Strength in Numbers: Threshold ECDSA to Protect Keys in the Cloud

Marc Green and Thomas Eisenbarth

Worcester Polytechnic Institute, Worcester, MA, USA
{marcgreen,teisenbarth}@wpi.edu

Abstract. Side-channel attacks utilize information leakage in the implementation of an otherwise secure cryptographic algorithm to extract secret information. For example, adversaries can extract the secret key used in a cryptographic algorithm by observing cache-timing data. Threshold cryptography enables the division of private keys into shares, distributed among several nodes; the knowledge of a subset of shares does not leak information about the private key, thereby defending against memory disclosure and side-channel attacks. This work shows that applying threshold cryptography to ECDSA—the elliptic curve variant of DSA—yields a fully distributive signature protocol that does not feature a single point of failure. Our security analysis shows that Threshold ECDSA protects against a wide range of side-channel attacks, including cache attacks, and counteracts memory disclosure attacks. We further provide the first performance analysis of Threshold ECDSA, and provide a proof of concept of the protocol in practice.

Keywords: Threshold Cryptography, Elliptic Curve Cryptography, ECDSA, SSL/TLS, Side-channel Attacks, Cloud Computing

1 Introduction

Side-channel attacks such as cache attacks are receiving increased attention since the advent of cloud computing. Cloud-based services such as Amazon's EC2 and Microsoft's Azure run multiple virtual machines (*VM*s), one (or more) per customer, on each hardware unit for efficiency. Since VMs are known to provide *logical isolation* between themselves, each other, and their host operating system, it was (mistakenly) assumed that they were secure in terms of keeping sensitive data confidential. However, resource sharing—the base concept behind cloud computing—prevents complete isolation between VMs, thus creating side-channels that can be exploited by adversaries. These attacks exploit shared microarchitectural resources such as caches to infer sensitive data across VM boundaries. With an increasing number of studies demonstrating that the VM co-location problem is surmountable [33, 34, 36], several successful cross-VM side-channel attacks were proposed with applicability in the cloud: The Flush+Reload cache attack, for example, was shown to recover RSA keys [38] and AES keys [17]

across cores and across VM boundaries; Benger et al. recovered ECDSA private keys using Flush+Reload [2]. Recently, Liu et al. and Irazoqui et al. have both developed new cross-VM attacks using the Prime+Probe method that do not rely on memory deduplication [25, 16], making cache attacks a realistic concern for today's public clouds.

Existing defenses against cache attacks fall under two categories: architectural countermeasures and algorithmic countermeasures. The former prevent the exploitability of side-channels on the architectural level. They can be implemented at the hardware [35, 23, 10], the hypervisor, or the OS level [21, 39]. Architectural countermeasures generally come at a performance overhead. Since the overhead usually applies to all applications, not only security critical ones, they are often viewed as too expensive. Algorithmic countermeasures, on the other hand, usually armor the implementation or algorithm of the used cryptosystems. Protection methods include: (*i*) The use of isochronal algorithms, thus preventing data dependent timing differences from arising [27]; (*ii*) The use of software diversification to randomize control flow and thus *hide* leaked information [8]; (*iii*) The use of threshold cryptography to prevent sensitive data from appearing in memory at all (e.g., [30]). Unlike architectural countermeasures, only the security-critical code is protected in algorithmic countermeasures. Hence, performance penalties are restricted to security-critical code and do not affect other applications.

Our Contribution: In this paper, we propose Nephele, a cloud-based system resilient to cross-VM side-channel attacks on ECDSA through the use of threshold cryptography. Our approach is completely transparent to the end user and can work as a drop-in replacement for existing ECDSA implementations. Our fundamental technique comes from the application of threshold cryptography: the ECDSA private key is split into multiple shares and distributed to multiple VMs. To sign a message, each VM will use its private key share to compute a *partial signature*; the partial signatures are then combined to produce the full signature. A certain *threshold* of shares is needed to compute signatures or recover the full private key. This means we've raised the bar for adversaries so that they must now successfully compromise *several* VMs to steal the private key. Furthermore, for added security, our design calls for frequent *re-sharing* of the private key. Key re-sharing invalidates all previous private key shares and replaces them with new ones, thus nullifying any partial work the adversary has accomplished. We also provide a fully distributed key generation algorithm for ECDSA so the private key can be generated without the need for a trusted third party, effectively removing the last remaining single point of failure. The distributed key generation, together with the distributed signature generation and distributed key re-sharing algorithms, provide a fully functional ECDSA interface without the need for a private key to *ever appear in memory*. Hence, our approach helps mitigate a wide range of attacks, since the private key isn't available to simply be stolen by an adversary who has root-compromised a subset of VMs. Our approach can also scale with changing security requirements. The number of shares needed to recover the full key can be adjusted to meet security,

performance, and budget constraints. Our benchmarks show that Nephele incurs a reasonable performance overhead over unprotected ECDSA. In summary, Nephele:

- presents the first *fully distributed* threshold implementation of ECDSA. This is achieved by providing fully distributed key generation and signature generation algorithms. Based on the protocol proposed in [15], we show several tweaks that further enhance performance.
- allows for fully distributed key re-sharing, effectively limiting the number of observations and hence the amount of information that a side-channel adversary can aggregate per key share.
- significantly raises the bar for any attacker by requiring the compromise of at least $t + 1$ machines to recover any sensitive information.

Results are backed up by a detailed security and performance analysis of Nephele, as well as a proof-of-concept implementation[1].

The rest of the paper is organized as follows. We give relevant background information in Section 2. Threshold ECDSA and our modifications are described in Section 3, and we analyze their security in Section 4. We deploy a proof-of-concept of Nephele to a realistic cloud-based setting and analyze its performance in Section 5. We discuss Nephele with respect to related and future work in Section 6. Finally, we conclude in Section 7.

2 Background

2.1 Threshold Cryptography

Threshold cryptography, an application of secure multiparty computation [13], splits sensitive data into multiple *shares* and distributes them among a set of n participants. To compute cryptographic operations, a subset of participants use their shares to compute partial results, which are then combined to produce the final result. A threshold $t \leq n$ is defined such that t or less shares *cannot* be used to compute the final result, but $t + 1$ or more shares can be. We refer to these as (t, n)-threshold secret sharings. When applied to a digital signature scheme, the signer's private key is shared so that partial signatures are computed by the participants. The partial signatures are combined to produce the full signature, identical to one that could be computed with the unshared private key. The mathematical techniques used in these operations are designed such that the unshared private key never appears during computation. Thus, while side-channels could still be used to attack a threshold signature scheme, the adversary *cannot* recover the sensitive data unless $t + 1$ shares are recovered. Note that the use of threshold cryptography thus protects against *all* side-channel attacks, not only cache-timing attacks.

Gennaro, et al., apply threshold cryptography to DSS in [12]. We use their protocol as a basis for Nephele. In [30], Pattuk, et al. use existing threshold

[1] All source code will be available online upon acceptance of this paper.

Table 1. Notation and Definitions

m	Message being signed.
n	The number of participants.
d	Private key, shared into $(d_1, \dots, d_n)$
CURVE	The elliptic curve field and equation used.
G	A generator of CURVE with large prime order q.
q	Order of G, cardinality of $CURVE$.
t	"Threshold", the maximum number of shares that reveal no information about the secret. The secret can be recovered with $t + 1$ shares.[2]

cryptographic algorithms to create a cloud-based system resilient to key leakage. To defend against cross-VM side-channel attacks, their system distributes key shares among a set of cooperating VMs. We adopt their system model in developing our own proof-of-concept system for NEPHELE.

2.2 ECDSA

ECDSA, standardized by NIST in 2003 [28], is a variant of DSA that uses elliptic curve cryptography. It is supported in TLS as part of the key exchange and client authentication algorithms [4]. ECDSA is also supported in OpenSSL, first introduced in version 0.9.8. The main benefit of using elliptic curve cryptography is that the keys and generated signatures are significantly smaller in size. For example, for 128 bits of security, ECDSA requires a 256 bit key, whereas DSA and RSA require 3248 bits [9]. This additional security of ECDSA comes from the intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP). Unlike the Integer Factorization Problem and the ordinary Discrete Logarithm Problem (DLP), there is no subexponential-time algorithm known for the ECDLP, substantially increasing its strength-per-key-bit [19]. Another benefit is that the performance of ECDSA signature generation scales well with key sizes. While ECDSA and RSA have comparable signing performance for small keys (less than 409 and 7680 bits, respectively), ECDSA becomes significantly faster with larger keys [18].

3 Threshold ECDSA (TECDSA)

A robust threshold ECDSA has been proposed by Ibrahim et al. in [15]. That protocol is, in principle, the elliptic curve version of the threshold DSS introduced by Gennaro et al. [12]. Note that, for NEPHELE, we have slightly modified key generation and signature generation by dropping robustness, making key generation explicitly distributed, and improving efficiency by reordering some of the steps of the algorithms. We further introduce a key re-sharing algorithm used to refresh the secret key shares as an additional defense mechanism. ECDSA's signature verification algorithm processes no secret information and is not discussed further. The parameters of the protocol are introduced in Table 1 and are used throughout the rest of this paper.

[2] This definition is from [12].

Algorithm 1 TECDSA Key Generation

Domain Parameters: $CURVE$, cardinality q, generator G
Input: None
Output: Public key: Q, private key shares: d_i

1: Generate private key shares $d_i \leftarrow \mathbb{Z}_q^*$ with $JRSS$ ▷ JRSS requires n broadcasts.
2: Broadcast $f_i = G \times d_i$
3: $Q = \text{Exp-Interpolate}(f_1, \ldots, f_n)$ ▷ $[= G \times d]$; See below for Exp-Interpolate()'s definition.
4: **return** (d_i, Q)

Algorithm 2 Re-sharing private key d

Input: Participant P_i's share of private key, d_i
Output: P_i's new private key share, d'_i

1: Generate zero share $z_i \leftarrow \mathbb{Z}_q^*$ with $JZSS$ ▷ JZSS requires n broadcasts.
2: $d'_i = d_i + z_i$
3: **return** d'_i

Key Generation We have modified the ECDSA key generation algorithm to make our signature scheme *fully distributed*. The private key is collectively chosen at random by all participants using JRSS. The threshold key generation algorithm is given in Algorithm 1. Each step of the algorithm is to be executed by every participant and ought to be carried out synchronously due to its interactive nature. Note that, while Ibrahim et al. [15] did not realize they could simply make the key generation fully distributed, their joint random *verified* secret sharing scheme (EC-JRVSS, c.f. Appendix A) can be used to make this key generation *robust*.

Signature Generation While threshold signature generation is well explained in [15], we give a slightly modified version in Algorithm 3 in Appendix B. Our version is closer to the protocol found in [12] (titled *DSS-Thresh-Sig-1*) and shows slightly more efficient performance that the version by Ibrahim et al.

Re-sharing the Private Key The fully distributed key re-sharing is achieved by having the participants execute one round of JZSS, and adding the resulting zero-share to the participant's private key share (see Algorithm 2). As long as one participant is honest during JZSS and introduces randomness, the resulting zero-shares will be random. The use of JZSS allows us to additively mask the private key share without changing the actual private key. Note that key re-sharing is an excellent measure to prevent aggregation of potential key leakage. Like the other distributed algorithms, each step of this algorithm is to be executed by every participant and ought to be carried out synchronously due to its interactive nature.

4 Security Analysis

The proposed protocol, which we will refer to as TECDSA, extends ECDSA, and as such any security results are upper-bounded by the security of ECDSA. The threshold implementation of ECDSA improves security with respect to attacks

that arise when cryptography is used by an implementation in practice. These attacks are usually not considered in the security discussion of the cryptosystem itself. Specifically, our system protects against memory disclosure attacks as long as at most t parties are affected. It also protects against implementation attacks such as side-channel attacks.

4.1 Scenario

We assume protocol participants are to be instantiated as independent VMs and there are enough VMs to support the desired threshold. Specifically, $n \geq 2t + 1$. The participants have pair-wise private channels they can use to communicate with one another. These are used whenever the protocol needs to execute an instance of SSS, JRSS, or JZSS. Also, the private key is created in a shared fashion by invoking the key generation algorithm described in Section 3. The key is to be re-shared according to Section 3. Under these assumptions, we define our **security goals** as follows:

1. The system should withstand side-channel attacks of a passive, potentially co-located adversary.
2. The system should be able to tolerate memory disclosure attacks on up to t participants. These memory disclosure attacks can be viewed as an attacker managing to break into a VM and read the entire memory content, including all keys. Similarly, it describes a system that has been successfully compromised by means of a side-channel attack.
3. The system should remain operational when up to t participants have been compromised by an adversary. In addition, the remaining honest majority should be able to identify misbehaving nodes that try to compromise the protocol operation.

Based on the scenario and the security goals, we now define the adversary. The adversary's primary goal is to learn the secret key d. A secondary goal is to disrupt service, e.g., by dropping network messages or sending bogus partial signatures. We define three types of adversaries:

An eavesdropping side-channel adversary is able to observe all traffic sent over the network. In addition, the adversary can observe a subset of physical machines participating in TECDSA. This will allow the adversary to collect partial information on secrets processed by the monitored participants through side-channels. This passive adversary is only trying to steal keys and will not actively corrupt messages.

A persistent adversary is one who compromises one of the participating VMs in order to witness at least one execution of the protocol. This includes eavesdropping on secure channels and gaining root privileges on the VM. Like the eavesdropping side-channel adversary, the persistent adversary will not attempt to corrupt signatures throughout execution of the protocol.

An active adversary is a persistent adversary who additionally has the means and intent to *manipulate* or halt communication sent by compromised VMs. Compromised VMs will be able to exchange information and collude.

4.2 Security Analysis

The security of TECDSA against the above adversaries comes from the use of threshold cryptography; the adversary can no longer directly extract the secret key d from any one participant, since no participant has knowledge of d on its own. Instead, at least $t + 1$ shares of d must be extracted, each of which is stored with a different participant (in the cloud, each participant would be a different VM, and each VM would run on a different machine). Ideally, t is chosen such that it is infeasible to execute $t + 1$ successful cross-VM attacks before the secret key is re-shared. Recall that the aforementioned VM co-location problem is hard and thus it is unlikely an adversary can mount these attacks concurrently. In [12], a threshold signature scheme is secure if it is *unforgeable* and *robust*. While TECDSA is unforgeable (discussed below), it does not currently satisfy the robustness requirement. However, Ibrahim et al. [15] showed that adding robustness to TECDSA is possible and straightforward. We chose to exclude robustness, as it incurs a large network performance penalty. Thus, TECDSA protects against the eavesdropping side-channel adversary and the persistent adversary only.

An *eavesdropping side-channel adversary* does not win upon extraction of a single participant's secret key share d_i; the adversary must extract at least $t + 1$ secret key shares from the participants. A proof of this is given in [12] for DSS. The proof applies to ECDSA as long as the ECDLP remains as intractable as the original DLP. Since the secret key d is periodically re-shared, the adversary's window to extract $t + 1$ shares is limited; Note that any extracted key shares become invalid and useless upon re-sharing the secret key. Both t and the frequency of re-sharing can be adjusted to achieve an appropriate security level.

A *persistent adversary* who eavesdrops on secure channels will learn the secret shares. During signature generation, the adversary can observe k_i, a_i, b_i, and c_i of the compromised participant and thus reconstruct its secret key share d_i (see step 10 of Algorithm 3). While the adversary will still need to acquire an additional t shares of the private key d to win, re-sharing d will not thwart it due to its persistence.

An *active adversary* who influences any of the three algorithms of TECDSA can send bogus data to the other participants during the protocol to force generation of an invalid signature. While TECDSA cannot currently withstand bogus data, the use of Verifiable Secret Sharing (VSS) [11] would enable the detection of bogus shares (and their authors), and would allow the protocol to continue normally, even under attack. (That is, replacing SSS with VSS will allow TECDSA to satisfy the *robustness* requirement.) An active adversary can also prevent the compromised VM from even participating in the protocol. Assuming $n \geq 3t + 1$, TECDSA allows for up to t participants to not participate (see [12] for a proof). Note that, even without VSS, each participant can detect potential manipulation by verifying the generated signature.

None of these adversaries are able to extract meaningful information from other, non-compromised participants. The use of JRSS and JZSS during ephemeral key generation provides randomness as long as there is at least *one* honest

participant supplying random input. (Generating a unique, random ephemeral key k for every signature is crucial for protecting the private key d.) Further, the use of the random masks throughout the protocol prevents information from being leaked during computation.

Regardless of the type of adversary, generating a signature in TECDSA requires knowledge of at least $t+1$ secret key shares. By definition, TECDSA is unforgeable in the presence of an eavesdropping side-channel adversary and a persistent adversary. Since an active adversary, despite being able to influence signature generation, cannot learn the secret key shares of non-compromised participants, TECDSA is unforgeable in its presence as well.

Key Re-sharing Most side-channel attacks aggregate key leakage over several observations to finally recover a full long-term key. In fact, this is usually considered one of the strengths of these attacks, as it allows an attacker to exploit even the faintest leakage channels. The more generic cross-core attacks not relying on deduplication [25, 16], i.e. the more realistic ones, need at least a few hundred observations to find and exploit critical key leakage. Even the ECC-centric cache-attacks that assume deduplication [2, 32] or even core-co-residency [6] need several observations to efficiently extract key information. For this reason, we propose to periodically re-share the private key across all participants. This key re-sharing increases the security of the system by bounding the adversary's time to aggregate leakage on at least $t+1$ shares of d. If the adversary fails to extract $t+1$ secret shares within the given time period, the discovered secret shares become invalid, and the adversary must start over. Unlike [30], which relies on a trusted dealer for key re-sharing, our system re-shares the secret key fully distributively, avoiding a single point of failure that the dealer poses in Hermes.

Conclusion By tolerating up to t key share compromises, and by periodically re-sharing these key shares, TECDSA can withstand side-channel attacks of a passive adversary. Further, TECDSA can tolerate up to t memory disclosure attacks because there is not enough information in the collective memory to reconstruct the secret key. Thus, TECDSA meets the first two security goals we defined in Section 4.1.

5 Protocol in Practice

Nephele is a system similar to [30] that uses TECDSA in a cloud setting to combat various attacks including side-channel attacks. Since our system is *fully distributed*, key generation, signing, and key re-sharing happen in a distributed fashion. The system consists of a set of n networked VMs, $P_i, 1 \leq i \leq n$, each ready to participate in the protocol. The value of n determines the maximum threshold allowed: $t \leq \frac{n-1}{2}$ (equivalently, $n \geq 2t+1$). The intuition behind this is as such: Since the protocol essentially involves multiplying two t-degree polynomials (step 5 of Algorithm 3), we need $2t+1$ shares to interpolate the $2t$-degree polynomial formed by their product (step 6 of Algorithm 3).

Key generation is performed once during setup. Key re-sharing can be performed at frequent intervals and, like key generation, only involves interaction

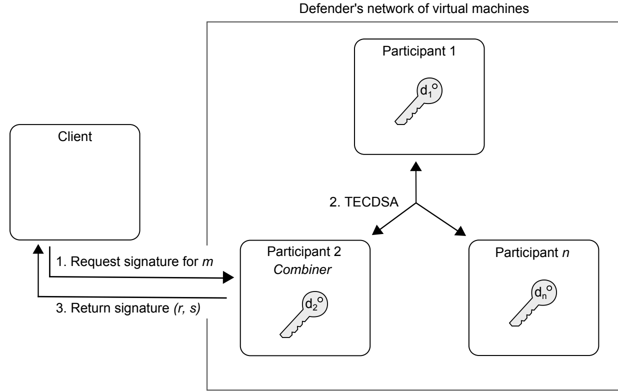

Fig. 1. System Overview. The client connects to Participant 2, making it the combiner. The combiner initiates TECDSA and returns the resulting signature to the client.

between the participants. Signature generation is triggered by an outside node. The input is a message m to be signed, sent from a client to one of the participating computers, and the output is the signature (r, s) for m, sent from the participant back to the client. Without loss of generality, we use the term *combiner*, denoted P_c, to refer to the participant that initiates the protocol, i.e., the one requesting a signature for m on behalf of the client. See Figure 1 for an overview of the system.

5.1 Implementation Details

We modify Algorithm 3 slightly to optimize the protocol under the assumption that network latency is a limiting factor while bandwidth is not. Note that key generation and re-sharing can be done any time, i.e. only the signing itself will influence latency. The applied optimizations are discussed in detail in Section 5.2. The abridged modified signing protocol is as follows:

1. P_c broadcasts m to all participants. *($n-1$ network messages)*
2. (a) The participants generate k_i, a_i, b_i, and c_i through known secret sharing techniques. *($n^2 - n$ network messages, can be computed offline)*
   (b) Each participant computes v_i and w_i.
   (c) All participants send v_i and w_i to P_c. *($n-1$ network messages)*
3. Upon receiving all v_is and w_is, P_c computes r and broadcasts it to the others. *($n-1$ network messages)*
4. (a) Each participant computes s_i.
   (b) All participants send s_i to P_c. *($n-1$ network messages)*
5. P_c calculates s and outputs the signature (r, s).

The proof-of-concept is written in Python 2.7 using Sage 6.5. In order to achieve reasonably good performance in spite of Python/Sage, we optimized the polynomial interpolation and the ECC point multiplication. Our *polynomial interpolation* implementation is optimized to only calculate the free term of the

polynomial, as that is the shared secret and the only term of interest. The relevant parts of the inverse of the Vandermonde matrix is precomputed, so that only the matrix vector multiplication must be performed during computation. Our elliptic curve *point multiplication* implementation is written in C and interfaced through Python. This bypasses a technical limitation in Sage and helps give more accurate performance results. We use *Elliptic Curve P-192*, standardized by NIST is 1999, which provides 96 bits of security [29, 9].

5.2 Performance Overhead

Theoretical Analysis The modified protocol reduces the number of networks messages down to $4(n-1)$, not including messages involved with secret sharing (i.e., generating k_i, a_i, b_i, and c_i), by taking advantage of the fact that only P_c needs the signature. This is in comparison with Algorithm 3, which calls for about $2n^2$ network messages (again, not including the messages involved with secret sharing). However, because messages are sent out in parallel in each of the four broadcast steps, the network latency per signature is roughly equivalent to 2 round-trip-times (RTTs).

The secret sharing steps (specifically *JRSS* and *JZSS*, steps 1-3 in Algorithm 3) use a mesh network of communication, generating $n(n-1) \approx n^2$ network messages (assuming all three steps are carried out simultaneously, and their data is bundled into a single message). These steps, however, are independent of the input to the algorithm. Thus, in practice, these secret shares are generated offline when a client is not waiting for a signature. For example, in a (1,3)-threshold secret sharing of an arbitrary secret, 8 messages are sent through the network during signature generation, in addition to the 6 messages needed for the secret sharing steps.

The computational overhead for the combiner consists of 2 point multiplications, 2 calls to Interpolate(), and 1 call to Exp-Interpolate(). The other participants compute neither the polynomial interpolations nor one of the point multiplications. Note that the point multiplication computed by the combiner and not the others (calculating r) does not depend on the system's input and can thus be computed offline. Integer additions and multiplications are negligible and are not included in the overhead.

Experimental Analysis Several experiments were conducted to explore the performance overhead of Nephele, using our proof of concept implementation. These experiments were run on a Dell PowerEdge R720 server with an Intel Xeon 2670v2 processor (10 cores) and 32GB of RAM, running 64-bit Ubuntu Linux 14.04. Our experiments involved running Nephele on Ubuntu 14.04 virtual machines hypervised by KVM, each given 1 CPU core and 2GB of RAM.

We benchmarked our own implementation of ECDSA, written in Python with Sage, as a control group. We ran the control group twice: once on a single VM with no other CPU activity, and once on three VMs simultaneously to better model the conditions that Nephele will be run. We then benchmarked our proof of concept implementation of Nephele, also written in Python with Sage, with

Table 2. Average ($N = 1000$) ECDSA and NEPHELE itemized signature generation times (in milliseconds). The Total without Secret Sharing is included because secret sharing can be conducted offline.

	Comp.	**Network**	**Sec. Sh.**	**Total**	**Total w/o Sec. Sh.**
ECDSA (1 VM)	1.77	N/A	N/A	1.77	1.77
ECDSA (3 VMs)	2.32	N/A	N/A	2.32	2.32
(1,3)-TECDSA	6.81	1.16	8.83	16.8	7.97

Table 3. Average ($N = 1000$) NEPHELE itemized signature generation times and ($N = 100$) key generation times for varying t and n (in milliseconds). Each column represents a different (t,n)-threshold setup.

Operation	**(1,3)**	**(2,5)**	**(3,7)**	**(4,9)**	**(1,9)**
Signature Generation					
Computation	6.81	7.64	9.51	12.99	12.58
Network	1.16	2.24	3.37	4.94	4.82
Secret Sharing	8.83	12.67	17.41	24.60	20.31
Total	16.80	22.55	30.29	42.53	37.71
Total w/o Secret Sharing	**7.97**	**9.88**	**12.88**	**17.93**	**17.40**
Key Generation	33.03	45.08	55.22	67.96	65.30
Key Re-sharing	4.47	7.11	9.41	12.18	11.50

three VMs ($t = 1, n = 3$). One VM took the role of the combiner, and the other two took the role of normal participants. The combiner did not interact with a client, but it initiated the protocol with random messages as if it had been. The results of these experiments are in Table 2. Our benchmarks measure the average signature generation time, itemizing the time spent in computation and sending data through the network. We also separately itemize the time spent secret sharing, since that can be computed offline in an optimized implementation.

NEPHELE incurs non-trivial computational overhead, taking 6.81 ms compared to 2.32 ms for ECDSA. The dominating computational factors in NEPHELE are the elliptic curve point multiplications and the exp-interpolation operations. Table 4 shows the time it takes to execute each of these operations for various (t,n)-threshold schemes. In every threshold scheme, they together constitute more than 80% of the total computation time. In a production version of NEPHELE, the point multiplication could be sped up by using an elliptic curve optimized for speed (for example, Curve25519 [3]). Further, the exp-interpolation operation could be implemented in a clever manner that reduces redundant computation. Note that the exp-interpolation is initially faster than the tabulated point multiplications, because the scalars used are initially very small. Even without these improvements, assuming secret sharing is conducted offline, (1,3)-NEPHELE is not significantly slower than ECDSA.

Table 3 shows the performance degradation of NEPHELE as t and n increase. Since the exp-interpolation computation significantly degrades with more participants, the higher threshold schemes take longer than the (1,3)-threshold scheme. The time it takes to secret share also significantly increases with the num-

Table 4. Average ($N = 1000$) NEPHELE point multiplication and polynomial interpolation times (in milliseconds). Each column represents a different (t,n)-threshold setup.

	(1,3)	**(2,5)**	**(3,7)**	**(4,9)**	**(1,9)**
$w_i = G \times a_i$	2.84	2.80	2.93	3.46	3.38
$R_x = [\beta \times \mu^{-1}]_x$	2.15	2.22	2.33	2.78	2.66
$\beta = \text{Exp-Interp}(w_1, \ldots, w_n)$	0.70	1.49	2.97	5.20	5.08
Sum	**5.69**	**6.51**	**8.23**	**11.44**	**11.12**
% of Total Computation	83.55%	85.20%	86.54%	88.06%	88.39%

Table 5. Comparison of average signature generation time between HERMES and NEPHELE (in milliseconds).

	(t,n) = (1,3)	**(t,n) = (2,5)**	**(t,n) = (4,9)**
HERMES [30]	12.03	12.04	15.88
NEPHELE	7.97	9.88	17.93

ber of participants, emphasizing the necessity to move this operation offline. The time spent waiting on network communications (ignoring secret sharing) increases with the number of participants as well, but not as significantly. However, in an implementation that has optimized the point multiplications and exp-interpolations, the network communications might become the bottleneck of the protocol. Table 3 also shows the performance of NEPHELE's key generation and key re-sharing algorithms. Expectedly, the performance degrades as the number of participants increases. Note that key generation is a one-time cost and key re-sharing will be executed offline where performance is not as critical.

Comparing the benchmarks of the (1,9)-threshold and (4,9)-threshold schemes, we see that increasing t has a much smaller effect on performance than increasing n. Since the threshold t determines the security of the system, we recommend setting it to the theoretical maximum, $\frac{n-1}{2}$, for a given n.

Table 5 compares our results with the performance results of HERMES, a comparable leakage-resilient system for RSA (from [30]). These results do not take into account secret sharing, as that can be done offline in both systems. We see that NEPHELE is faster in the lower threshold schemes, but is slower in the (4,9) scheme. This is due to the poor scaling of the exp-interpolation implementation in NEPHELE. While this comparison gives insight into the relative performance of NEPHELE, there are significant differences between implementations that must be noted: HERMES is written in C as an extension to OpenSSL and exclusively works with RSA, whereas NEPHELE is written in Python and works with ECDSA. Being written in a lower-level language, we expect HERMES to have a faster run-time execution. However, the results from HERMES include the time it takes to establish an SSL connection, whereas the results from NEPHELE do not. In addition, although ECDSA *can* outperform RSA in terms of signature generation, this only happens with large enough keys (according to [18], this happens around 409 and 7680 bits, respectively). HERMES does

not state the RSA key size used, but it is unlikely they picked a large enough one to see significant performance degradation.

6 Related Work

NEPHELE is an implementation of Threshold ECDSA that protects against a range of practical attacks, including cache and other side-channel attacks, as well as full memory disclosure attacks of up to $t$participant machines. A natural application scenario for NEPHELE is for cryptographic services run in the cloud, where additional participants can be created in a cheap and effective manner.

Algorithmic Countermeasures Protecting cryptographic implementations against side-channel leakage has been well-studied since the mid-nineties. Countermeasures originally proposed to protect fast modular exponentiation such as base and exponent blinding [22] or the Montgomery powering ladder [27] are now also widely used to protect elliptic curve cryptography [26, 7]. Various other methods to make scalar multiplication constant-time and constant execution flow have been proposed, e.g. [3, 20, 5]. Unlike the above countermeasures, our method even protects against memory disclosure and hence full compromise of up to t participants. Furthermore, it forces an adversary to compromise at least $t + 1$ participants to recover a secret key. This means practical attacks will become harder, since, e.g., the co-location problem needs to be overcome $t + 1$ times, not just once.

In addition, the key re-sharing limits the number of aggregatable observations per key. That means that many of the above countermeasures can and should be combined with NEPHELE: while the above countermeasures reduce the amount of observable leakage per execution, key re-sharing reduces the amount of aggregatable observations per key. Hence, in combination, they can thwart much stronger attacks. This is especially useful for attacks that succeed with a single observation (e.g. [37]), where key re-sharing by itself is not effective, and thus the security would entirely rely on the threshold property.

Threshold Cryptography Modern works such as [24] improve over existing work in threshold signatures by allowing fully distributed key generation and non-interactive signature generation. While NEPHELE also provides fully distributed key generation, our servers need to communicate during signature generation. However, unlike NEPHELE, which implements ECDSA signatures and can thus be used in existing implementations of TLS libraries, the scheme by Libert et al. requires bilinear maps, which are computationally expensive and are not compatible with commonly standardized crypto-protocols such as TLS.

Threshold cryptography was first applied to elliptic curves in [15]. Like the approach presented here, the scheme builds on the concepts of [12]. We improve over that work by introducing a key re-sharing mechanism to counteract side-channel analysis and providing a *fully distributed key generation* that does not rely on a trusted dealer. We furthermore present first implementation results and show that the scheme is applicable in practical scenarios, while only providing a limited overhead over a non-threshold ECDSA implementation.

NEPHELE is more comparable to HERMES [30], which also uses threshold cryptography to prevent various cloud-based attacks. Unlike HERMES, our work does not require a server out of the reach of the attacker that holds the secret key at all times. Instead, NEPHELE is able to create, use, and re-share keys *fully distributively*, i.e. in a shared fashion in the cloud, without ever creating a single point of failure during the entire life cycle. A more complex and robust threshold implementation of ECDSA has been proposed to secure the protection of bitcoin wallets [14]. While their protocol is robust and requires $n \geq t+1$ instead of $n \geq 2t+1$, this comes at the cost of a huge performance overhead. Please note that they measure execution time in (single to double digit) seconds while we use (single to double digit) milliseconds.

Future Work NEPHELE currently only exists as a proof-of-concept implementation and does not come as a drop-in replacement of the ECDSA engine of TLS for any of the current cryptographic libraries. In order to make NEPHELE a plugin for a cryptographic library like OpenSSL, the code needs to be entirely ported to C, which should give further performance benefits. Furthermore, TECDSA is not currently *robust*, meaning it cannot work unhindered in the presence of corrupted participants. Providing robustness can be achieved by using verifiable secret sharing instead of Shamir's secret sharing. While the solution is simple, VSS requires significantly more network overhead than SSS, so research must be done on how to implement VSS efficiently.

7 Conclusion

We have developed NEPHELE, the first cross-VM side-channel resistant defense for ECDSA. We have analyzed the security of our protocol from a theoretical and practical perspective; our protocol remains secure in the presence of a limited number of compromised machines, and thus resists cross-VM side-channel attacks. We also found NEPHELE to resist all other side-channel attacks, and, in fact, all memory disclosure attacks. We developed a realistic system model in which our protocol could be applied, and found the induced performance overhead is small.

References

1. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing. pp. 1–10. STOC '88, ACM, New York, NY, USA (1988)
2. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: "Ooh Aah... Just a Little Bit": A Small Amount of Side Channel Can Go a Long Way. In: Batina, L., Robshaw, M. (eds.) Cryptographic Hardware and Embedded Systems — CHES 2014, LNCS, vol. 8731, pp. 75–92. Springer (2014)
3. Bernstein, D.J.: Curve25519: new diffie-hellman speed records. In: In Public Key Cryptography (PKC), Springer-Verlag LNCS 3958. p. 2006 (2006)
4. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492

5. Brumley, B.B.: Faster software for fast endomorphisms. Cryptology ePrint Archive, Report 2015/036 (2015), http://eprint.iacr.org/
6. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Matsui, M. (ed.) Advances in Cryptology — ASIACRYPT 2009, Lecture Notes in Computer Science, vol. 5912, pp. 667–684. Springer Berlin Heidelberg (2009)
7. Coron, J.S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Koc, C., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, LNCS, vol. 1717, pp. 292–302. Springer (1999)
8. Crane, S., Homescu, A., Brunthaler, S., Larsen, P., Franz, M.: Thwarting cache side-channel attacks through dynamic software diversity (2015)
9. of Excellence in Cryptology II, E.N.: ECRYPT II yearly report on algorithms and keysizes (2011-2012) (September 2012)
10. Domnitser, L., Jaleel, A., Loew, J., Abu-Ghazaleh, N., Ponomarev, D.: Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. ACM Trans. Archit. Code Optim. 8(4), 35:1–35:21 (Jan 2012)
11. Feldman, P.: A practical scheme for non-interactive verifiable secret sharing. In: Proceedings of the 28th Annual Symposium on Foundations of Computer Science. pp. 427–438. SFCS '87, IEEE Computer Society, Washington, DC, USA (1987)
12. Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T.: Robust threshold dss signatures. In: Proceedings of the 15th Annual International Conference on Theory and Application of Cryptographic Techniques. pp. 354–371. EUROCRYPT'96, Springer-Verlag, Berlin, Heidelberg (1996)
13. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing. pp. 101–111. PODC '98, ACM, New York, NY, USA (1998)
14. Goldfeder, S., Gennaro, R., Kalodner, H., Bonneau, J., Kroll, J.A., Felten, E.W., Narayanan, A.: Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme (June 2015), available at http://cryptolibrary.org/handle/21/456
15. Ibrahim, M., Ali, I., Ibrahim, I., El-sawi, A.: A robust threshold elliptic curve digital signature providing a new verifiable secret sharing scheme. In: Circuits and Systems, 2003 IEEE 46th Midwest Symposium on. vol. 1, pp. 276–280 (2003)
16. Irazoqui, G., Eisenbarth, T., Sunar, B.: S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES. In: 36th IEEE Symposium on Security and Privacy (2015)
17. Irazoqui, G., Inci, M., Eisenbarth, T., Sunar, B.: Wait a minute! a fast, cross-vm attack on aes. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) Research in Attacks, Intrusions and Defenses, Lecture Notes in Computer Science, vol. 8688, pp. 299–319. Springer International Publishing (2014)
18. Jansma, N., Arrendondo, B.: Performance comparison of elliptic curve and rsa digital signatures. Tech. rep., University of Michigan College of Engineering (2004)
19. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ecdsa). International Journal of Information Security 1(1), 36–63 (2001)
20. Käsper, E.: Fast elliptic curve cryptography in openssl. In: Danezis, G., Dietrich, S., Sako, K. (eds.) Financial Cryptography and Data Security, Lecture Notes in Computer Science, vol. 7126, pp. 27–39. Springer Berlin Heidelberg (2012)
21. Kim, T., Peinado, M., Mainar-Ruiz, G.: Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In: USENIX Security symposium. pp. 189–204 (2012)

22. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96, LNCS, vol. 1109, pp. 104–113. Springer (1996)
23. Kong, J., Aciicmez, O., Seifert, J.P., Zhou, H.: Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 2Nd ACM Workshop on Computer Security Architectures. pp. 25–34. CSAW '08, ACM, New York, NY, USA (2008)
24. Libert, B., Joye, M., Yung, M.: Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing. pp. 303–312. PODC '14, ACM, New York, NY, USA (2014)
25. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: 36th IEEE Symposium on Security and Privacy (2015)
26. López, J., Dahab, R.: Fast multiplication on elliptic curves over gf(2m) without precomputation. In: Koc, C., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, LNCS, vol. 1717, pp. 316–327. Springer (1999)
27. Montgomery, P.L.: Speeding the pollard and elliptic curve methods of factorization. Mathematics of Computation 48, 243–264 (1987)
28. National Institute of Standards and Technology: FIPS PUB 186-4: Digital Signature Standard (DSS) (Jul 2013)
29. NIST: Recommended elliptic curves for federal government use (July 1999)
30. Pattuk, E., Kantarcioglu, M., Lin, Z., Ulusoy, H.: Preventing cryptographic key leakage in cloud virtual machines. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 703–718. USENIX Association, San Diego, CA (Aug 2014)
31. Pedersen, T.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) Advances in Cryptology — CRYPTO '91, LNCS, vol. 576, pp. 129–140. Springer (1992)
32. van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Nyberg, K. (ed.) Topics in Cryptology — CT-RSA 2015, Lecture Notes in Computer Science, vol. 9048, pp. 3–21. Springer International Publishing (2015)
33. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 199–212. ACM (2009)
34. Varadarajan, V., Zhang, Y., Ristenpart, T., Swift, M.: A placement vulnerability study in multi-tenant public clouds. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 913–928. USENIX Association, Washington, D.C. (Aug 2015)
35. Wang, Z., Lee, R.B.: New cache designs for thwarting software cache-based side channel attacks. In: Proceedings of the 34th Annual International Symposium on Computer Architecture. pp. 494–505. ISCA '07, ACM, NY, NY, USA (2007)
36. Xu, Z., Wang, H., Wu, Z.: A measurement study on co-residence threat inside the cloud. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 929–944. USENIX Association, Washington, D.C. (Aug 2015)
37. Yarom, Y., Benger, N.: Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140 (2014), http://eprint.iacr.org/
38. Yarom, Y., Falkner, K.: Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 719–732. USENIX Association, San Diego, CA (Aug 2014)

39. Zhang, Y., Reiter, M.K.: Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. pp. 827–838. CCS '13, ACM, New York, NY, USA (2013)

A Existing Tools for Secret Sharing

The following known techniques are used to share secrets in our protocol.

Shamir's Secret Sharing (SSS) [11]: Given a secret s, choose a random polynomial $f(x)$ with coefficients in $\mathbb{Z}_q$ of degree t, such that $f(0) = s$. Give each participant P_i (where $i \in \mathbb{Z}_q^*$) the value $f(i)$ mod q, as its share. We denote this sharing as $(s_1, \ldots, s_n) \overset{(t,n)}{\longleftrightarrow} s \bmod q$.[3] The secret can be reconstructed using polynomial interpolation over $t+1$ (or more) shares. SSS works because $t+1$ points define a polynomial of degree t. Since SSS chooses a t-degree polynomial and distributes $n > t$ coordinates to the participants, as long as $t+1$ participants collaborate, the polynomial will be able to be recovered. Evaluating the polynomial at $x = 0$ reveals the secret.

Joint Random Secret Sharing (JRSS) [31]: The goal of this technique is for a group of participants to collectively share a secret without knowledge of the secret. Each participant picks a random local secret and shares it with the group using *SSS*. Each participant then adds all of the shares received from the participants, including its own. This sum is the joint random secret share. Note that the randomness introduced by a single honest participant is enough to keep the joint secret unknown (in the event that other participants pick non-random local secrets).

Joint Zero Secret Sharing (JZSS) [1]: Similar to *JRSS*, but each participant locally shares 0 instead of a random value. Shares produced with this technique are used to blind potential weakpoints in the algorithm.

Ibrahim et al. [15] also proposed two verifiable secret sharing protocols for elliptic curve cryptography, Joint Random Verifiable Secret Sharing (EC-JRVSS) and Joint Zero Verifiable Secret Sharing (EC-JZVSS). These protocols extend JRSS and JZSS by having each party broadcast its polynomial coefficients and evaluation points, protected by EC scalar multiplication. Then all parties verify the correctness of each other's polynomials and shares, adding robustness, but also significant performance overhead to the protocol.

B Threshold ECDSA Signature Generation

The threshold ECDSA signature generation protocol is based on the threshold DSS signature generation protocol found in [12] and has first been proposed in [15]. Unsurprisingly, the fundamental difference between our protocol and

[3] This notation is from [12].

Algorithm 3 TECDSA Signature Generation

Domain Parameters: $CURVE$, cardinality q, generator G
Input: Message m to be signed, private key share $d_i \in \mathbb{Z}_q^*$
Output: Signature $(r, s) \in \mathbb{Z}_q^{*2}$ for m

Distributed Key Generation ▷ JRSS/JZSS require n broadcasts.
1: Generate ephemeral key shares $k_i \leftarrow \mathbb{Z}_q^*$ with $JRSS$
2: Generate mask shares $a_i \leftarrow \mathbb{Z}_q$ with $JRSS$
3: Generate masks shares $b_i, c_i \leftarrow {\mathbb{Z}_q}^2$ with $JZSS$
Signature Generation
4: $e = H(m)$
5: Broadcast $v_i = k_i a_i + b_i \bmod q$ and $w_i = G \times a_i$
6: $\mu = \text{Interpolate}(v_1, \ldots, v_n) \bmod q$ ▷ $[= ka \bmod q]$
7: $\beta = \text{Exp-Interpolate}(w_1, \ldots, w_n)$ ▷ $[= G \times a]$
8: $(R_x, R_y) = \beta \times \mu^{-1}$ ▷ $[= G \times k^{-1}]$
9: $r = R_x \bmod q$. If $r = 0$, go to step 1.
10: Broadcast $s_i = k_i(e + d_i r) + c_i \bmod q$
11: $s = \text{Interpolate}(s_1, \ldots, s_n) \bmod q$. If $s = 0$, go to step 1.
12: **return** (r, s)

theirs stems from the difference between ECDSA and DSA. This can be seen in steps 5 and 8 of Algorithm 3, in which instances of the ECDLP are constructed instead of instances of the DLP. Note that this protocol outputs the computed signature to every participant. See Section 5 for a modified version that only outputs the signature to the initiating participant for efficiency. The threshold signature generation algorithm is given in Algorithm 3. We describe the algorithm in prose below. It is assumed that the secret key generation has already been performed and the message was distributed prior to initiating the algorithm. Each step of the algorithm is to be executed by every participant and ought to be carried out synchronously due to its interactive nature.

1. **Generate ephemeral key shares** k_i
   The participants generate the ephemeral key k, uniformly distributed in $\mathbb{Z}_q^*$, with a polynomial of degree t, using $JRSS$, which creates shares $(k_1, \ldots, k_n) \overset{(t,n)}{\longleftrightarrow} k \bmod q$. Shares of k are to be kept **secret** by each participant.
2. **Generate mask shares** a_i
   The participants generate a random value a, uniformly distributed in $\mathbb{Z}_q$, with a polynomial of degree t, using $JRSS$ to create shares $(a_1, \ldots, a_n) \overset{(t,n)}{\longleftrightarrow} a \bmod q$. These are used to multiplicatively mask k_i. The shares of a are to be kept **secret** by the corresponding participant .
3. **Generate mask shares** b_i, c_i
   Execute two instances of $JZSS$ with polynomials of degrees $2t$. Denote the shares created in these protocols as $(b_1, \ldots, b_n) \overset{(2t,n)}{\longleftrightarrow} b \bmod q$ and $(c_1, \ldots, c_n) \overset{(2t,n)}{\longleftrightarrow} c \bmod q$. These are used as additive masks. The polynomial must be of degree $2t$ because the numbers being masked involve the products of two polynomials of degree t, doubling the number of shares required to recover the secret. The shares of b and c are to be kept **secret** by the participants.
4. **Compute** digest of message m: $e = H(m)$

5. **Broadcast** $v_i = k_i a_i + b_i \bmod q$ **and** $w_i = G \times a_i$
   Participant P_i broadcasts $v_i = k_i a_i + b_i \bmod q$ and $w_i = G \times a_i$. If P_i does not participate his values are set to *null*. Notice that $(v_1, \ldots, v_n) \overset{(2t,n)}{\longleftrightarrow} ka \bmod q$.
6. **Compute** $\mu =$ **Interpolate**$(v_1, \ldots, v_n) \bmod q$
   Interpolate() [12]: If $\{v_1, \ldots, v_n\}(n \geq 2t + 1)$ is a set of values, such that at most t are *null* and all the remaining ones lie on some t-degree polynomial $F(\cdot)$, then $\mu = F(0)$. The polynomial can be computed by standard polynomial interpolation.
7. **Compute** $\beta =$ **Exp-Interpolate**$(w_1, \ldots, w_n)$
   Exp-Interpolate() [12]: If $\{w_1, \ldots, w_n\}(n \geq 2t + 1)$ is a set of values, such that at most t are *null* and the remaining ones are of the form $G \times a_i$, where the a_i's lie on some t-degree polynomial $H(\cdot)$, then $\beta = G \times H(0)$. This can be computed by $\beta = \sum_{i \in V} w_i \times \lambda_i = \sum_{i \in V} (G \times H(i)) \times \lambda_i$, where V is a $(t + 1)$-subset of the correct w_i's and λ_i's are the corresponding Lagrange interpolation coefficients.
8. **Compute** $(R_x, R_y) = \beta \times \mu^{-1}$
9. **Assign** $r = R_x \bmod q$
   If $r = 0$, go to step 1.
10. **Broadcast** $s_i = k_i(e + d_i r) + c_i \quad \bmod q$
    If P_i does not participate, its values are set to *null*. Notice that $(s_1, \ldots, s_n) \overset{(2t,n)}{\longleftrightarrow} k(m + dr) \bmod q$.
11. **Compute** $s =$ **Interpolate**$(s_1, \ldots, s_n) \bmod q$
    If $s = 0$, go to step 1. See Step 6 for the definition of *Interpolate()*.
12. **Return** (r, s)