting set of linear equations has a solution in the range or not.) Also, $\Pr(A_0 = a) = 1/R$ always.

As for $\Pr(Q(1), \ldots, Q(k-1))$, each value $Q(i)$ is distributed almost uniformly over values with are far from the boundaries of the range of possible values for $Q(i)$, and the probability of each possible value in that range is about $\frac{1}{i^{k-1}R}$. The likely case is that the $Q(i)$ values are far from the boundaries of their respective ranges. In this case it holds that $\Pr(Q(1), \ldots, Q(k-1)) < \frac{1}{((k-1)!)^{k-1} R^{k-1}}$ (we ignore here the fact that some values in the range $[1, k-1]$ are not co-prime to each other, and use this more conservative bound).

Plugging in these values, we get that $\Pr(A = a \mid Q(1), \ldots, Q(k-1)) < ((k-1)!)^{k-1}/R$. Since k is at most the number of servers n, n is typically small and R is large, then the resulting bound is very small. For example, for $n = 8$ and $R = 2^{4096}$ the bound is $((n-1)!)^{n-1}/R \approx 2^{-4010}$.

Algorithm 2 Malicious Model Share Recovery Algorithm for Existing Node i, Recovering Node r

```
    Input parameters:
    $s_i$ - current share of node $i$
    $p$ - helper polynomial coefficient limit
    $n$ - number of nodes
    $f$ - maximal number of faulty nodes ($f = k - 1$)
    $v$ - used as the base for verification of exponents
    $v^{s_1}, \ldots, v^{s_n}$ - verification values
    $v_p$ - verification field

    Round 1:
 1: Choose $\alpha_0^i, \ldots, \alpha_{k-1}^i \sim U([0,p))$ to create a random polynomial $p^i(x) = \sum_{q=0}^{k-1} \alpha_q^i \cdot x^q$ over the integers.
 2: Let $c^i = p^i(r)$
 3: Define the polynomial $z^i(x) = p^i(x) - c^i$
 4: Compute shares $z_1^i = z^i(1), \ldots, z_n^i = z^i(n)$
 5: Compute $v^{\alpha_0^i}, \ldots, v^{\alpha_{k-1}^i}, v^{c^i}$ over $v_p$
 6: Compute $Sig_i \leftarrow \mathcal{H}(v^{\alpha_0^i}, \ldots, v^{\alpha_{k-1}^i}, v^{c^i})$
 7: for each existing node $\ell \neq i$ do
 8:     Send $z_\ell^i, (v^{\alpha_0^i}, \ldots, v^{\alpha_{k-1}^i}, v^{c^i}), Sig_i$ to node $\ell$
 9: end for

    Round 2:
10: for each received share $z_i^\ell$ from node $\ell$ do
11:     Verify that $\mathcal{H}(v^{\alpha_0^\ell}, \ldots, v^{\alpha_{k-1}^\ell}, v^{c^\ell}) = Sig_\ell$
12:     Verify that $v^{z_i^\ell} = \prod_{q=0}^{k-1} \left(v^{\alpha_q^\ell}\right)^{i^q} \cdot \left(v^{c^\ell}\right)^{-1} \mod v_p$
13:     if verification failed then Report node $\ell$
14: end for
15: if verified at least $f + 1$ shares then
16:     Let $s_i^* = s_i + \left(\sum_{\text{verified shares } \ell} z_i^\ell\right)$. (Summation is over the integers)
17:     Send OK messages to everyone with $Sig_\ell$ of each verified sender $\ell$, report missing or invalid shares.
18: else Abort
19: end if

    Round 3:
20: Compare signatures in all received OKs
21: Publicly announce everything known by node $i$ on disputed and missing shares to everyone (there are up to $f$ such shares)

    Round 4:
22: Complete missing information using information sent in Round 3: Update $s_i^*$, Ignore OKs and shares of identified malicious nodes.
23: if received at least $f + 1$ valid OKs then
24:     Let $valid_i$ be a bitmap where $valid_i(j) = 1$ if $z_i^j$ is valid and 0 otherwise.
25:     Send the following values to node $r$: $s^*(i), valid_i, (v^{s_1}, \ldots, v^{s_n}), \left(v^{\alpha_1^i}, \ldots, v^{\alpha_{k-1}^i}, v^{c^i}\right), (Sig_1, \ldots, Sig_n)$
26: else Abort
27: end if
```

Algorithm 3 Malicious Model Share Recovery Algorithm for Recovered Node r

```
$\lambda^{p}_{i,j}$ - Lagrange interpolation coefficient for $p(i)$ when interpolating $p(j)$

1: Send request to nodes $1, \dots, n$
2: **for each** received response from node $\ell$ **do**
3:     Verify that $(v^{s_1}, \dots, v^{s_n}), (Sig_1, \dots, Sig_n)$ are the same as previously received responses
4:     Verify that $Sig_\ell = H\left(v^{\alpha^i_0}, \dots, v^{\alpha^i_{k-1}}, v^{c^i}\right)$
5:     Verify that $valid_\ell$ is equal to previous $valid$ received. If not, restart the process with the result of applying AND on all $valid$ bitmaps from verified senders.
6: **end for**
7: **if** received at least $f+1$ verified responses **then**
8:     Verify that $v^{s^*_\ell} = v^{s_\ell} \cdot \prod_{j \in valid_\ell} \prod_{q=0}^{k-1} \left(v^{\alpha^j_q}\right)^{\ell^q} \cdot \left(v^{c^j}\right)^{-1} \mod v_p$
9:     Let $s^*(r) = \sum_{\text{verified responses } \ell} s^*(\ell) \cdot \lambda^{s^*}_{\ell,r}$
10:    Send a notification to all nodes
11: **end if**
```