Cloud Key Management using Trusted Execution Environment

Jaouhara Bouamama[1], Mustapha Hedabou[1] and Mohammed Erradi[2]
[1]*University Mohammed VI Polytechnic , Benguerir, Morocco*
[2]*ENSIAS, University Mohammed V, Rabat, Morocco*

Keywords: Cloud Computing, Key Management, Intel SGX.

Abstract: Cloud storage represents a primordial component in most information technology infrastructures. Using cloud instead of on-premise storage raises several security issues, especially when secret keys are stored on the cloud. In such a setting, a robust cloud key management system is a must. Using traditional key management systems (KMS) in the cloud suffers from performance and scalability limitations. This paper, proposes an efficient and secure cloud KMS based on Trusted Execution Environment, precisely Intel SGX. The suggested system (KMSGX), while being deployed on the cloud, is fully controlled by the end-user. Therefore, KMSGX allows running on-premise software key management securely on the cloud provider side, protecting the exchanged and stored data. The security properties of the suggested design have been formalized using the Applied Pi Calculus and proved with ProVerif. The experimental results have demonstrated the system's high performance in terms of the upload and download durations and the limited overhead compared to the plain design.

1 INTRODUCTION

Cloud computing consists of using networks of remote servers to store, manage and process data. The adoption of the cloud computing paradigm is expanding at an astonishing pace for both personal and professional usages. Users consider cloud computing mainly to profit from their high storage capacity and the available computational power. This is in addition to the cost reduction, efficiency, and flexibility. However, security represents a main cloud user requirement that must be considered while designing solutions based on the cloud.

Usually, to profit from the cloud services while ensuring data privacy, users store the data in their encrypted form. Using such data requires special handling when managing the secret keys allowing access to that data. Depending on the type of cloud service in use, most key management functions are partially or fully controlled by the cloud providers (Schneier, 2007). Accordingly, due to their sensitivity, cryptographic keys must be handled with care (Chokhani, 2013). For instance, they should be generated randomly, stored safely and exchanged via secure protocols.

Recently, several approaches were proposed to protect personal keys, ranging from software to hardware solutions (Rosen, 2012; Amazon, 2015; Phegade et al., 2017; Chakrabarti et al., 2017). These approaches raise many limitations in terms of performance, scalability, and on-demand self-service. On the other hand, Trusted Execution Environments (TEEs) were introduced to guarantee a high level of trust on remote platforms (Sabt et al., 2015). In a nutshell, TEEs were designed to build an isolated environment for secret code execution. They ensure the authenticity, integrity, and confidentiality of the executed code. In this context, existing popular TEE such as Intel SGX (Intel, 2017) relies on hardware to isolate sensitive code from untrusted access. Intel SGX operates over one CPU, which can have multiple secure environments called enclaves executed over untrusted software (Mukhtar et al., 2019).

In this paper, we design KMSGX, an efficient and secure cloud key management system based on Intel SGX. The system is configured and deployed by the end-user on the cloud platform. The system leverages trusted execution tools to store keys and perform sensitive computation on the cloud platform in a secure manner. KMSGX allows executing on-premise software key management securely in the cloud provider through accessible hardware facilities to keep control over sensitive keys at a low cost.

The proposed system ensures the authentication

Bouamama, J., Hedabou, M. and Erradi, M.
Cloud Key Management using Trusted Execution Environment.
DOI: 10.5220/0010558905600567
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 560-567
ISBN: 978-989-758-524-1

and the confidentiality properties following the Dolev Yao attacker model (Cervesato, 2001). The experimental results have demonstrated the system's efficiency in terms of the upload and the download duration, and the limited overhead compared to the plain design. To summarize, our contribution in this paper consists of:

- The design of KMSGX, a new cloud key management system based on a Trusted Execution Environment, aims to give complete control of the secret keys to the end-user while allowing basic cloud operations.
- The formal proof of the security properties of the designed system during different states of the secret keys; in transit, in use and at rest.
- The implementation and the evaluation of the suggested system.

The paper is organized as follows. Section 2 presents the TEE technology used in the proposed model. We discuss the existing related works in section 3. The overview of the system model and the threat model are given in section 4. In section 5, we describe the suggested key management system. Section 6 aims to formalize and prove the security properties. Section 7 is dedicated to the experimental results and a discussion. Finally, we conclude the paper in section 8.

2 INTEL SOFTWARE GUARD EXTENSION

Cryptographic operations have evolved from protecting the confidentiality of communication over public channels to more complicated tasks ranging from enabling access control to data, ensuring user authentication, to checking virtual machine integrity in cloud providers platforms. Accordingly, a key management system must be designed and implemented in securely since encryption is believed to be the only way to guarantee the confidentiality and authentication of outsourced data (Bentajer and Hedabou, 2020). In this direction, the key management process consists of multiple states, ranging from key generation, key storing, key processing, key activation/deactivation, key archival, and finally, key destruction (Chokhani, 2013). In this paper, we leverage the trusted execution environment functionalities to design KMSGX, a key management system based on Intel SGX.

Intel SGX is a technology that aims to enable high-level protection of secrets against all nonauthorised access, including operating systems and hypervisors. It has known a huge success by its involvement in designing multiple secure solutions (Kaaniche et al., 2020). It works by allocating hardware-protected memory where data and code reside. This protected memory is called an enclave. The enclave code can be invoked only via special instructions such as ECALL and OCALL. The ECALL instruction is a call to a predefined function inside the enclave from the application, whereas an enclave can invoke a predefined function in the application via an OCALL.

Additionally, Intel SGX provides dedicated attestation and sealing mechanisms to build a secure model. To this end, two hardware keys are generated. Typically, a *Report Key* to verify the signature of an enclave and a *Seal Key* to encrypt sensitive data outside the enclave (Intel, 2017).

Generally, a software attestation aims to convince a challenger that a software is trustworthy and running on the same platform. Accordingly, an enclave can prove its identity to either another enclave in the same platform or to a remote party (figure 1) .

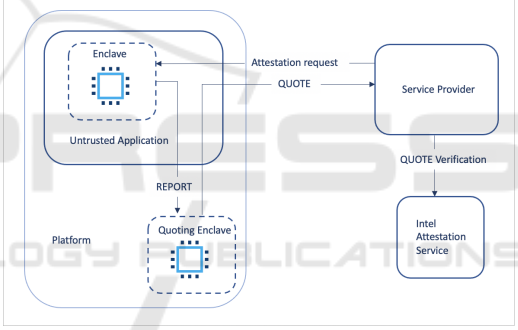

Figure 1: Intel SGX Attestation.

Attestation. The enclave starts by receiving the service provider's attestation request, and it generates a signed REPORT. This REPORT is sent to the quoting enclave, which is responsible for converting local REPORT to a QUOTE verifiable by the service provider. The quoting enclave generates a QUOTE by signing the REPORT with the private key of the enclave. Then, it forwards the QUOTE to the service provider. Finally, the service provider verifies the QUOTE via Intel Attestation Service, which is a public web service operated by Intel that proves the trustworthiness of QUOTES.

3 RELATED WORKS

Several research efforts have been dedicated to designing efficient cloud key management systems. Existing approaches might be categorized into homo-

morphic encryption and trusted environment-based approaches. In the following, we are presenting an overview of the most well-known approaches. In (Rosen, 2012), the author has presented Porticor, a Homomorphic Key Management Protocol. This protocol is based on the multiplicative homomorphic properties of El-Gamal scheme (ElGamal, 1985). The design depends mainly on the key split approach, which consists of splitting the key into two parts. The first one, a master key, which is under the control of the user. The second part is generated by the Porticor Virtual Appliance and stored within the Porticor Virtual Key management (PKVM) Service. The appliance uses both parts of the key to encrypt and decrypt data. The master key is homomorphically encrypted and never exposed in plain. Although, the master key is exposed to neither the Porticor Virtual Appliance nor the PVKM. The protocol relies on the assumption that an appliance and the PVKM do not collude. Moreover, as presented by their schema, the keys generated by the Porticor Virtual Appliance are stored as cleartext in the appliance memory, which runs in the cloud. If the memory is compromised, the keys and the encrypted data would be exposed as well.

The author of (Hamid, 2013) suggested the usage of Hardware Security Module (HSM) for key management in the cloud. HSM is used as the root of trust for various key management services where it safeguards and manages digital keys for strong security guarantees. Although HSM's isolated hardware, which is intrusion-resistant and tamper evident against physical attacks, was developed before the birth of the cloud computing paradigm. Thus, using HSM in cloud settings turns out to be costly and has inherent limitations related to a lack of elasticity and interoperability.

Fortinax (Phegade et al., 2017) is a trusted execution environment that uses Intel SGX enclaves for cryptographic protection from threats. The solution establishes a Self-Defending Key Management Service (SDKMS). It is considered the first commercial implementation of Intel SGX to create runtime encryption capsules. However, the paper lacks detailed description on how the protection and storage of the key is being handled.

OpenStack Barbican (Benjamin et al., 2017) is a platform developed by the Openstack community to design REST APIs that provide secure storage, provisioning and secrets management (i.e. passwords, encryption keys and X.509 Certificates). The Openstack Barbican key management service stores secrets in a dedicated HSM. Recently, an improvement of this work has been proposed by (Chakrabarti et al., 2017), which uses an intel SGX plugin that supports SGX attestation instead of using HSM. However, the suggested solution has only been evaluated within the OpenStack Barbican platform. It does not operate as a standalone solution where it can be used with other systems that require key management. In both approaches (Phegade et al., 2017; Chakrabarti et al., 2017), the key management system is not directly managed by the end-user.

4 SYSTEM MODEL

This section presents, the system model of the Cloud Key Management based on Intel SGX (KMSGX) and, the threat model. The system is configured and run on the cloud platform by the end-user. The proposed solution involves two entities: The conventional key management system (KMS) and a virtual appliance (VA). The KMS is run on the user's infrastructure perimeter. Whereas, the VA is uploaded in the cloud platform to serve as a bridge to link the KMS to the encryption application running on the cloud platform.

System Model. The proposed cloud key management delegates the control of the key management steps to the key management system on-premise. This goes from the key generation to the key deletion. The virtual appliance is leveraged to perform computation on stored data in cloud platform after decrypting it.

The main components of the proposed design are the key management service on-premise and a virtual appliance running inside the cloud and data storage. The appliance is composed of untrusted/trusted applications, where a secret code is executed inside an SGX enclave. The protocol is divided into three main phases: Setup, Upload, and Operation request, as presented in figure 2.

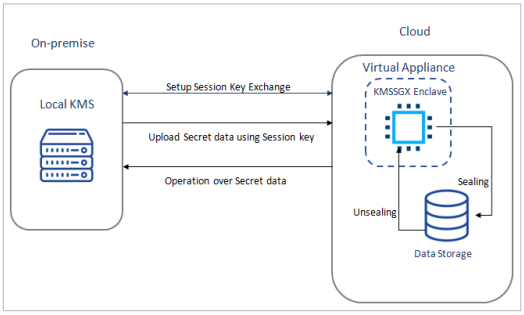

Figure 2: System Model.

During the *Setup* phase, the KMS starts an appliance that initializes an SGX enclave. Then, the enclave es-

tablishes a secure channel with the KMS on-premise. For the *Upload* phase, the KMS upload a data using the exchanged session key. Where, the KMSGX's enclave decrypts the data using the same session key. Finally, the enclave seals the data using the unique Seal Key. Regarding the *Operation request* phase, the KMS sends an encrypted request to perform an operation on an uploaded data *(i.e. download)*. The Enclave decrypts the request, extracts the requested data and unseal it using the Seal Key. Later, the enclave performs the requested operation and encrypts the results to be sent back to the KMS.

Threat Model. The communication between the different entities is not assumed to be secure. An attacker may have complete control over the communication channels. So, the adversary's goal is to intercept the user's secret key to learn its sensitive data.

Moreover, the threat model is based on the Dolev Yao attacker model (Cervesato, 2001), where an arbitrary adversary is powerful enough to intercept/modify the communication. We consider that an attacker cannot perform any hardware attacks over the SGX appliance in this scope. These attacks were covered in previous works (Van et al., 2018; Kocher et al., 2019; Lipp et al., 2018).

On the other hand, the attacker has high privileges within the cloud side, where all system software considered malicious (i.e. operating system, hypervisor, etc.). In the presence of an attacker given this threat model, different security proprieties must be ensured.

Secrecy. The KMS secret data requires to remain confidential, by securing the encryption key. Thus, they must be processed inside a secure environment and stored in encrypted memory. With the presence of high privileged attacker in the VA, the secret key cannot be learned. The appliance should guarantee the confidentiality of the sensitive data.

Authentication. The KMS sends sensitive data only to trusted appliances, where the identity of the software is verified before any exchange. An attacker cannot impersonate the appliance and claims to be a genuine software.

5 KMSGX: CLOUD KEY MANAGEMENT SYSTEM

In this section the proposed solution is described in more detail. The Key Management System challenges the remote cloud through the KMSGX appliance to prove its trustworthiness using the remote attestation provided by Intel SGX. . Furthermore, a secure channel is established via an authenticated Diffie-Hellman key exchange (Diffie and Hellman, 1976; Krawczyk, 2003). Accordingly, the interactions between the Key Management System, the Virtual Appliance, and the SGX enclave are described in figure 3.

5.1 Setup

Practically, the session establishment is done through sigma protocol (Ludlum, 2002). It is a three-phase protocol based on a Diffie-Hellman key exchange authenticated with digital signatures.

Firstly, the KMS sends an attestation request to the VA. The request is in the form of a nonce (a random number generated for this exchange) and a public key. The KMS on-premise chooses a generating g in a finite cyclic group G, and it picks a private random number a to generate the public key g^a. Then, the VA performs a call to the enclave to initialize the remote attestation taking the KMS request as an input. The KMSGX enclave, in its turn, generates a Diffie-Hellman public key (g^b). A QUOTE is created and sent back to the VA. The QUOTE includes information about the public key and signed with the enclave's identity. The VA forwards the QUOTE back to the KMS on-premise. Finally, the KMS extracts the Diffie-Hellman public key embedded in the QUOTE after a successful verification with the IAS.

5.2 Upload

After establishing a secure channel through the setup phase, a shared key is exchanged. In the meantime, a given user sends a file to the KMS on-premise to be stored in the cloud.

After receiving the file, the KMS generates an Encryption Key used to encrypt the file. This Encryption Key is encrypted using the Session Key. Next, the KMS sends the encrypted file and the encrypted Key to the appliance. The VA invokes a call to the trusted code, taking as an input the received data from the remote KMS. Inside the enclave, the encryption key is decrypted using the session, key and it is re-encrypted using the enclave's Seal Key. The Seal Key is considered to be a hardware key known only to the actual enclave. Later on, the enclave sends back the results to the VA using an OCALL. Finally, the application stores the encrypted data in data storage.

5.3 Operation Request

After receiving an operation request from the given user, the KMS encrypts it using the session key. Then,

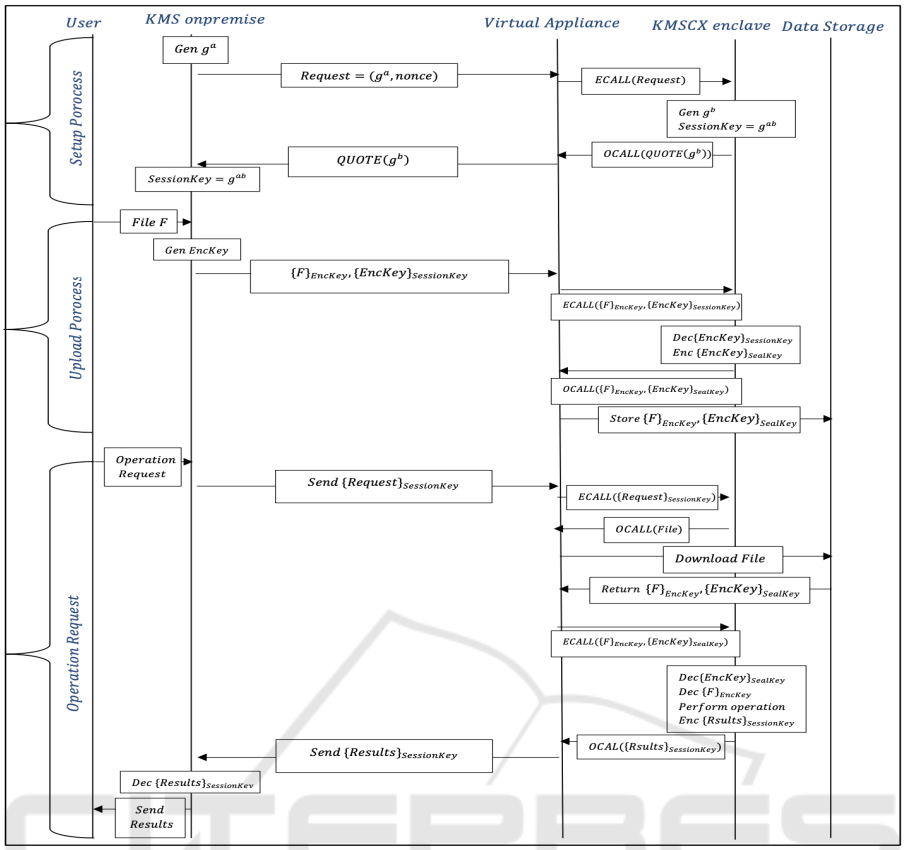

Figure 3: KMSGX Design.

it sends the encrypted request to the VA, which invokes a call to the trusted code, taking the request as an input. Inside the enclave, the operation request is decrypted using the session key. Next, the enclave invokes an OCALL to download the specific file included in the request. The requested file is extracted from the data storage. Then, it is sent to the KMSGX enclave. Afterward, the enclave decrypts the encryption key using the Seal Key. And, using this encryption key, the file can be decrypted. At the end, the enclave performs the requested operation and sends back the encrypted results through an OCALL. Finally, the VA forwards the encrypted results to the remote KMS, which decrypts the results and sends them to the specific user.

6 FORMALIZATION & PROOF

In this section, we present a formal verification of the KMSGX model, using the applied pi-calculus, an extension of the pi-calculus used with cryptographic functions defined by an equational theory, (Abadi et al., 2017; Cortier and Kremer, 2014). Explicitly, the applied pi-calculus can study and analyze security protocols.

The protocol is presented with the applied pi-calculus syntax semantic. Then, the analysis phase is automated using the Proverif software. This tool which is a tool that verifies symbolic protocols.

6.1 Semantic of the Model

The syntax of the calculus includes an infinite set of names, an infinite set of variables, and a signature Σ consisting of a finite set of function symbols associated with an arity. In this work, the function symbols consist of the cryptographic primitives used by the protocols. The suggested model is presented in the following grammar:

The symbol functions are described as follows:

6.2 Formal Model

As described in the system model the protocol is composed of three phases: Setup, Upload and operation request. We simply refer to download as a particular case of operation request, and this, does not impact

$M,N,Q,S,U ::=$	Terms
sk, ek	Variables (seal/encryption key)
$SEnc(M,ek)$	Symmetric Encryption
$SDec(M,ek)$	Symmetric Decryption
$Quote(M)$	Quote Generation
$CheckQuote(Q)$	Quote Verification
$Seal(S,sk)$	Sealing
$Unseal(U,sk)$	Unsealing
$fst(<M,N>)$	First Projection
$snd(<M,N>)$	Second Projection

$SDec(SEnc(M,ek),ek) = M$	Encryption
$Quote(M) = Q$	Quote Generation
$CheckQuote(Q) = M$	Quote Verification
$Unseal(Seal(S,sk),sk) = S$	Sealing
$fst(<M,N>) = M$	First Projection
$snd(<M,N> = N$	Second Projection

the proof. The main processes in this design are the Key Management System, the SGX appliance, and the storage server processes. Therefore, we have two communication channels :
C_{enc} : Channel between the KMS and the SGX appliance. This channel is used to transfer secret data.
C_{seal} : Channel between the SGX appliance and the storage server. This channel is used to seal secret data.
$C_i : \overline{C_i}$: Input/output in a channel i.
Note that, we formally present each process within each phase. These processes capture the communication exchanges and do not abstract the hardware initialization and specification.
The aim is to prove that the KMS will send his sensitive data only to a trustworthy SGX appliance after a successful session key exchange. Then, we need to ensure that data transited is secret and protected in a Dolev Yao attack model (Cervesato, 2001) .

6.3 Verification

The security properties of the system were verified using Proverif. It is a tool for automatically analyzing the security properties of different cryptographic protocols (Blanchet, 2005).
Authentication. The authentication property is captured using correspondence assertion. This latest is used to capture relationships between events which can be expressed in the form: *"if an event e has been executed, then event e' has been previously executed"* (Blanchet, 2005).

The proposed scheme makes sure that if the KMS is sending his sensitive data to the SGX appliance, the identity of the appliance is verified. Hence, we present the following events:
event KMSattestSGX(pkey): The KMS attests the appliance SGX after verifying a report using the supplied public key.
event SGXstartattestKMS(pkey): The SGX appliance starts an attestation process to generate a report verifiable by its public key.

As the KMS is willing to send secret data only to reliable appliance, we define the following correspondence:

```
query: pkey; event(KMSattestSGX(x)) ==>
event(SGXsttartattestKMS(x)).
```

The correspondence proves the authentication of the SGX appliance. If the KMS receives a report from the SGX appliance, then the SGX appliance should have sent it before.
Confidentiality. The aim of this part is to prove that the exchanged secret data remains secure while it is in transit via the secure channel and while it is in use inside the enclave and while it is at rest within the cloud storage. In fact, the specification of the secrecy inside a trusted environment was already presented in several works (Sinha et al., 2015; Subramanyan

SETUP PROCESSES

Key Management System	
νa	new private value
$\overline{C_{enc}} < g^a >$	output Pubkey
$C_{enc} < x_E >$	wait Quote
$let\ y = CheckQuote(x_E)\ in$	get enclave Pubkey
$let\ K_{se} = y^a\ in$	session key
Virtual Appliance	
νb	new private value
$C_{enc} < x_S >$	wait Pubkey
$\overline{C_{enc}} < Quote(g^b) >$	output Quote
$let\ K_{se} = x_S^b\ in$	session key

UPLOAD PROCESSES

Key Management System	
$\nu M, K_{enc}$	new data, ENCkey
$let\ M_{enc} = SEnc(M,K_{enc})\ in$	data encryption
$\overline{C_{enc}} < M_{enc}, SEnc(K_{enc},K_{se})) >$	output ENCdata
Virtual Appliance	
$C_{enc} < x_S >$	wait secret data
$let\ K_{dec} = DEnc(snd(x_S),K_{se})$	decrypt ENCkey
$\overline{C_{seal}} < fst(x_S), Seal(K_{dec},K_{seal})) >$	seal in storage
Storage Server	
$C_{seal} < x_E >$	wait data

DOWNLOAD PROCESSES

Key Management System	
$C_{enc} < x_E >$	wait data
$let\ M_{dec} = DEnc(x_E,K_{se})$	decrypt data
Virtual Appliance	
$C_{seal} < x_{seal} >$	wait stored data
$let\ K_{unseal} = Unseal(snd(x_{seal}),K_{seal})$	unseal secret key
$let\ M_{dec} = DEnc(fst(x_{seal}),K_{unseal})$	decrypt data
$\overline{C_{enc}} < SEnc(M_{dec},K_{se}) >$	output ENCdata
Storage Server	
$\overline{C_{seal}} < M,K >$	output stored data

et al., 2017). Therefore, we assume that the underlying hardware is secure. Also, we assume that the cryptographic primitives are perfect, and an attacker can decrypt or verify signature only when he has the required keys. To test the secrecy of the secret *S* in the model, the following query is included in the Proverif input file:

```
query attacker(S)
```

The query asks for the secrecy of the exchanged secret in the presence of an attacker.

After modeling the proposed protocol in the applied pi-calculus and performing verification using Proverif, the security properties hold for both secrecy and authentication. A summary of the verification is presented in figures 4 and 5.

```
-- Query not attacker(s[]) in process 0.
nounif mess(c[],pkX_1)/-5000
Completing...
Starting query not attacker(s[])
RESULT not attacker(s[]) is true.
```

Figure 4: Confidentiality test.

```
-- Query event(KMSattestSGX(x_1)) ==> event(SGXstartattestKMS(x_1)) in process 0.
nounif mess(c[],pkX_1)/-5000
Completing...
Starting query event(KMSattestSGX(x_1)) ==> event(SGXstartattestKMS(x_1))
RESULT event(KMSattestSGX(x_1)) ==> event(SGXstartattestKMS(x_1)) is true.
```

Figure 5: Authentication test.

7 EXPERIMENTAL RESULTS

We evaluate KMSGX on Intel i5-8259U processors. Due to the lack of SGX support in current cloud platforms, we simulate the protocol using a virtual appliance hosted in a Ubuntu 18.04 LTS 64bit virtual machine. We run the SGX driver, SDK, and platform software version 2.9.1.

To illustrate the key management on-premise, we used the PyKMIP version 0.10.0. It is a python implementation of the Key Management Interoperability Protocol. Thus, the KMIP server is responsible for generating, deleting, and encrypting data on behalf of the user in the implemented model. The tests are performed over files of different sizes: 10 MB, 20 MB, 50 MB, 100 MB, 200 MB, and 500 MB.

We measured the performance of the proposed protocol within the three phases and compared it with the plain design where no additional cryptography and SGX primitives are added (encryption/decryption, sealing/ unsealing). The distribution of the time performance is as follows:

a) Setup : The secure channel creation duration using the sigma protocol within the remote attestation included in the Intel SGX.

b) Upload : The duration of the secret data transfer through the secure channel, in addition to the execution time inside the enclave.

c) Download: The time of the encrypted data transfer after unsealing from external storage.

The experimental results are grouped in tables 1 and 2.

Table 1: Upload Performance.

File Size (Mb)	Setup (ms)	Upload (ms)	KMSGX Design (ms)	Plain Design (ms)	Overhead (%)
10	10,0980	20,12	30,218	20,57	0,4690
20	10,2200	36,13	46,35	40,33	0,1492
50	10,3500	88,13	98,48	93,899	0,04878
100	10,0200	174,05	184,07	182,902	0,0063
200	9,0600	224,95	234,01	233,002	0,0043
500	10,6700	499,95	510,62	510,547	0,00014

Table 2: Download Performance.

File Size (Mb)	Setup (ms)	Download (ms)	KMSGX Design (ms)	Plain Design (ms)	Overhead (%)
10	10,0980	14,99	25,088	19,12	0,31213
20	10,2200	31,31	41,53	36,23	0,14628
50	10,3500	64,03	74,38	71,97	0,03348
100	10,0200	131,49	141,51	140,6619	0,00496
200	9,0600	211,25	220,31	219,598	0,00324
500	10,6700	533,22	543,89	543,829	0,00011

We note that for both phases (Upload/Download), the running time of the setup phase is added. The analysis shows that the added security properties such as Diffie-Hellman key exchange in the Setup phase, Symmetric encryption, Sealing, and Unsealing in both the upload and download phases did not heavily penalize the performance of the plain design. We can notice from figures 6, 7 that the overhead cost does not exceed 0,5 % of the total cost of the plain design. Therefore, the overhead cost of both upload and download is negligible when the file size is increased.

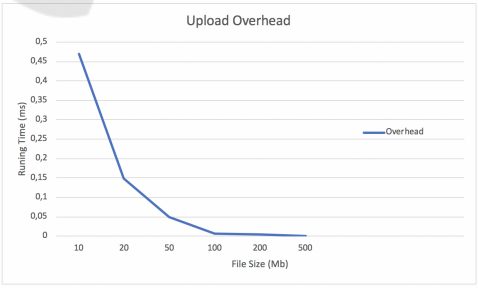

Figure 6: Upload Overhead.

8 CONCLUSION

We presented in this work a Cloud Key Management based on a Trusted Execution Environment, namely Intel SGX. Throughout the model, we show the pos-

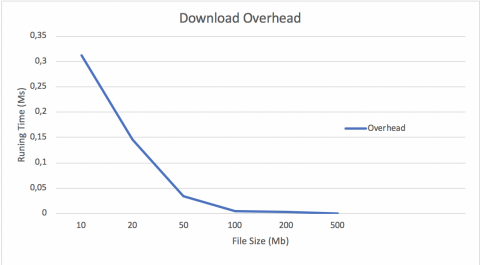

Figure 7: Download Overhead.

sibility of achieving a secure key management fully controlled by the end cloud customer. Accordingly, transitive trust is established between the key Management System on-premise and the SGX appliance hosted in the cloud. Hence, the suggested KMSGX allows executing on-premise software key management system securely in cloud providers side through accessible hardware facilities to control sensitive keys at low cost directly by the end-user.

We strengthen this work by formally proving the security properties of the protocol using the Applied Pi Calculus. Thus, we formally proved that the suggested scheme provides authentication of the SGX appliance, and the secret keys confidentiality of the secret keys. As a future direction, we plan to extend this work by considering a distributed setting where multiple nodes or clouds are involved. The aim is to design a system that operates in an untrusted multi-cloud environment.

REFERENCES

Abadi, M., Blanchet, B., and Fournet, C. (2017). The applied pi calculus: Mobile values, new names, and secure communication. *Journal of the ACM (JACM)*, 65(1):1–41.

Amazon (2015). Amazon, cloudhsm.

Benjamin, B., Coffman, et al. (2017). Data protection in openstack. In *2017 IEEE 10th International Conference on Cloud Computing*, pages 560–567. IEEE.

Bentajer, A. and Hedabou, M. (2020). Cryptographic key management issues in cloud computing. *Advances in Engineering Research*, 34:78–112.

Blanchet, B. (2005). Proverif automatic cryptographic protocol verifier user manual. *CNRS, Departement dInformatique, Ecole Normale Superieure, Paris*.

Cervesato, I. (2001). The dolev-yao intruder is the most powerful attacker. In *16th Annual Symposium on Logic in Computer Science—LICS*, volume 1.

Chakrabarti, S., Baker, B., and Vij, M. (2017). Intel sgx enabled key manager service with openstack barbican. *arXiv preprint arXiv:1712.07694*.

Chokhani, R. C. M. I. S. (2013). Cryptographic key management issues & challenges in cloud services. *National Institute of standards and Technology*.

Cortier, V. and Kremer, S. (2014). Formal models and techniques for analyzing security protocols: A tutorial. *Foundations and Trends in Programming Languages*, 1(3):117.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654.

ElGamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472.

Hamid, L. (2013). Cloud-based hardware security modules. US Patent App. 13/826,353.

Intel (2017). Intel software guard extensions.

Kaaniche, N., Belguith, et al. (2020). Prov-trust: Towards a trustworthy sgx-based data provenance system. In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications, SECRYPT*, pages 225–237. INSTICC.

Kocher, P., Horn, et al. (2019). Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE.

Krawczyk, H. (2003). Sigma: The 'sign-and-mac'approach to authenticated diffie-hellman and its use in the ike protocols. In *Annual International Cryptology Conference*, pages 400–425. Springer.

Lipp, M., Schwarz, et al. (2018). Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, pages 973–990.

Ludlum, R. (2002). *The Sigma Protocol*. Macmillan.

Mukhtar, M. A., Bhatti, et al. (2019). Architectures for security: A comparative analysis of hardware security features in intel sgx and arm trustzone. In *2nd International Conference on Communication, Computing and Digital systems (C-CODE)*, pages 299–304. IEEE.

Phegade, V., Schrater, et al. (2017). Self-defending key management service with intel® software guard extensions.

Rosen, A. (2012). Analysis of the porticor homomorphic key management protocol. *Porticor Cloud Security*.

Sabt, M., Achemlal, et al. (2015). Trusted execution environment: what it is, and what it is not. In *IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE.

Schneier, B. (2007). *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons.

Sinha, R., Rajamani, et al. (2015). Moat: Verifying confidentiality of enclave programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1169–1184.

Subramanyan, P., Sinha, et al. (2017). A formal foundation for secure remote execution of enclaves. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450.

Van, Bulck, J., Minkin, et al. (2018). Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, pages 991–1008.