Preventing Cryptographic Key Leakage in Cloud Virtual Machines

Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, and Huseyin Ulusoy,
The University of Texas at Dallas

https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pattuk

This paper is included in the Proceedings of the
23rd USENIX Security Symposium.

August 20–22, 2014 • San Diego, CA

ISBN 978-1-931971-15-7

Preventing Cryptographic Key Leakage in Cloud Virtual Machines

Erman Pattuk, Murat Kantarcioglu, Zhiqiang Lin, Huseyin Ulusoy
The University of Texas at Dallas
800 W. Campbell Rd, Richardson TX, 75080

Abstract

In a typical infrastructure-as-a-service cloud setting, different clients harness the cloud provider's services by executing virtual machines (VM). However, recent studies have shown that the cryptographic keys, the most crucial component in many of our daily used cryptographic protocols (e.g., SSL/TLS), can be extracted using cross-VM side-channel attacks. To defeat such a threat, this paper introduces HERMES, a new system that aims to protect the cryptographic keys in the cloud against any kind of cross-VM side-channel attacks by simply partitioning the cryptographic keys into random shares, and storing each share in a different VM. Moreover, it also periodically re-shares the cryptographic keys, thereby invalidating the potentially extracted partial ones. We have implemented HERMES as a library extension that is transparent to the application software, and performed deep case studies with a web and a mail server on Amazon EC2 cloud. Our experimental results show that the runtime overhead of the proposed system can be as low as 1%.

1 Introduction

Recent advances in cloud computing enable customers to outsource their computing tasks to the cloud service providers (CSPs). Typically, CSPs manage extensive amount of computational resources, and provide services, such as *Infrastructure-as-a-service* (IaaS) [40], *Platform-as-a-service* (PaaS) [31], *Software-as-a-service* (SaaS) [44]. By *outsourcing* core computing to the cloud, customers can mitigate the burden of resource management, and concentrate more on the core business tasks. A recent study on the cloud usage [3] reported that nearly 30% of enterprise IT organizations use public IaaS, such as Microsoft's Azure Service [12], Amazon's Elastic Compute Cloud (EC2) [4], or Google's Compute Engine [9].

Despite its numerous advantages, cloud computing also introduces new challenges and concerns, primarily the security and privacy risks [48]. The concerns simply stem from outsourcing critical data (e.g., health records, social security numbers, or even cryptographic keys) and/or computing capabilities to a distant computing environment, where the resources are shared with other potentially untrusted customers.

In particular, to increase efficiency and reduce costs, a CSP may place multiple *virtual machines* (VMs), belonging to different customers, to the same physical machine. In such an execution platform, VMs should be logically isolated from each other to protect the privacy of each client. The CSPs use *virtual machine monitors* (VMM) to realize logical isolation among VMs running on the same physical machine. However, recent studies show that a clever adversary can perform *cross-VM side-channel attacks* (for brevity, cross-VM attack) to learn private information that resides in another VM, even under carefully enforced logical isolation in public cloud infrastructures. More specifically, Ristenpart et al. [41] showed heuristics to improve an adversary's capabilities to place its VMs alongside the *victim* VMs, and learn crude information (e.g., aggregate cache usage). Most recently, Zhang et al. [51] managed to extract ElGamal decryption keys by cross-VM attacks. These studies have clearly demonstrated that logical isolation and trustworthy cloud provider are not necessarily enough to guarantee the security of sensitive information.

It would be too optimistic to assume that an adversary is only limited to the two aforementioned attacks. Unfortunately, there exists a wide variety of side-channel attacks, each with its own setup and methodology (e.g., [13–15, 19, 26, 28, 34, 43]). Simply, the absence of such attacks on public cloud infrastructures does not necessarily mean that they are inapplicable. In fact, there are side-channel attacks that target virtualized environments, and leverage timings of cryptographic operations or monitoring of common resource usage [39, 47]. Those attacks may be just one step behind being directly applicable to the public cloud setting; which is why proposing prevention mechanisms is extremely vital for the security and privacy of the sensitive data in the VMs including the cryptographic keys.

To this end, we present HERMES, a system that remedies the cryptographic key disclosure vulnerabilities of VMs in the public cloud by using well-established cryptographic tools such as *Secret Sharing* and *Threshold Cryptography*. Specifically, the key technique in our system is to partition a cryptographic key into several pieces, which are computed using threshold cryptosystems, and to store each *share* on a different VM. This makes it harder for an adversary to capture the complete cryptographic key itself, since it now has to extract shares from multiple VMs (note that there is no single key or a centralized key anymore in HERMES). To further improve the resilience, the *same* cryptographic key is *re-shared* periodically, such that a share is *meaningful* in only one time period. Consequently, we introduce two significant challenges against a successful attack: (i) multiple VMs should be attacked, and (ii) each attack should succeed within a certain time period. As a proof-of-concept, we apply HERMES to protect the cryptographic keys of *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS) protocols.

Contributions. In short, this paper makes the following four contributions:

1. We present HERMES, a novel system to prevent the leakage of cryptographic keys in cloud VMs via mathematically proven techniques – *Secret Sharing* and *Threshold Cryptography*.

2. As a proof-of-concept, we build a prototype of HERMES and apply it to protect the SSL/TLS cryptographic keys, which is significantly more resilient to any cross-VM attack.

3. We empirically evaluate HERMES with micro benchmarks, and case studies for a web server and a mail server, and show that with optimal setup, HERMES can operate with overheads as low as 1%.

4. We formalize the problem of finding *good* HERMES configurations, which minimizes the security risk for given monetary and performance constraints.

Organization. The rest of the paper is structured as follows: We start by providing some background information in §2 about the protocols and techniques used in HERMES. It is followed by the threat model in §3, and the full technical details of HERMES in §4. Then, we evaluate HERMES regarding its efficiency in §5, and discuss its security in §6. Finally, we review the related work in §7, and conclude in §8.

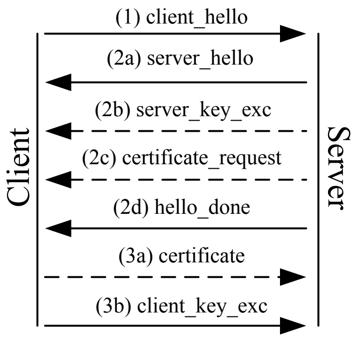


Figure 1: Overview of SSL Protocol Handshake.

2 Background

2.1 SSL/TLS Protocols

SSL and TLS are widely accepted communication protocols to establish a secure channel between two mutually-distrusting parties, where two protocols contain only a few minor differences [16,25,29,37]. For brevity, we will refer the protocols as SSL; and any statement for SSL is also applicable to TLS.

The SSL protocol consists of a *handshake* and a *record* process, where the parties in the protocol are called the *client* and the *server*. In the handshake process, they use *public key cryptography* (PKC) to authenticate each other and agree on the session keys. The session keys are bound for only one session, and used for confidentiality and integrity. To calculate session keys, parties need to share a *master secret*, which is derived from random data exchanged, and *pre-master secret*.

Fig. 1 shows an overview of the handshake process. First, the client starts by sending *client_hello* message (**Step 1**), which contains a set of supported cryptographic algorithms (*cipher suites* in SSL terms), and some random data to be used in key generation. Then, the server sends its certificate, some random data, and the accepted cipher suite (**Step 2a**); and key exchange parameters if necessary (**Step 2b**). Moreover, if the server wants to authenticate the client, it requests the client's certificate (**Step 2c**). The server finishes by sending *hello_done* message (**Step 2d**). If requested, the client sends its certificate to the server, along with some random data signed by its private key (**Step 3a**). Next, it creates a random pre-master secret, encrypts it with the server's public key, and sends to the server (**Step 3b**). Now, both parties can calculate the master secret from the pre-master secret and random data using protocol specific combinations of pseudo-random functions.

Based on the chosen cipher suite, the number and the content of the messages may vary. For instance, when Diffie-Helman (DH) [22] is used for pre-master agreement, the parties sign their DH parameters with their private keys, and send them in **Step 2b** and **Step 3b**. On the other hand, they may use RSA to agree on the pre-master secret, where the client encrypts the pre-master secret with the server's RSA public key, and the server decrypts it using its private key.

2.2 RSA Variants

The following variants of the RSA algorithm alter the way that a message is exponentiated with the private key. In both versions, the dealer holds a public-private RSA key pair, and wants to partition the private key over l non-colluding parties.

Distributed RSA (D-RSA). Given a private key d, D-RSA uses *additive secret sharing*, and partitions d into l random shares $d_1, \ldots, d_l$, where $d \equiv d_1 + \ldots + d_l \bmod \phi(n)$, n is the modulus, and ϕ is Euler's totient function. Given the public key (n, e) and a share, none of the parties can learn anything about d. Furthermore, an adversary should capture all l shares to learn d.

To exponentiate a message $M \in \mathbb{Z}_n$ with d, one of the parties acts as the *combiner*, whose job is to combine partial results from all parties. Each party p_i for $1 \leq i \leq l$ calculates M^{d_i}, and sends it to the combiner. Then, the combiner simply multiplies each message and finalizes the operation. At the end of the process, the combiner does not learn anything about the private key, but only the final result M^d. For a detailed security analysis, we refer to the original paper [24].

Threshold RSA (T-RSA). In this variant, the given private key is partitioned using *shamir secret sharing*, in which only $1 < k \leq l$ shares are needed to complete an exponentiation with d. The key technique in T-RSA is to embed the private key into a degree $(k-1)$ polynomial, evaluate the polynomial on l different points, and share the results over the set of parties. Once again, a party cannot learn the partitioned private key simply from the public key and its share.

To exponentiate a message, k parties are chosen uniformly at random, where the combiner once again does not learn anything other than M^d. On the other hand, an adversary should capture k shares to learn the private key. In App. B, we present more details on private key partitioning and usage, while an intensive security analysis is performed in the original paper [42].

3 Threat Model

Entities. The entities in our threat model are the *Cloud Service Provider* (CSP), the *Defender*, and the *Adversary*, where the last two are simply the clients of the first. The *CSP* offers IaaS and PaaS, which the clients can benefit by initiating VMs. The defender and the adversary use the same CSP, where the latter attacks the former to retrieve private information. Although the CSP has a potential to violate its clients' privacy and integrity, we assume that the CSP is trusted. This is a valid assumption, since (i) *Service Level Agreements* (SLA) provide a clear-cut distinction between what a CSP can and cannot perform on a client's data/VM, and (ii) disobeying a SLA may impose prohibiting punishment for the CSP.

Logical isolation. To improve utilization, the cloud provider may perform multiplexing. Hence, multiple VMs may run on the same physical machine, which means a VM of the adversary may run on the same physical machine with a VM of the defender; and they may share the same physical resources (e.g., CPU, memory, hard-drives, cache). On the other hand, we have no distinction on the VMM that the CSP uses, as long as it provides logical isolation between the VMs on the same physical machine. We assume that the adversary knows the software running on the defender VMs, but cannot leverage the memory vulnerabilities of those software to compromise (i.e., to take full control of) the VMs.

Adversary's goal. The defender has multiple VMs in the cloud, and each one may contain a set of private cryptographic information. This set of information includes temporary symmetric keys (e.g., AES key), or a share of a distributed private key (e.g., share of an RSA key) that is created by HERMES. An adversary's aim is to capture PKC keys, since capturing a session key is useful for only one session, while acquiring PKC keys grants full access. To fulfill its desire, the adversary is allowed to execute any cross-VM attack in its disposal to extract private information from each VM, where the attack itself is applicable to the cloud setup. For instance, in access-driven attacks, the adversary may need to co-reside its VMs with the defender VMs. In such a case, the adversary should achieve co-residency, and make the attack applicable in a typical cloud setup. Moreover, the attacks on the defender VMs are not necessarily executed in serial manner. Each separate adversary VM can employ the cross-VM attack in parallel, if the nature of the attack enables such setup.

Finally, since the adversary uses the same cloud as the defender, we assume that all channels may be eavesdropped by the adversary, starting from right after the bootstrapping of HERMES. Giving this capability to the adversary may seem like an overprovision. However, we take precautions to handle even the worst case scenario, in which the adversary, somehow by-passing CSP's security mechanisms, listens to the conversations between the defender VMs.

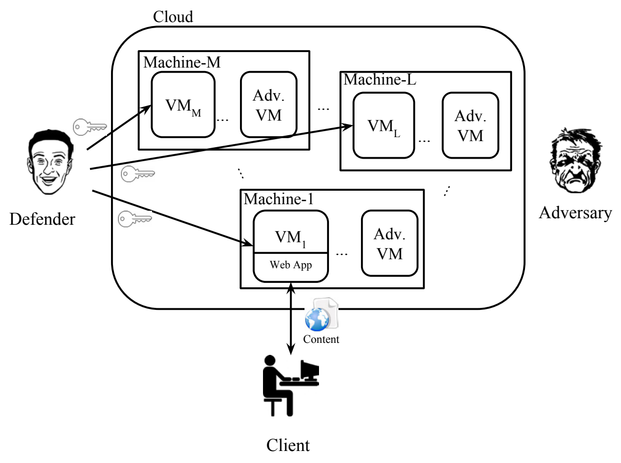


Figure 2: Overview of HERMES Layout.

Misc. We do not consider the placement of the defender VMs, and its effects on the security of HERMES. For instance, one platform (e.g., a region, a physical machine, etc.) may, somehow, be more susceptible to a certain range of cross-VM attacks; or one can claim that the more distributed the defender VMs, the better the security. Zhang et al. aim to physically isolate a defender VM as much as possible [50], thus preventing only access-driven side-channel attacks. Such precautions will tighten the defense against access-driven attacks; however, it will fail to stop the adversary from executing different attacks. On the other hand, HERMES aims to protect the cryptographic keys from all cross-VM attacks, no matter how the VMs are placed.

4 The Proposed System – HERMES

In SSL, a certificate may contain public parameters of different PKCs (e.g., RSA, DSA, ECC), which are employed to encrypt secret information, or to sign and show that certain temporary data is authentic. In HERMES, we assume that the parties use RSA as the PKC; however, extension to other PKCs is trivial as long as a threshold cryptosystem for that new PKC is provided.

Setup. Figure 2 shows an overview of the entities in HERMES: the defender, the adversary, l number of VMs that belong to the defender, and the clients who want to establish secure connection to the defender's VMs using SSL and benefit from the defender's web application. The defender holds a set of private RSA keys, and partitions them over the set of defender's VMs. Each VM holds one share for each partitioned private key, and they act together to exponentiate with it. The VM that directly talks with the client is called the *combiner*, while the remaining VMs are called *auxiliary VMs*. The adversary aims to learn at least one of the partitioned RSA private keys by (i) performing cross-VM attacks on each VM to capture its shares, and by (ii) listening each message flowing between the VMs. To achieve secure communication, each channel is established using our enhanced SSL protocol. More specifically, inter-VM channels are established with mutual verification (i.e., both end of the parties authenticate each other), while only the combiner VM is authenticated in a channel between that VM and a client/defender. The defender re-shares the same private keys every τ seconds. The time window between two consecutive re-sharing moments will be referred to as an *epoch*, while the shares of a private key in any two sessions are independent.

Modes. HERMES has two modes of operation, namely D-RSA and T-RSA modes, using the corresponding RSA variant (cf. §2.2). When the system runs in D-RSA mode, the adversary has to capture all shares of a private key to learn the key itself; whereas in T-RSA mode, it has to capture at least k shares. The benefits of the second mode are two-fold: (i) The system is more fault-tolerant to server failures, and (ii) the system can achieve better utilization by distributing work among different subsets of VMs, especially when $k \leq (l/2)$.

Stages. The execution of HERMES is composed of several stages: (i) Partitioning a private key (§4.2); (ii) Bootstrapping the system by handing in the initial set of shares, and establishing initial inter-VM SSL channels (§4.3); (iii) Establishing connection between a defender VM and a client (§4.4); (iv) Renegotiating an inter-VM SSL channel (§4.5); and (v) Distributing new shares of the same private keys (§4.6).

4.1 Enhancing the SSL Protocol

In SSL, the communicating parties may execute *mutual verification* or *server-only* verification. In any case, the server uses its private key at two possible steps (cf. Fig. 1): (i) After **Step 2a** to sign temporary parameters; (ii) after **Step 3b** to decrypt the pre-master secret. On the other hand, the client uses its private key before **Step 3a** only in the mutual verification. With respect to a regular SSL execution, we change the way that the server or client computes the modular exponentiation of a message with its RSA private key at those steps.

Fig. 3 shows the outline of our modifications in a server-only verified SSL execution. The client performs SSL handshake with the combiner, while the VMs communicate over already established secure channels. After **Step 2a**, the combiner may create temporary key parameters and sign them in collaboration with the auxiliary VMs. It sends a `help_sign` message to all auxiliary VMs in D-RSA mode (or up to k in T-RSA mode), where the

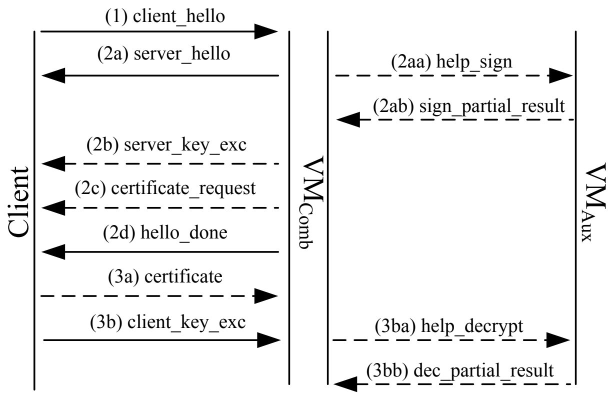


Figure 3: Security Enhanced SSL Outline for Server-only Verification.

message content is simply the parameters to be signed (**Step 2aa**). Each auxiliary VM in the computation calculates its partial result using its share of the private key and gives it to the combiner in the `sign_partial_result` message (**Step 2ab**). On the other hand, if the combiner has to decrypt an incoming message from the client, it sends a `help_decrypt` message to all auxiliary VMs in D-RSA mode (or up to k in T-RSA), containing the *masked* or plain version of the client's message (**Step 3ba**). Then, each auxiliary VM sends the computed partial result with the `dec_partial_result` message to the combiner (**Step 3bb**).

Whether or not the content of the `help_decrypt` message should be masked with a random number depends on the mode of operation. Even in the worst case, where the adversary knows each message exchanged between VMs, the combiner does not have to mask the message in D-RSA. The reason comes from the security of D-RSA. Assume the client sends $M^e \bmod n$ to the combiner, where M is the pre-master secret, and (n,e) is the public key. In order to learn M, the adversary needs each VM's partial result, just like the combiner does. However, even if the adversary cracks down all secure channels and captures all messages, it can only learn $l-1$ parties' partial results, since the combiner does not send its partial result to anyone. Thus, the adversary cannot learn any useful information, and cannot compute M. If T-RSA is employed, then the combiner selects k VMs, $S = \{i_1,\dots,i_k\}$, uniformly at random from the set of VMs and sends the message to them. There are two cases to consider: (i) If the combiner is included in S, then the message does not need masking similar to D-RSA case. The adversary needs k partial results, but can only capture $k-1$. (ii) Otherwise, the combiner masks the message with a random number; since the adversary may have captured k partial results sent from k different auxiliary VMs, and the other parameters in the calculation are public (e.g., $\Delta = l!$, a and b can be calculated from $gcd(e, 4\Delta^2)$), the adversary can now calculate M.

In addition to server-only verified SSL channels, HERMES necessitates mutual verification, since any two defender VMs, VM_i and VM_j, may perform key renegotiation to refresh session keys. Without loss of generality, assume that VM_i is the client in SSL protocol, while VM_j acts as the server. Now, both parties should communicate with the auxiliary VMs to perform operation with their own private keys. The server may need co-operation after steps **2a** and **3b**, while the client may need to sign random data with its private key before **Step 3a**. The server acts as mentioned in server-only authenticated Enhanced SSL. On the other hand, the client sends `help_sign` message to auxiliary VMs before **Step 3a**, and combines the partial results. By following those steps, two defender VMs can execute a successful handshaking process, using already established secure and authenticated SSL channels with the auxiliary VMs.

4.2 Partitioning Keys

Given an RSA key pair (n,e,d) and the number of VMs l, the defender performs partitioning and calculates the shares of each defender's VM. In case HERMES is running in T-RSA mode, the defender uses the third parameter k, minimum number of VMs needed to operate.

In D-RSA, the share of the i^{th} VM, denoted by sh_i, is simply a uniformly randomly chosen value from the domain $\mathbb{Z}_{\phi(n)}$, where $sh_1 + \dots + sh_l$ is equal to d. Hence, the defender chooses $l-1$ random values, and calculates the final share as $sh_l = d - (sh_1 + \dots + sh_{l-1}) \bmod \phi(n)$.

On the contrary, key partitioning process is a bit more complicated in T-RSA. Algo. 1 shows an outline of preparations of each VM's share. For each subset $S_\alpha \subseteq \{1,\dots,l\}$, where $|S_\alpha| = k$, the defender calculates the interpolation constants $\lambda_{0,j}^{S_\alpha}$, and exponents for each $VM_j \in S_\alpha$ (line 10-16). Moreover, the defender stores the modulus values for a,b in V_i's share for d, since i (the function input) states that the given private key belongs to VM_i (line 18-19).

4.3 Bootstrapping the System

The defender creates l VM instances on the CSP, and an RSA key pair (n_i,e_i,d_i) for each VM_i, $1 \le i \le l$. Next, she partitions each private key into shares and gives each VM a unique ID $i \in [1,l]$, the shares that correspond to that ID, and the certificate for the i^{th} RSA key pair.

At this stage, the VMs need to establish initial authenticated and secure SSL channels using our Enhanced SSL. However, as mentioned in §4.1, Enhanced SSL necessitates already established secure SSL channels to transfer messages between VMs. We have to make an assumption here, which will allow us to bypass this requirement, and

Algorithm 1 Preparing shares for T-RSA

```
1: Input: RSA Parameters $n$, $p = 2p' + 1$, $q = 2q' + 1$, $e$, $d$
2: Input: T-RSA parameters $l$, $k$, $i$
3: for $j \leftarrow 1$ to $l$ do
4:     $sh_j \leftarrow \emptyset$
5: end for
6: $\Delta \leftarrow l!$
7: Calculate $S = \{S_1, \dots, S_z\}$, where $z = \binom{l}{k}, \forall S_j \in S, |S_j| = k, S_j \subseteq \{1, \dots, l\}$, and each $S_j$ is distinct
8: $m = \phi(n)/4$
9: Create $f(X) = d + \sum_{j=1}^{k-1} a_j X^j$, where $\forall a_j \xleftarrow{R} \mathbb{Z}$
10: for all $S_\alpha \in S$ do
11:     for all $j \in S_\alpha$ do
12:         Calculate $\lambda_{0,j}^{S_\alpha}$
13:         $exp \leftarrow 4 \cdot \Delta \cdot f(j) \cdot \lambda_{0,j}^{S_\alpha} \bmod m$
14:         $sh_j \leftarrow sh_j \cup (i, S_\alpha, exp)$
15:     end for
16: end for
17: $(a, b) \leftarrow ecgd(e, 4\Delta^2)$, where $a4\Delta^2 + be = 1$
18: $sh_i \leftarrow sh_i \cup (a \bmod m, b \bmod m)$
19: return $sh_1, \dots, sh_l$
```

to establish initial inter-VM SSL channels. We assume that the VMs, and the initial set of SSL channels are provisioned securely, i.e., no adversarial attack occurs until the initial set of SSL channels are established for inter-VM communications. This is a reasonable assumption, since (i) locating defender VMs on the cloud takes time [41], and (ii) the whole process of bootstrapping takes short time, especially if key-partitioning is performed beforehand. Once the initial inter-VM SSL channels are established, HERMES gets ready to serve the clients. Note that a defender VM uses the same RSA key pair for inter-VM and client connections.

Finally, in HERMES, we assume that the number of VMs is fixed throughout the entire life-time of execution. However, to augment HERMES capabilities with dynamic expansion of the system, one should care about the bootstrapping of those new VMs in terms of planting the initial secrets and initiating secure channels. As will be clear in §4.6, during the key re-sharing process, the defender may hand in secret shares to the newly added VMs. Still, introducing dynamic expansion via new VMs may lead to security vulnerabilities that should be investigated thoroughly.

4.4 Connecting to a Client

Once the bootstrapping stage is over, a client or the defender may request connection to a defender VM (i) to consume the services offered by the defender, or (ii) to distribute new shares for the private keys. In any case, the connection is established using server-only verified Enhanced SSL, where the connected VM takes the role of the combiner VM.

Assume the client wishes to connect to VM_i using Enhanced SSL. Throughout the handshaking process, VM_i interacts with the auxiliary VMs (i.e., all VMs other than VM_i), and performs distributed signing or decryption procedures as described in §4.1. The whole distributed operations are transparent to the client, while the combiner or any auxiliary VM learns nothing, but the result.

4.5 Inter-VM Key Renegotiation

Over time, any two defender VMs may decide to end one SSL session, and renegotiate keys for the next one. In such a case, those two VMs use their RSA key pairs, and perform a new handshaking process using our Enhanced SSL with mutual verification. Assume VM_i and VM_j decides to perform renegotiation, where VM_i and VM_j act as the client and server, respectively. Both VMs execute our Enhanced SSL handshaking process using already established SSL channels with the auxiliary VMs. When VM_i or VM_j needs to perform exponentiation with its private key, it collaborates with the auxiliary VMs, and calculates the result.

HERMES allows only one simultaneous key renegotiation at a given time, since an on-going process necessitates already established SSL channels. When two defender VMs start the process, it issues a warning to all VMs, blocking any other attempt for key renegotiation. Once the on-going procedure halts, HERMES removes the warning and allows the first renegotiation attempt.

4.6 Key Re-sharing

At the end of each epoch, the defender creates new shares for the same private RSA keys that were partitioned and distributed in the bootstrapping stage. In essence, it uses the key-partitioning algorithm discussed in §4.2 and generates shares that are independent from the previous ones. Then, it simply connects to each defender VM with our Enhanced SSL, as in §4.4, and hands in the new shares for all partitioned private keys.

The reason to adhere such a process is to mitigate the risk of private key disclosure, since the adversary may have already captured a set of shares for a partitioned private key. It is obvious that partitioning the same key for the second time will result in a different set of shares, which are totally independent from the previous. Hence, if the adversary did not capture enough shares to identify the exact key in one epoch, it will have to start from scratch, since those captured shares mean nothing in the next

		Setup									
		(1,1)	(2,2)	(3,3)	(4,4)	(5,5)	(6,6)	(7,7)	(8,8)	(9,9)	(10,10)
1 Cli.	Total	2.65	8.96	10.40	11.57	15.40	16.13	13.58	17.44	14.05	13.76
	Network	–	2.77	2.82	2.75	2.29	2.54	2.37	1.89	1.07	1.56
	Combine	–	1.78	1.76	1.82	2.25	2.18	1.87	1.99	2.00	1.93
10 Cli.	Total	5.54	31.74	37.85	45.87	43.59	40.82	48.65	52.77	50.64	52.82
	Network	–	19.42	21.67	14.29	18.92	19.89	25.69	21.68	18.11	14.09
	Combine	–	1.95	2.05	2.22	2.20	2.18	1.87	2.17	2.58	2.23
100 Cli.	Total	40.90	179.01	178.14	187.67	209.74	212.33	229.38	246.39	257.03	269.73
	Network	–	121.05	113.36	122.96	108.82	125.52	108.25	106.98	98.71	113.15
	Combine	–	2.16	2.14	2.01	2.10	2.09	2.03	2.11	2.81	2.28
1000 Cli.	Total	146.94	640.40	728.56	928.75	1023.75	904.59	989.32	1097.64	1001.06	1174.54
	Network	–	210.36	197.28	202.08	229.03	240.42	204.30	284.05	237.72	233.41
	Combine	–	2.26	2.08	1.96	2.18	2.17	2.20	2.43	2.24	2.62

Table 1: Average Connection, Network, and Combining Time Spent for D-RSA in milliseconds

epoch. The defender VMs do not immediately start using the new keys, since each defender should get the new shares, otherwise HERMES would have synchronization problems. Instead, a defender VM broadcasts a message to announce that it has the new shares. When all defender VMs have the new shares, they pass on to the next epoch, start using the new shares, and zeroise the old shares to leave no trace. Till then, the VMs continue using the *old* epoch shares.

5 Evaluation

We have implemented a prototype of HERMES atop the most commonly used open source SSL library, OpenSSL [10] v1.0.1e, the latest version as of this writing. Our implementation is a separate *shared library* compatible with the OpenSSL's *Engine API*. Without changing the OpenSSL source, programmers can plug-in our implementation and vary the way that RSA computations are performed with the private key. Meanwhile, we have also created multi-threaded applications (i) for the auxiliary VMs to establish SSL connections with the combiner VM, and to perform mathematical operations (e.g., exponentiation with the private key share); (ii) for the defender to partition the RSA private keys and hand in the shares to each defender VM. In this section, we present our evaluation result.

5.1 Experiment Setup

Case Studies. As it is challenging to exhaustively test HERMES with all the network benchmarks, we evaluated our system using a micro benchmark to profile the performance, and two representative case studies, in which SSL connection is necessary. The micro benchmark experiments evaluate the performance under varying system setups to target possible bottlenecks. Once the system dynamics are profiled, we execute two real-life case studies and check any efficiency deficits. The first case study is a web server, for which we used Apache HTTP Server [7] v2.4.4. A client connects to the server via HTTPS, and retrieves the default web page that comes with the application, which is a static HTML page of size $2KB$. The second case study is a mail server using Postfix v2.10 [11]. On top of that, we installed Dovecot [8] v2.2.4 as the IMAP(s)/POP3(s) server. A client connects to the Dovecot instance via IMAPS and checks the status of a mailbox, which contains a single mail of size $1KB$. Both server applications are executed with the *keep alive* property off (i.e., the server does not store SSL sessions, and performs a new handshake for every connection attempt by the clients).

One may argue that testing the web and mail servers with such low-sized content is applicable to real-world case. It is true that almost all web sites serve contents that may have much larger sizes. However, the purpose of the experiments is to put as much pressure as possible to HERMES in the given web and mail server case studies. As will be clear in the results, as the number of connections performed per unit time increases, HERMES acts more efficiently due to decreased network overhead. Hence, increasing the content sizes would increase the amount of time the server spends on processing a query, and decrease the number of requests per unit time. Instead, we used 1 and 2KB contents, and tried pushing HERMES as much as possible.

Benchmarks. To extract micro benchmark results, we developed applications that connect to the given defender VM, using given number of concurrent clients. For the web server application, we used two different benchmarking tools: Apache HTTP server benchmarking tool [5] (AB) v2.4.4, which allows us to send HTTPS queries with a variety of execution options; and Apache JMeter [6] (AJ) v2.9, where we used the default HTTPS request sampler that comes with the standard AJ binaries. For the mail server application, we used AJ again, with the default mail reader sampler. Similar to the server-side, we

		Setup								
		(10,2)	(10,3)	(10,4)	(10,5)	(10,6)	(10,7)	(10,8)	(10,9)	(10,10)
1 Clients	Total	12.06	12.27	13.44	11.57	16.10	17.94	16.80	19.80	13.76
	Network	4.97	5.14	4.89	2.75	5.07	5.59	5.29	1.15	1.56
	Combine	0.52	0.56	0.58	1.82	2.36	1.56	2.44	2.20	1.93
10 Clients	Total	19.78	23.22	28.14	45.87	39.48	48.99	49.97	60.70	52.82
	Network	9.72	10.15	10.19	14.29	16.30	23.15	27.17	34.03	14.09
	Combine	1.27	1.07	1.26	2.22	2.42	2.64	3.09	2.81	2.23
100 Clients	Total	54.90	71.07	88.31	187.67	130.17	163.24	182.12	206.00	269.73
	Network	11.77	25.27	37.77	122.96	69.74	84.05	82.96	121.32	113.15
	Combine	1.24	1.62	1.74	2.01	2.15	2.34	2.80	3.01	2.28
1000 Clients	Total	318.24	418.07	435.98	928.75	653.12	642.48	877.42	995.89	1174.54
	Network	88.12	123.72	130.21	202.08	196.29	212.05	214.20	216.97	233.41
	Combine	1.50	2.20	1.85	1.96	2.08	2.52	2.82	3.24	2.62

Table 2: Average Connection, Network, and Combining Time Spent for Fixed $l = 10$ in milliseconds

did not use the keep alive functionality in the benchmarking tools, which forces the clients to perform a new SSL handshaking for each request.

Hardware. For our experiments, we created 10 VM instances on Amazon EC2. The VM that serves that is augmented with the web and mail server applications is of the type *m1.xlarge* with 4 virtual CPU and 15GB of RAM. The remaining VMs are of the type *m1.small* with 1 virtual CPU, 1 virtual core, 1.7GB of RAM, and 64-bit Red Hat Enterprise Linux 6.4. The reason that we don't perform experiments with more number of VMs is that our results for 10 VMs are enough to extrapolate relations between HERMES modes, parameters (e.g., l, k, τ), and the performance metrics (e.g., average latency, throughput). All instances are created in the same EC2 region, US-West at Oregon. The instances communicate with each other over Amazon's private network, while a client or the defender interacts with the VMs over the public network. On the other hand, we used a single machine to send connection, web page, or mail check queries, where the machine is an IBM x3500m3 server with 16GB of RAM, and 4 quad-core CPUs at 2.4 GHz. Our client machine is located in our university campus, and is connected to the defender VMs over the Internet.

Parameters. We vary the number of concurrent clients from 1 to 1000 exponentially to observe the effects of increasing load on HERMES. We believe that the number 1000 is enough, since the number of web page views for most popular web sites goes up to 37 billion per year, which is approximately 1100 per second [1,2]. Each experiment ran for 5 minutes, and the average value of 5 runs is shown as the final result. As will be shown in the following subsection, we observe that a key re-sharing process takes approximately 50 *msec*. Combined with the observation that the average time to process a query may go up to 2 *sec*, we vary τ (i.e., the key re-sharing period) from 5 to 125 seconds.

We perform experiments using 10 VMs, and represent the setup as (l,k), where l is the number of active VMs, and k is the number of shares needed to calculate the RSA result. When l is equal to k, the system runs with D-RSA mode of operation using l VMs. Furthermore, $(1,1)$ represents the **single VM setup**, where the *default SSL (i.e., the one without our modification and there is only one key)* is used. Also, as l must be greater than or equal to k, it is important to note that we do not have any experiment set up of (l,k) where $l < k$.

5.2 Results

Micro Benchmarking. In this set of experiments, we aimed to observe the sole effects of HERMES on the performance, where the client simply connects to the combiner VM, and immediately closes the SSL channel, without sending any additional query. Naturally, we expected to observe a massive load on the combiner VM, since all it does is to establish SSL channel with the client using our enhanced SSL, and nothing else. Thus, the number of requests per unit time will be high, which will introduce an increased network overhead.

Table 1 shows the micro benchmark results for D-RSA with up to 10 VMs (e.g., $l = 10$). We vary the number of concurrent clients from 1 to 1000, and measure the average connection time, average time spent for inter-VM communication, and average time spent for combining partial results in milliseconds. It is observed that combining partial results from the auxiliary VMs do not incur more than 3 *msec* overhead; thus, does not affect a successful enhanced SSL connection in terms of efficiency. The reason is the simplicity of combining partial results (i.e., l modular multiplication). On the other hand, inter-VM communication dominates the overhead introduced by the enhanced SSL in D-RSA mode. Especially when the number of concurrent clients is 1000, average time required to execute an SSL connection exceeds 1 *sec* if $l > 7$. Thus, the network communication becomes the bot-

		Setup								
		(2,2)	(3,2)	(4,2)	(5,2)	(6,2)	(7,2)	(8,2)	(9,2)	(10,2)
1 Clients	Total	8.96	12.03	11.23	11.97	12.37	13.47	12.16	9.58	12.06
	Network	2.77	4.77	4.82	5.28	5.10	4.77	5.42	4.76	4.97
	Combine	1.78	2.13	0.90	1.30	1.35	0.51	0.53	0.55	0.52
10 Clients	Total	31.74	33.45	23.77	25.34	23.57	20.18	18.81	19.26	19.78
	Network	19.42	19.10	11.00	9.78	9.62	8.20	10.49	9.26	9.72
	Combine	1.95	2.26	1.52	1.47	1.26	1.50	1.29	1.33	1.27
100 Clients	Total	179.01	164.65	95.30	82.50	80.98	66.95	73.26	58.08	54.90
	Network	121.05	95.84	52.52	38.03	30.00	25.90	25.93	25.26	11.77
	Combine	2.16	2.19	1.82	1.92	1.42	1.59	1.72	1.15	1.24
1000 Clients	Total	640.40	665.95	548.22	504.12	450.09	340.10	350.46	320.84	318.24
	Network	210.36	197.43	150.84	123.75	60.88	55.09	59.19	46.50	88.12
	Combine	2.26	1.93	1.91	1.91	1.55	1.41	1.42	2.36	1.50

Table 3: Average Connection, Network, and Combining Time Spent for Fixed $k = 2$ in milliseconds.

tleneck for D-RSA in high load, in case the combiner VM closes the connection right after a successful connection. In the results for our case studies, we observe that if the combiner has to process a request (e.g., prepare a web page, or check a mailbox) after a successful SSL connection, the network overhead decreases, which results in less average latency.

As previously mentioned, we introduce T-RSA mode to reduce the overhead by simply distributing work amongst different sets of VMs. Given the performance of 10 VMs in D-RSA mode, we check if the performance can be improved in T-RSA mode by reducing k (i.e., the number of needed VMs). Table 2 shows the results for a fixed $l = 10$ and varying number of k values. Furthermore, we perform experiments to observe the effect of increasing number of VMs for a fixed $k = 2$, and show the results in Table 3. The performance metrics and the client parameters are the same as in Table 1. It is observed that for a fixed l value, the average latency to complete an enhanced SSL connection drops down as k gets smaller, especially when $k \leq (l/2)$. The reason is that different sets of auxiliary VMs are consulted to complete a single SSL connection each time, which results in less network connection overhead. Hence, per each inter-VM connection, we observe less load, resulting up to 3 times better performance than D-RSA mode with same l value (e.g., between $(10,10)$ and $(10,2)$). On the other hand, it is still reasonable to pass to T-RSA, even if $k > (l/2)$, since decreasing k has, definitely, positive effects on the performance. For a fixed k value, increasing l by introducing new defender VMs has positive effects on the average time to complete an enhanced SSL connection, by simply reducing the inter-VM communication overhead. The value of k, indeed, affects the number of VMs that should be introduced to reduce the average completion time. We extrapolate that introducing nearly $2k$ new VMs into HERMES helps decreasing the overhead by nearly 50%. To solidify our derivations, we performed the same T-RSA experiments for different fixed values of k and l. For brevity, we moved the results of those experiments to App. A, from which the same observations can be easily made.

Key re-sharing time (*msec*)					
l	*Avg.Lat.*	l	*Avg.Lat.*	l	*Avg.Lat.*
2	17.59	5	37.89	8	54.64
3	23.60	6	43.65	9	57.34
4	29.51	7	49.82	10	61.28

Table 4: Average Completion Time for Key Re-sharing in milliseconds.

We, further, measured the average time of completion for a single key re-sharing process for varying number of defender VMs. Since, the number of connections that the defender has to do in the re-sharing process depends on l, but not on k, we performed the experiments in D-RSA mode with l number of VMs. Table 4 shows the results, where the defender re-shares the same partitioned keys every 5 seconds, and no other client attempts to connect to the VMs. The experiments ran for 5 minutes, and the average time to complete a single key re-sharing is calculated. We observe that the average time increases with the number of VMs, since the defender has to connect each VM separately, incurring additional inter-VM communication overhead. We see that the values in Table 4 coincides with the values in Table 1. When l is equal to 10, the defender has to make 10 simultaneous connections to HERMES, resulting a similar result as 10 clients D-RSA for the setup $(10,10)$. In case of high load (e.g., 1000 clients), the key re-sharing process would, definitely, take longer time. Thus, the optimal τ value for the key re-sharing epoch should be chosen while considering the server load, and the number of defender VMs.

Web Server. In our first case study, our aim is to show that the performance improves as the combiner executes a CPU intensive operation (e.g., prepare a web page) once connected to the client. The experimental setup is the same as the micro benchmarking setup, except now the combiner VM is a web server. When a client employs

SSL to connect to the combiner VM and retrieve a web page, the combiner VM collaborates with the auxiliary VMs, and executes our enhanced SSL.

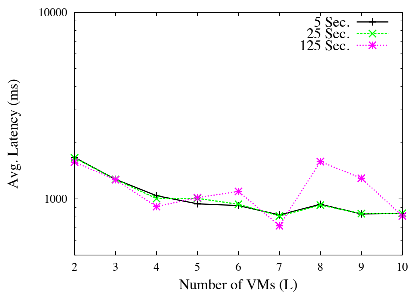
Fig. 4a and 4d show the results for HERMES in D-RSA mode, where the number of VMs changes from 1 to 10, and the number of concurrent clients changes between 1 to 1000 for τ of 125 *sec*. We use AB and AJ benchmarking tools, run the experiments for 5 minutes, and report the average time needed to execute a web page retrieval request, and the number of requests per second. We observe similar performance patterns for both of our metrics (e.g., performance decrease when l is increased) in compare to the micro benchmarks. However, the performance difference between the two end points (i.e., between $(1,1)$ and $(10,10)$) is narrower, due to more CPU-intensive processing done by the combiner. For 1000 concurrent clients, average latency and throughput in $(1,1)$ is 740 *msec* and 255 req/sec, respectively. On the other hand, the $(10,10)$ setup results in nearly 2 *sec* average latency, and 120 req/sec throughput. Compared to nearly 10 times increase in the micro benchmarking results, we see that the more CPU-intensive job the server does, the closer the gap between the $(1,1)$ and $(10,10)$ setups is.

Once again, we check if the performance can be boosted by passing to T-RSA mode, with decreased number of needed VMs. Fig. 4b and 4e show the results for fixed $l = 10$ and $\tau = 125$ *sec*. We observe that especially when $k \leq (l/2)$, the overhead reduces down to nearly 10% with respect to $(1,1)$ setup. For instance, in the $(10,5)$ setup, the average latency is 1088 ms, while the throughput is 220 req/sec. Even better, the throughput increases to 248 in $(10,3)$ setup, and to 250 in $(10,2)$ setup, which is just 2% less than $(1,1)$. The reason stems from distributing workload to more VMs by keeping seperate parts of the network busy at the same time, which reduces the inter-VM communication overhead.

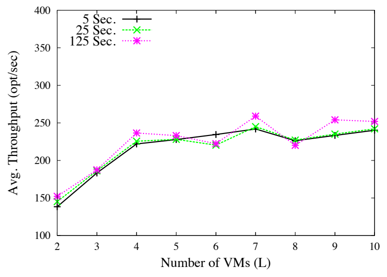
We remark that the results are gathered using the second *slowest* VM instances in Amazon EC2. The defender can instantiate stronger VM instances, with faster network, which will definitely improve the performance, since the network latency turns out to be the bottleneck. Furthermore, the defender can distribute the combiner role to multiple VMs to achieve further workload distribution.

The next results for the web server case study are given in Fig. 4c and 4f, where we measure the performance for varying l parameter and a fixed $k = 2$. We observe that having $l > 2k$ boosts the performance. Even in the $(4,2)$ setup, we measure that the average latency and throughput is 909 *msec* and 236 req/sec, respectively, which means less than 10% overhead for the second metric. When the number of VMs is more than $3k$, HERMES performs nearly the same as the $(1,1)$ setup.

To show that our choice of $\tau = 125$ *sec* does not have major effects on the overall performance, we vary the length of an epoch exponentially from 5 to 125 *sec* for different number of VMs, and fixed number of concurrent clients of 1000. We chose to execute epoch experiments for the fastest HERMES setup, namely fixed k with high l values, and to check if performance degradation occurs for decreased key re-sharing period. Fig. 5 shows the results for fixed $k = 2$ and 1000 concurrent clients, and varying τ values. We observe that even when $\tau = 5$ *sec*, the performance metrics behave similar to $\tau = 125$ *sec* case. This stems from the server being already loaded with enough concurrent clients, so that the seldom requests to re-share keys are only minor issues that does not take too much time to process.



(a) Avg. Latency for Epoch Times

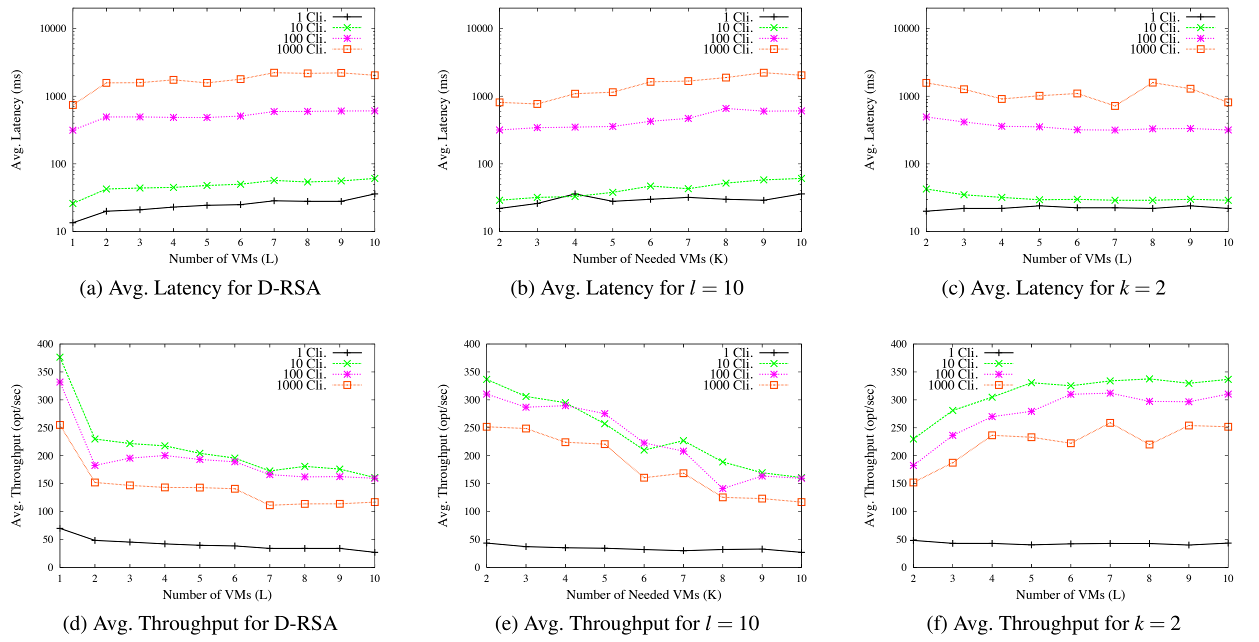


(b) Avg. Throughput for Epoch Times

Figure 5: Web Server results for $k = 2$ with varying τ

Mail Server. Mail Server is our second case study, where the clients establish connection using SSL via IMAPS protocol, and check a mailbox that contains a single mail. The default setting with regular SSL (i.e., $(1,1)$ setup) already results in an average 5758 *ms* latency, and 49.5 req/sec for 1000 concurrent clients. Hence, the combiner has to do more CPU-intensive operation for each client request. We claim that the margin between $(1,1)$ and $(10,10)$ setups will be less, since the network will be less occupied at a given time; thus, it will result in less inter-VM communication overhead.

Fig. 6 shows the results for the mail server case study, where the clients vary from 1 to 1000, and re-sharing period is 125 *sec*. First of all, we observe that the performance of each setup looks very similar, with nearly at most 8% overhead with respect to $(1,1)$. The reason to



(a) Avg. Latency for D-RSA (b) Avg. Latency for $l = 10$ (c) Avg. Latency for $k = 2$

(d) Avg. Throughput for D-RSA (e) Avg. Throughput for $l = 10$ (f) Avg. Throughput for $k = 2$

Figure 4: Web Server results

observe such a pattern is, as hypothesized, the fact that an average mail inquiry takes too much time to process. It causes low throughput values, resulting less number of SSL handshakes being made per unit time, which in turn causes less inter-VM communication overhead.

Once more, we observe that increasing the number of VMs for D-RSA mode has negative effects on the performance metrics, as shown in Fig. 6a and 6d. On the contrary, increasing l for fixed k value in T-RSA mode enhances the overall performance, due to better distribution of the defender VMs, as seen in Fig. 6c and 6f.

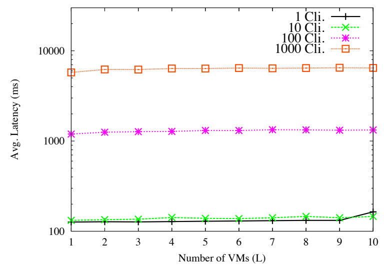
6 Security Analysis

The theoretical security of HERMES is based on the formally proven security of D-RSA and T-RSA, as discussed in §2.2. Combined with key re-sharing, the adversary should successfully capture at least l shares in D-RSA or k in T-RSA to calculate the shared cryptographic key. On the other hand, in practice, HERMES should give guarantees on the probability of a successful attack based on some assumptions on the nature of the attack and the system parameters (e.g., τ, l, k). The defender may have limited budget, or have certain performance requirements. In any case, HERMES must minimize any security risk by choosing l, k, and τ *optimally*. In this section, we first formalize the problem of finding such *optimal* values for those parameters, and then apply the optimization technique to a sample configuration: the micro benchmarking scenario discussed in §5. Our choice to apply optimization to only one configuration is due to space constraints; however, our approach is modular, and is easily applicable to any other cases.

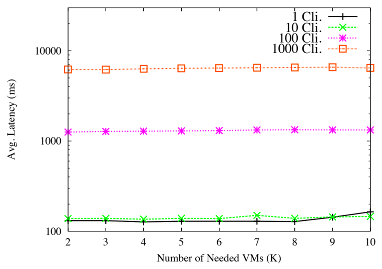
6.1 Problem Formalization

In our formalization, we consider three main aspects: security, cost, and performance. Security aspect allows us to provide an upper bound on the possibility of a successful key extraction attack on HERMES for the given k, l, and τ values. Theoretically, increasing k and l, or decreasing τ will make it harder for the adversary to achieve its goal. However, increasing l implies more defender VMs running on the cloud, which increases the total cost. Moreover, our experiments showed that the performance degrades as l and k increase together. Hence, the optimal values should be assigned to k, l, τ for the given constraints (e.g., budget, performance limit).

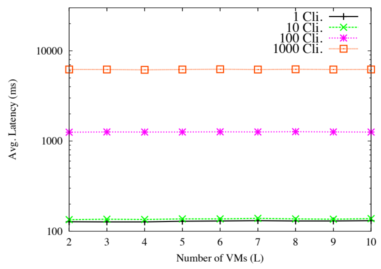
Security Aspect. To quantify the probability of a successful attack in an epoch, we assume that the adversary has to start from scratch in each epoch, which implies that it loses all its previously acquired information. This is a valid assumption, since shares for each epoch are independent from one another, and a captured share does not contribute any information to the next epoch. The inability of conducting acquired information to the following epochs makes it convincing to model the probability of a successful attack as an *exponentially distributed* ran-



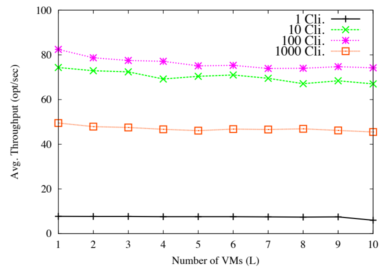
(a) Avg. Latency for D-RSA



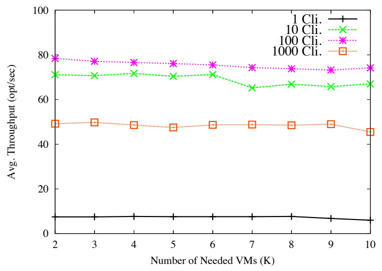
(b) Avg. Latency for $l = 10$



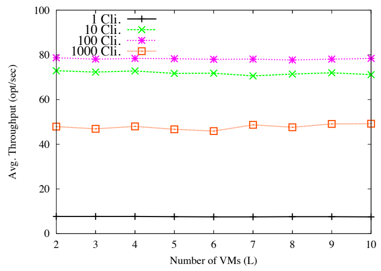
(c) Avg. Latency for $k = 2$



(d) Avg. Throughput for D-RSA



(e) Avg. Throughput for $l = 10$



(f) Avg. Throughput for $k = 2$

Figure 6: Mail Server results

dom variable. Given the success rate parameter θ, the probability distribution for the attack is:

$$f(t) = \begin{cases} \frac{1}{\theta} e^{-t/\theta} & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Since the exponential distribution is *memoryless*[1] and the cryptographic key is re-shared in each epoch, we can simply assume that the input to f is the time difference from the last re-sharing moment. Then, given the length of the epoch τ, the probability of a successful attack is:

$$F(\tau, \theta) = \int_0^\tau f(t).dt = 1 - e^{-\tau/\theta} \quad (2)$$

Finally, assuming that the probability of capturing shares from a single VM is identical to and independent from all other VMs, the probability of capturing at least k shares from l defender VMs in an epoch is:

$$Sec(l,k,\tau,\theta) = \sum_{i=k}^{l} \binom{l}{i} (1 - e^{-\tau/\theta})^i (e^{-\tau/\theta})^{l-i} \quad (3)$$

Cost Aspect. Modeling monetary cost in HERMES is rather simple compared to the other two aspects. Assuming that the cloud provider does not charge money for the inter-VM communications, the total monetary cost is $Cost(l) = l.\beta$, where β is the unit cost of running a single VM on the cloud provider. The cost of communication with the client is also neglected, since this is not an additional cost incurred by HERMES.

Performance Aspect. The method to formalize the expected performance depends heavily on the application that HERMES is running for, and the metrics that the defender considers. For instance, one may value throughput more than the latency while running HERMES. On the other hand, the effects of changing parameters (i.e., k, l) in the mail server case study is far different than changing the same parameters in the micro benchmarking experiments. For brevity, we show the performance of HERMES for the given k and l as $Perf(l,k)$, and leave it to the defender to define the characteristics of the function.

Optimization Problem. Given the success rate parameter θ, the unit cost of a VM β, the budget limit L_{cost}, and the performance limit L_{perf}, the aim of the optimization problem is to minimize the probability of a successful attack in an epoch while keeping the total monetary cost below L_{cost} and the performance below L_{perf}. Formally, the optimization problem is:

$$\begin{aligned} \text{minimize:} \quad & Sec(l,k,\tau,\theta) \\ \text{subject to:} \quad & Cost(l) \le L_{cost},\ Perf(l,k) \le L_{perf} \\ & l \ge k > 1,\ \tau > 0 \end{aligned}$$

6.2 Application to Micro Benchmarking

Modeling performance is highly dependent on the case study and the aimed configuration, thus it is challenging to apply the optimization to every single case. Instead, we targeted to optimize HERMES for 100 concurrent clients

	$\theta = 600$		$\theta = 3600$	
L_{cost}/yr	Conf.	Sec()	Conf.	Sec()
\$1820	$(2,2)$	$6.8 \cdot 10^{-5}$	$(2,2)$	$1.9 \cdot 10^{-6}$
\$3640	$(4,3)$	$2.2 \cdot 10^{-6}$	$(4,3)$	$3.7 \cdot 10^{-8}$
\$7280	$(8,5)$	$2.1 \cdot 10^{-9}$	$(8,5)$	$2.8 \cdot 10^{-13}$
\$14560	$(16,10)$	$1.1 \cdot 10^{-17}$	$(16,10)$	$2.1 \cdot 10^{-25}$

Table 5: Optimal setup and resulting successful attack probabilities in an epoch for fixed expected latency limit $L_{perf} = 150$ *msec*, and $\theta = \{600, 3600\}$

	$\theta = 600$		$\theta = 3600$	
L_{perf}	Conf.	Sec()	Conf.	Sec()
50 *msec*	$(16,6)$	$2.4 \cdot 10^{-9}$	$(16,6)$	$5.6 \cdot 10^{-14}$
100 *msec*	$(16,8)$	$2.7 \cdot 10^{-13}$	$(16,8)$	$1.7 \cdot 10^{-19}$
150 *msec*	$(16,10)$	$1.1 \cdot 10^{-17}$	$(16,10)$	$2.1 \cdot 10^{-25}$
200 *msec*	$(16,11)$	$5.4 \cdot 10^{-20}$	$(15,11)$	$4.9 \cdot 10^{-29}$

Table 6: Optimal setup and resulting successful attack probabilities in an epoch for fixed monetary budget $L_{cost} = \$14560/yr$, and $\theta = \{600, 3600\}$

in the micro benchmarking scenario, since all experiment results for the chosen configuration are given in §5.2 and App. B. For brevity, we make a further assignment of parameters by choosing re-sharing period as $\tau = 5$ *sec* and success rate parameter as $\theta = 3600$. $\tau = 5$ *sec* is the smallest value that we have tested, and is a valid value that allows HERMES to complete several computations in each epoch. Furthermore, choosing small re-sharing period will tighten the overall security, since the adversary has to complete the attack in a very short period. On the other hand, choosing θ as 3600 is due to the existing cross-VM attacks (i.e., [41, 51]), which necessitates hours to capture the cryptographic key. In an exponential distribution, expected *waiting time* to observe one success is θ. Since, we expect the attack to succeed in an hour, we assign $\theta = 3600$, representing the number of seconds in an hour. In addition, we check $\theta = 600$ to observe changes in optimal values.

In this example, we picked latency as the target performance metric to consider, assuming that the defender aimed to serve 100 concurrent clients as fast as possible. The important step to model performance is to figure out $Perf(l,k)$. To overcome this, we applied *multiple linear regression* on our experiment results, and came up with a formula that gives the *expected* latency value for the given l and k values. As it is challenging to test every possible formula, and increasing the number of variables may over-fit the training data, we chose a simple polynomial $Perf(l,k) = c_0 + c_1.l + c_2.k + c_3.(l/k)$ to model the expected latency, where the coefficients are $c_0 = 118$, $c_1 = -18$, $c_2 = 31$, and $c_3 = 7$. Finally, to observe the effects of different performance limits L_{perf}, we calculated optimal HERMES setups for $L_{perf} \in [50, 200]$. Finally, assuming that the defender will use the second cheapest VM instance on Amazon EC2, she will pay \$0.104/*hour*, which is approximately \$910/*yr*. We vary the monetary budget between \$1820/*yr* and \$14560/*yr* to check optimal values, which is simply $l \in [2, 16]$.

Table 5 shows the results of the optimization procedure for varying monetary budget, and fixed $L_{perf} = 150$. The results include the optimal HERMES setup and the probability of a successful attack in one epoch, for both $\theta = 3600$ and 600. We observe that as we increase the monetary budget, HERMES is allowed to run with more VMs, resulting in lower probabilities of success for the adversary. For instance, when the budget is \$7280/*yr* and $\theta = 3600$, HERMES can be configured to run in $(8,5)$ setup, while the adversary has only $2.8 \cdot 10^{-13}$ chance to capture the partitioned cryptographic key.

Table 6 shows similar set of results, this time for fixed monetary budget of \$14560/*yr*, but varying expected performance limit L_{perf}. We deduce that as HERMES is allowed to respond slower, it can be configured to run with increased k, which decreases the attack success probability. For instance, increasing expected latency from 50 *msec* to 150 *msec* decreases the attack success probability nearly 8 and 11 fold, when θ is 600 and 3600, respectively.

Note that the probabilities are calculated for a successful attack in **one epoch**, one would question if the adversary would accomplish its goal in a longer period, say a year. For \$7280/*yr* and an expected latency of 150 *msec* in our micro benchmarking case study, the probability of capturing the complete cryptographic key in one year is $1.8 \cdot 10^{-6}$ if the average time to capture a key is predicted as 3600 *sec*, which is a very small probability.

7 Related Work

Attacks. There exists a myriad of side-channel attacks with different assumptions and setups. The adversary may leverage observations made on the shared hardware to execute *access driven* attacks (e.g., [13, 14, 28, 43]); measure timings of certain cryptographic operations of the defender to perform *time-driven* attacks (e.g., [15, 19, 20, 34, 47]); or physically observe the defender machines and run *trace driven* attacks (e.g., [18, 26, 33]). The specific type of attacks, which HERMES aims to mitigate, is *cross-VM* side-channel attacks, in which both the defender and the adversary are customers to a third-party cloud infrastructure. Both entities initialize VMs in the cloud, where the victim's VMs are attacked by the adversary's VMs. Ristenpart et al. [41] showed the first cross-VM side-channel, in which they first *co-reside* their VMs with the defender VMs, and execute an access-driven attack to retrieve crude information (e.g., aggregate cache usage). In another work, though not for adversarial purposes, Zhang et al. [50] present *HomeAlone* that performs a co-residency check between two VMs using classifiers on cache timing. In another attack, Zhang et al. push it one step further, and extract an ElGamal private key using cross-VM attacks [51]. Those works showed that VMs in public cloud infrastructures are vulnerable to side-channel attacks, and protection mechanisms are needed to secure private information.

Prevention. Among the variety of side-channel prevention techniques, the most popular ones are randomization-based approaches. *MIST* is one of such examples, in which the *square-and-multiply* method is extended with an additional division by a randomly chosen number [38, 45]. Other approaches include adding random noise between squaring and multiplying operations, or applying *always-multiply* techniques. To countermeasure those side-channel prevention techniques, Karlof et al. promotes *Hidden Markov Model* based cryptanalysis as a powerful tool [30]. On the other hand, Witteman et al. shows a trace driven side-channel attack to break down *always-multiply* technique and message binding in RSA [49]. Although the latter is trace driven, those two works show that even randomization based side-channel prevention approaches could have vulnerabilities that can be used by different types of adversaries.

There exist several works that aim to prevent side-channel attacks in public clouds. *HomeAlone* [50] uses co-residency checks to see if a VM is physically isolated from any other VM, and to achieve maximum physical isolation. Our work aims to prevent the leakage of private keys even if the adversary co-resides with the defender, whereas they aim to prevent access-driven side-channel attacks by assuring physical isolation. In *HyperSafe*, Wang and Jiang aim to provide hypervisor integrity throughout the execution [46]. We assume that the cloud provider and its infrastructure (including the hypervisor) are trusted. Other prevention mechanisms include [17], which aims to prevent side-channel attacks that use communication traffic; *StealthMem* that hides memory access patterns to protect private information [32]. Compared to these works, HERMES is applicable to any type of cross-VM attacks against cryptographic keys.

8 Conclusion

In this paper, we present HERMES, a novel system to protect cryptographic keys in cloud VMs. The key idea is to periodically partition a cryptographic key using additive or Shamir secret sharing. With two different case studies, we show that the overhead can be as low as 1%. With such small overhead in an average request, cryptographic keys become more leakage-resilient against any adversary. Furthermore, we model the problem of finding *optimal* parameters for the given monetary and performance constraints, which minimizes the security risk. Using our formal model, the defender can calculate the probability of a successful attack, and take precautions (e.g., increase the number of VMs, decrease epoch length). As a proof-of-concept, the current implementation of HERMES mainly focuses on the protection of the RSA private key, which is widely used in many daily web site and mail server communications. However, there exists a myriad of works on threshold signature schemes for different cryptosystems, (e.g., [21, 23, 27, 35, 36]), which may be applicable to HERMES with slight modifications.

9 Acknowledgement

We thank the anonymous reviewers for their insightful comments. This work was partially supported by Air Force Office of Scientific Research FA9550-12-1-0082 and FA9550-14-1-0119, National Institutes of Health Grants 1R0-1LM009989 and 1R01HG006844, National Science Foundation (NSF) Grants Career-CNS-0845803, CNS-0964350, CNS-1016343, CNS-1111529, CNS-1228198 and Army Research Office Grant W911NF-12-1-0558.

References

[1] Us web statistics released for may 2012: which sites dominate, and where do we go for online news? http://www.theguardian.com/news/datablog/2012/jun/22/website-visitor-statistics-nielsen-may-2012-google, 2012.

[2] Internet 2012 in numbers. http://royal.pingdom.com/2013/01/16/internet-2012-in-numbers/, 2013.

[3] Public, private and hybrid clouds when, why and how they are really used. Tech. rep., Summary report, Neovise, 2013.

[4] Amazon elastic compute cloud. http://aws.amazon.com/pricing/ec2/, 2014.

[5] Apache http server benchmarking tool. http://httpd.apache.org/docs/2.4/programs/ab.html, 2014.

[6] The apache jmeter desktop application. http://jmeter.apache.org/, 2014.

[7] Apache: The number one http server on the internet. http://httpd.apache.org/, 2014.

[8] Dovecot, an open source imap and pop3 email server. http://www.dovecot.org, 2014.

[9] Google compute engine. https://cloud.google.com/products/compute-engine, 2014.

[10] The openssl project. http://www.openssl.org, 2014.

[11] The postfix home page. http://www.postfix.org/start.html, 2014.

[12] Windows azure. http://www.windowsazure.com/en-us/, 2014.

[13] ACIIÇMEZ, O., BRUMLEY, B. B., AND GRABHER, P. New results on instruction cache attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer, 2010, pp. 110–124.

[14] ACIIÇMEZ, O., KOÇ, Ç. K., AND SEIFERT, J.-P. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security* (2007), ACM, pp. 312–320.

[15] ACIIÇMEZ, O., SCHINDLER, W., AND KOÇ, Ç. K. Cache based remote timing attack on the aes. In *Topics in Cryptology–CT-RSA 2007*. Springer, 2006, pp. 271–286.

[16] ALLEN, C., AND DIERKS, T. The tls protocol version 1.0.

[17] BACKES, M., DOYCHEV, G., AND KOPF, B. Preventing side-channel leaks in web traffic: A formal approach. In *NDSS* (2013).

[18] BERTONI, G., ZACCARIA, V., BREVEGLIERI, L., MONCHIERO, M., AND PALERMO, G. Aes power attack based on induced cache miss and countermeasure. In *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on* (2005), vol. 1, IEEE, pp. 586–591.

[19] BRUMLEY, B. B., AND TUVERI, N. Remote timing attacks are still practical. In *Computer Security–ESORICS 2011*. Springer, 2011, pp. 355–371.

[20] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. *Computer Networks 48*, 5 (2005), 701–716.

[21] DESMEDT, Y. Some recent research aspects of threshold cryptography. In *Information Security*. Springer, 1998, pp. 158–173.

[22] DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *Information Theory, IEEE Transactions on 22*, 6 (1976), 644–654.

[23] FOUQUE, P.-A., AND POINTCHEVAL, D. Threshold cryptosystems secure against chosen-ciphertext attacks. In *Advances in CryptologyASIACRYPT 2001*. Springer, 2001, pp. 351–368.

[24] FRANKEL, Y. A practical protocol for large group oriented networks. In *Advances in Cryptology EUROCRYPT 1989* (1990), Springer, pp. 56–61.

[25] FREIER, A. The ssl protocol version 3.0. http://ci.nii.ac.jp/naid/10015295976/en/, 1996.

[26] GANDOLFI, K., MOURTEL, C., AND OLIVIER, F. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems CHES 2001* (2001), Springer, pp. 251–261.

[27] GENNARO, R., JARECKI, S., KRAWCZYK, H., AND RABIN, T. Robust threshold dss signatures. In *Advances in CryptologyEUROCRYPT96* (1996), Springer, pp. 354–371.

[28] GULLASCH, D., BANGERTER, E., AND KRENN, S. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 490–505.

[29] HICKMAN, K., AND ELGAMAL, T. The ssl protocol. *Netscape Communications Corp 501* (1995).

[30] KARLOF, C., AND WAGNER, D. Hidden markov model cryptanalysis. In *Cryptographic Hardware and Embedded Systems–CHES 2003: 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings* (2003), vol. 5, Springer, p. 17.

[31] KELLER, E., AND REXFORD, J. The platform as a service model for networking. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (2010), vol. 4, USENIX Association.

[32] KIM, T., PEINADO, M., AND MAINAR-RUIZ, G. Stealthmem: system-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association, pp. 11–11.

[33] KOCHER, P., JAFFE, J., AND JUN, B. Differential power analysis. In *Advances in Cryptology CRYPTO 1999* (1999), Springer, pp. 388–397.

[34] KOCHER, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology CRYPTO 1996* (1996), Springer, pp. 104–113.

[35] KUROSAWA, K. New eigamal type threshold digital signature scheme. *IEICE transactions on fundamentals of electronics, communications and computer sciences 79*, 1 (1996), 86–93.

[36] LANGFORD, S. K. Threshold dss signatures without a trusted party. In *Advances in CryptologyCRYPT095*. Springer, 1995, pp. 397–409.

[37] OPPLIGER, R. *SSL and TLS: Theory and Practice*. Artech House, 2009.

[38] OSWALD, E., AND AIGNER, M. Randomized addition-subtraction chains as a countermeasure against power attacks. In *Cryptographic Hardware and Embedded SystemsCHES 2001* (2001), Springer, pp. 39–50.

[39] OWENS, R., AND WANG, W. Non-interactive os fingerprinting through memory de-duplication technique in virtual machines. In *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International* (2011), IEEE, pp. 1–8.

[40] PRODAN, R., AND OSTERMANN, S. A survey and taxonomy of infrastructure as a service and web hosting cloud providers. In *Grid Computing, 2009 10th IEEE/ACM International Conference on* (2009), IEEE, pp. 17–25.

[41] RISTENPART, T., TROMER, E., SHACHAM, H., AND 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* (2009), ACM, pp. 199–212.

[42] SHOUP, V. Practical threshold signatures. In *Advances in Cryptology EUROCRYPT 2000* (2000), Springer, pp. 207–220.

[43] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology 23*, 1 (2010), 37–71.

[44] VAQUERO, L. M., RODERO-MERINO, L., CACERES, J., AND LINDNER, M. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review 39*, 1 (2008), 50–55.

[45] WALTER, C. D. Mist: An efficient, randomized exponentiation algorithm for resisting power analysis. In *Topics in Cryptology CT-RSA 2002*. Springer, 2002, pp. 53–66.

[46] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 380–395.

[47] WEISS, M., HEINZ, B., AND STUMPF, F. A cache timing attack on aes in virtualization environments. In *Financial Cryptography and Data Security*. Springer, 2012, pp. 314–328.

[48] WINKLER, V. Cloud computing: Cloud security concerns. Tech. rep., 2011.

[49] WITTEMAN, M. F., VAN WOUDENBERG, J. G., AND MENARINI, F. Defeating rsa multiply-always and message blinding countermeasures. In *Topics in Cryptology–CT-RSA 2011*. Springer, 2011, pp. 77–88.

[50] ZHANG, Y., JUELS, A., OPREA, A., AND REITER, M. K. Homealone: Co-residency detection in the cloud via side-channel analysis. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 313–328.

[51] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 305–316.

		Setup							
		(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)	(9,9)
1 Clients	Total	9.58	10.70	11.53	15.88	13.93	13.97	14.13	14.05
	Network	4.76	4.49	4.85	5.30	5.01	4.84	2.45	1.07
	Combine	0.55	1.44	1.18	1.65	1.77	1.65	2.75	2.00
10 Clients	Total	19.26	21.29	24.14	31.43	32.81	39.67	46.73	50.64
	Network	9.26	10.84	12.87	15.85	14.55	22.92	23.74	18.11
	Combine	1.33	1.51	1.68	2.07	2.61	2.83	2.76	2.58
100 Clients	Total	58.08	68.32	91.87	144.08	164.54	209.26	247.54	257.03
	Network	25.26	37.91	51.19	98.69	108.02	101.33	111.37	98.71
	Combine	1.15	1.64	2.01	2.24	2.56	2.14	2.81	2.81

Table 7: Average Connection, Network, and Combining Time Spent for Fixed $l = 9$ in milliseconds

		Setup							
		(3,3)	(4,3)	(5,3)	(6,3)	(7,3)	(8,3)	(9,3)	(10,3)
1 Clients	Total	10.4	12.98	12.04	11.96	13.33	11.57	10.70	12.27
	Network	2.82	6.57	5.11	5.04	5.28	4.92	4.49	5.14
	Combine	1.76	1.46	1.66	1.51	1.90	1.49	1.44	0.56
10 Clients	Total	37.85	35.85	29.35	24.22	24.31	21.66	21.29	23.22
	Network	21.67	21.81	15.76	12.43	11.53	10.56	10.84	10.15
	Combine	2.05	2.02	1.97	1.69	1.71	1.57	1.51	1.07
100 Clients	Total	178.14	209.99	146.54	99.47	86.62	79.72	68.32	71.07
	Network	113.36	158.35	112.46	67.47	61.25	51.57	37.91	25.27
	Combine	2.14	2.49	2.00	1.85	1.90	1.65	1.64	1.62

Table 8: Average Connection, Network, and Combining Time Spent for Fixed $k = 3$ in milliseconds

A Additional Experiments

Tables 7 and 8 show the results for the micro benchmark.

B T-RSA Details

Key partitioning: The dealer creates two strong primes $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are also prime numbers. Next, it creates a random prime number $e > l$, and calculates $d = e^{-1}\ mod\ m$, where $m = p'q'$.

Then, the dealer creates a random polynomial $f(X) = \sum_{i=0}^{k-1} a_i X^i \in \mathbb{Z}[X]$, where $a_0 = d$, and $a_1, \ldots, a_{k-1}$ are random integers in $\mathbb{Z}$. Next, the dealer computes each party p_i's share as $s_i = f(i)\ mod\ m$. The public key is (n, e), while each party is given s_i as their share of the private key.

Using the secret key: For a given message $M \in \mathbb{Z}_n^*$, the chosen combiner selects a subset of the parties, $S = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, l\}$, where $|S| = k$, and sends M to each party in S. Each selected party p_{i_j} performs the following set of operations:

1. $\Delta = l!$
2. $\lambda^S_{0,i_j} = \Delta \frac{\prod_{i_x \in S \setminus \{i_j\}} -i_x}{\prod_{i_x \in S \setminus \{i_j\}} (i_j - i_x)}$
3. $w_{i_j} = M^{4\Delta s_{i_j} \lambda_{0,i_j}}$

λ^S_{0,i_j} is the polynomial interpolation constant for p_{i_j} in set S, where $\Delta f(0) \equiv \sum_{i_j \in S} \lambda^S_{0,i_j} f(i_j)\ mod\ m$. Once the combiner gets a partial result, w_{i_j}, from each party in S, it computes $w = \prod_{i_j \in S} w_{i_j}$. Then, it executes the extended Euclidean algorithm for e and $e' = 4\Delta^2$, and gets integers a and b, where $e'a + eb = gcd(e', e) = 1$. The greatest common divisor of e and e' is 1, since e is a prime number, and each factor of e' is smaller than e. Finally, the combiner computes $y = w^a M^b$ as the final result.

The final value y is in fact $M^d\ mod\ n$:

$$w \equiv \prod_{i_j \in S} w_{i_j} \equiv \prod_{i_j \in S} M^{4\Delta s_{i_j} \lambda_{0,i_j}} \equiv M^{4\Delta^2 d} \equiv M^{e'd}\ mod\ n$$

$$y \equiv w^a M^b \equiv M^{ae'd+b} \equiv M^{d(1-eb)+b} \equiv M^d\ mod\ n$$

Notes

[1] In a memoryless probability distribution, the cumulative probability depends on the distance from the starting time of the distribution to the current time.