EnclavePDP: A General Framework to Verify Data Integrity in Cloud Using Intel SGX

Yun He[1,2], Yihua Xu[3], Xiaoqi Jia[1,2*], Shengzhi Zhang[3], Peng Liu[4], and Shuai Chang[1,2]

[1]{CAS-KLONAT[†], BKLONSPT[‡]}, Institute of Information Engineering, Chinese Academy of Sciences
[2]School of Cyber Security, University of Chinese Academy of Sciences
[3]Metropolitan College, Boston University
[4]Pennsylvania State University

Abstract

As the cloud storage service becomes pervasive, verifying the integrity of their outsourced data on cloud remotely turns out to be challenging for users. Existing Provable Data Possession (PDP) schemes mostly resort to a Third Party Auditor (TPA) to verify the integrity on behalf of users, thus reducing their communication and computation burden. However, such schemes demand fully trusted TPA, that is, placing TPA in the Trusted Computing Base (TCB), which is not always a reasonable assumption. In this paper, we propose EnclavePDP, a secure and general data integrity verification framework that relies on Intel SGX to establish the TCB for PDP schemes, thus eliminating the TPA from the TCB. EnclavePDP supports both new and existing PDP schemes by integrating core functionalities of cryptography libraries into Intel SGX. We choose 10 existing representative PDP schemes, and port them into EnclavePDP with reasonable effort. By deploying EnclavePDP in a real-world cloud storage platform and running the 10 PDP schemes respectively, we demonstrate that EnclavePDP can eliminate the dependence on TPA and introduce reasonable performance overhead.

1 Introduction

Nowadays, many organizations demand to keep their data records, and then perform deep analysis over the data using machine learning or other techniques for their business purposes. For instance, e-health companies offer optimized health care plan for customers by analyzing customers' health records. However, not all organizations are able to build and manage their private data storage platform due to the high cost of building and maintaining such a platform. Hence, cloud storage service has become quite popular, due to the features like pay-as-you-go, elasticity, cost-saving, maintenance, etc. There are many popular cloud storage services today, such as Dropbox, Google Drive, Amazon S3, One Drive, etc.

*Corresponding author: jiaxiaoqi@iie.ac.cn
†Key Laboratory of Network Assessment Technology, CAS
‡Beijing Key Laboratory of Network Security and Protection Technology

However, users will lose control of their data stored on the cloud platform, which is an inherent issue in such data outsourcing model. Although the service providers can be bound by a Service Level Agreement (SLA) to ensure the data integrity, users still cannot fully trust them. On one hand, the cloud servers are not immune to data loss or corruption even the cloud providers are faithful to protect the outsourced data. For instance, Dropbox, Amazon and Tencent Cloud lost data due to improper operations, inadvertent administration errors or system bugs [1–3]. Although these incidents occurred unintentionally, the service providers may not immediately inform the data loss incidents to users to protect their reputation (i.e., service providers are "imperfect and selfish" [4]). For example, according to [5], healthcare data breaches are identified after 84.78 days and customers are notified after additional 68.31 days on average. On the other hand, a service provider may be actively malicious: deleting data that is infrequently accessed to save storage space but still charging users for the deleted data [6], or keeping fewer replicas violating the SLA.

In recent years, numerous data integrity verification approaches [7–19] have been proposed to ensure the integrity of the outsourced data. These approaches are referred to as Provable Data Possession (PDP) schemes. Such PDP schemes provide probabilistic guarantees that the outsourced data has not been maliciously tampered with, without accessing the entire data from the cloud storage server. PDP schemes (e.g., APDP [9], etc.) usually generate metadata (or tag) using the original data, and upload the metadata together with the original data to the cloud storage servers. The proof of data integrity is generated by cloud storage servers using this metadata and verified by the data owner. Other PDP schemes [14, 17–19] were proposed to support public auditing for multiple users via a Third Party Auditor (TPA). However, the TPA may steal users' private data (honest but curious TPA) or even conduct collusion attacks with the cloud service provider (inherently malicious TPA). Besides the trustworthiness concern, deploying TPA also involves extra cost.

In this paper, we propose **EnclavePDP** (Enclave-protected Provable Data Possession), a practical and general framework to verify the integrity of the outsourced data on cloud platforms relying on the Trusted Execution Environments (TEEs), i.e., Intel SGX [20], thus eliminating TPAs and reducing both the computation and communication burdens from users. We implemented a prototype of EnclavePDP using Intel SGX and ported the core functionalities of Intel SGX SSL crypto library [21], Intel SGX GMP library [22], and the PBC [23] (Pairing-Based Cryptography) library into EnclavePDP. Then, 10 representative PDP schemes, i.e., APDP [9], CPOR [12], SEPDP [10], MRPDP [11], PPPAS [19], DHT-PA [18], SEPAP [17], DPDP [7], FlexDPDP [8], and a basic Message Authentication Code (MAC) based scheme (MAC-PDP), are implemented on EnclavePDP with reasonable effort. We evaluated EnclavePDP on a real-world cloud storage service, FastDFS [24]. Experimental results show that EnclavePDP introduced negligible overhead to the response time per PDP request for all the 10 PDP schemes on different file sizes (1GB and 16KB), varying from 1.0% to 24.5%.

We summarize the main contribution of the paper as below:

- We proposed and implemented EnclavePDP, a novel and generic framework that can securely verify the integrity of the outsourced data relying on Intel SGX, thus eliminating the dependency on TPAs. The core functionaries of various cryptographic libraries are tailored and ported into Intel SGX to support both the existing and new PDP schemes, and 10 representative PDP schemes are implemented in EnclavePDP with reasonable effort.
- We performed a comprehensive evaluation of EnclavePDP by deploying it on a real cloud storage service, FastDFS. All the 10 PDP schemes are evaluated in EnclavePDP in the aspects of code base in Intel SGX and overhead of response time, thus eliminating the performance concerns of running PDP schemes in Intel SGX.

2 Background

2.1 Provable Data Possession in Clouds

To verify the integrity of the outsourced data on cloud platforms, lots of PDP schemes [7–16] were proposed. Generically, PDP schemes consist of two phases: a setup phase and a verification phase. In the setup phase, the client (or the data owner) generates keys (private or public, depending on the scheme), as well as metadata (or tag) using the keys and the original data. The metadata (or tag) and the original data are uploaded to the cloud storage server. In the verification phase, the client constructs a challenge that contains a random subset of file blocks, and sends the challenge to the prover (i.e., the cloud storage server). The prover uses the challenge, the metadata (or tag) and the file blocks to compute a proof of data possession and then sends it back to the client. The client uses the proof to verify that the data on the cloud is still intact. In addition, many literature surveys, e.g., [25–29] made a comprehensive comparison among existing PDP schemes. Below we choose four aspects: types of data, retrievability, encryption and auditing, to discuss the existing PDP schemes.

Types of data: There exist two types of data: static data and dynamic data. Static data (e.g., data archive, backups) is never modified but appended only, whereas dynamic data is frequently changed due to operations like update, write and delete. Some PDP schemes, e.g., APDP [9], are only suitable for static data, because they need re-generate tags of the complete file whenever new data is inserted. In contrast, other schemes like SEPDP [10] support dynamic data operations.

Retrievability: Generally, PDP schemes only provide probabilistic guarantees of the data integrity, i.e., identifying data corruption without data recovery, e.g., [8–11, 30], etc. In contrast, POR (Proof of Retrievability) schemes provide the guarantee that the data is intact and still retrievable even after corrupted by using the redundant encoding, e.g., CPOR [12], Mirror [16], Iris [31], etc.

Encryption: Some PDP schemes utilize symmetric key encryption to achieve scalability/efficiency, e.g., SEPDP [10] uses symmetric key encryption and cryptographic hash functions, while others use asymmetric key encryption for better security, e.g., APDP [9] uses RSA-based homomorphic verifiable tags (HVT) as the metadata.

Auditing: PDP schemes either support private auditing or public auditing. For the former, the verifier is always the data owner, e.g., APDP [9], SEPDP [10], FlexDPDP [8], etc. For the latter, the TPA sends challenges and verifies proofs on behalf of the data owner to reduce the computation and communication overhead of the data owner. Public auditing schemes can be further categorized into privacy preserving (e.g., PPPAS [19], DHT-PA [18]) and non-privacy preserving (e.g., PoS [32], SEPAP [17], MHT-PA [6]) schemes. It is worth noting most of the public auditing schemes are implemented using the BLS [33] signature cryptographic primitive.

2.2 Intel SGX

Intel SGX [20] creates an isolated code execution environment, which enables applications to maintain data confidentiality and integrity. Even the privileged software (OS, hypervisor and BIOS) cannot violate the protection of Intel SGX. Note that we do not consider side-channel attacks against SGX, which can be addressed orthogonally by corresponding countermeasures (e.g., [34]).

Enclave. Intel SGX constructs trusted execution environments referred to as *enclaves* and creates an encrypted memory region called Enclave Page Cache (EPC) for enclaves to store code and data. SGX uses a hardware Memory Encryption Engine (MEE) [35] to encrypt/decrypt the enclave data, and also provides a hardware access control mechanism to prevent illegal access to the enclave memory. An Intel SGX application generally contains two parts: secure code (enclave) and non-secure code (non-enclave or application). The application needs to launch the enclave, and uses `ecall`/`ocall` interfaces to switch control between the enclave and the non-enclave. Since privileged operations (e.g., system calls) cannot be executed inside enclaves, `ocall` is invoked to execute those privileged operations indirectly.

SGX Remote Attestation. Intel SGX remote attestation [36] is to ensure that the enclave is correctly initialized on a remote SGX enabled platform. It evaluates the enclave identity, its structure, the integrity of the code inside the enclave. Furthermore, remote attestation can provide shared secret between the enclave application and its owner to setup a secure communication channel over the untrusted network.

Sealing. Enclaves can write confidential data to persistent storage securely using *sealing* [36], a mechanism to encrypt and authenticate the enclave data. Each enclave is provided with a sealing key, derived from an enclave identity (either Enclave Identity or Signing Identity), private to the executing platform and the enclave. Data sealed against Enclave Identity (MRENCLAVE) can only be decrypted by the same enclave, whereas data sealed against Signing Identify (MRSIGNER) can be unsealed by any enclave signed by the same developer.

3 Overview

3.1 System and Threat Model

We consider a cloud storage scenario where usually three primary entities exist: *clients or users*, *Cloud Storage Service (CSS)*, and *TPA*. Specially, clients have a large amount of data to be stored on the cloud, and CSS is managed by the Cloud Service Provider (CSP) to provide data storage service (typically with a large amount of storage space and computational resources). To save the computational resources as well as the online burden potentially incurred by the periodic data integrity verification, clients resort to TPA (with extra capabilities that clients do not have, e.g., keeping always online) to verify the integrity of their outsourced data on cloud.

We assume the threats to the integrity of users' outsourced data on cloud can be both internal and external on the cloud storage platform, e.g., software bugs, hardware failures, malicious or accidental management errors, revenue-motivated hackers, etc. Moreover, the cloud storage platform may intendedly hide the data corruption incidents from users to maintain its reputation. Most prior works, e.g., MHT-PA [6], SEPAP [17], usually rely on TPA to provide a cost-effective way for users to verify the integrity of their outsourced data, with the assumption that TPA is reliable and trustworthy. However, such assumption is not always valid, since TPA could be (1) honest but curious, learning the users' data after the audit as described in PPPAS [19], and (2) untrusted, conducting collusion attack with the untrusted cloud service providers. Hence, the proposed solution in this paper does not rely on TPA. We assume the remote CPU (with Intel SGX security features) running on the cloud storage platform is trusted. We also assume that the adversary cannot extract secrets within the CPU packages, which implies that we trust CPUs to protect code and data hosted inside TEEs. Side-channel and denial-of-service attacks are outside the scope of this paper.

3.2 Motivation of Using Intel SGX

In this paper, we mainly focus on those PDP schemes that rely on TPA to verify the integrity of the outsourced data, since users' computation resources as well as online burden can be significantly reduced by TPA. However, such PDP schemes are still limited in the following aspects. **(P1) The honest but curious TPA**. Generally, TPA needs to be fully trustworthy, exactly following the PDP schemes to execute the core verification functionality. However, an honest but curious TPA may potentially learn users' data through the procedures of challenging and verifying [19] it gets involved in. **(P2) Collusion attack with untrusted cloud providers**. Although a few privacy-preserving public auditing schemes, e.g., PPPAS [19] and DHT-PA [18], can be used to address the data breach issues, they are still limited in eliminating the collusion attack when the TPA collaborates with the cloud storage server to deceive users. **(P3) Communication overhead**. The communication overhead (sending challenges and proofs between the cloud storage server and the verifier running in the TPA) is not negligible.

Fortunately, Intel SGX provides the trusted execution environment, enclave, which can be leveraged to solve the above problems. First, Intel SGX prevents the underlying untrusted OS or hypervisor from accessing the code/data inside the enclave. Hence, the PDP schemes can run faithfully in enclave on untrusted platforms (i.e., the cloud storage servers), thus eliminating the dependency on TPA (solving P1 and P2). Second, the enclave can also protect the private keys used by the PDP schemes against leaking to untrusted components, and it can also protect the integrity of verification against malicious modification. Therefore, we can also deploy private auditing PDP schemes inside the enclave, which reduces the computation overhead of the data owners and provides public-auditing-like support. Finally, EnclavePDP can be deployed on any of the cloud servers (as long as the underlying Intel CPU supports SGX), thus it can be co-located with the cloud storage services on the same physical machine. Hence, the communication overhead between the verifier in EnclavePDP and the cloud storage services (inter-process communication) is negligible (solving P3), compared with that of the native PDP schemes (network communication).

3.3 Possible Concerns of Using Intel SGX

Compatibility. The implementation of PDP schemes depends on some cryptography libraries (e.g., OpenSSL, PBC [23], etc.). Therefore, these libraries must be ported into Intel SGX before implementing PDP schemes inside the enclave. Currently, two libraries have been ported into enclave: (i)

Intel SGX SSL [21] library to support OpenSSL and (ii) Intel GMP [22] library to support the GNU multiple precision arithmetic (GMP) library. However, the PBC [23] library demanded by many BLS-based PDP schemes (e.g., PPPAS [19], DHT-PA [18], SEPAP [17], etc.) has not been ported into enclave yet. We provided a lightweight enclave-supported PBC library by trimming and porting the native PBC library into the enclave (Section 5.1).

Memory usage. In the current implementation of Intel SGX, the EPC that can be used by the enclave is limited to 128 MB, and only 93 MB is usable for applications. When enclave uses memory beyond the EPC size limit, SGX swaps some EPC pages to unprotected DRAM, which incurs high performance overhead. Therefore, when developing applications running in enclave, reducing the memory footprint is crucial. Especially when implementing PDP schemes, necessary support, i.e., cryptography libraries, needs to be ported into enclave as well, which further increases the overall memory consumption. We addressed this problem by trimming the unnecessary modules in cryptography libraries (Section 5.1) and running only the critical code of PDP schemes in enclave (Section 4.1). We also quantitatively measured the code base in enclave in our evaluation (Section 6.2), which demonstrates reasonable memory footprint.

Runtime performance. When the non-enclave code needs to execute a trusted function (running inside the enclave), it invokes the SGX `ecall` primitive to switch the execution flow into the enclave. Such enclave transitions (i.e., switching control between enclave and non-enclave) will impose high runtime overhead [37]. Therefore, the number of `ocalls/ecalls` needs to be carefully managed to avoid severe performance overhead. In Section 5 and Section 6, we discussed and evaluated the impact on practicality of EnclavePDP introduced by the enclave transitions overhead.

4 The Approach of EnclavePDP

4.1 The Architecture of EnclavePDP

Generally, there are five major functionalities that should be fulfilled by PDP schemes: *KeyGen*, *Tag*, *Challenge*, *Proof* and *Verify*. Brief description of each operation is as follows.

- *KeyGen*. During the initialization phase, the client generates public and private keys for the other functionalities.
- *Tag*. The file is divided into `n` blocks, and a tag is generated for each block using the private key[1].
- *Challenge*. After choosing a random set of file blocks to audit, a challenge is generated using the private key.
- *Proof*. When receiving a challenge, a proof of data possession is computed using the public key.

[1]The original file and all the tags will be uploaded to the remote cloud storage servers.

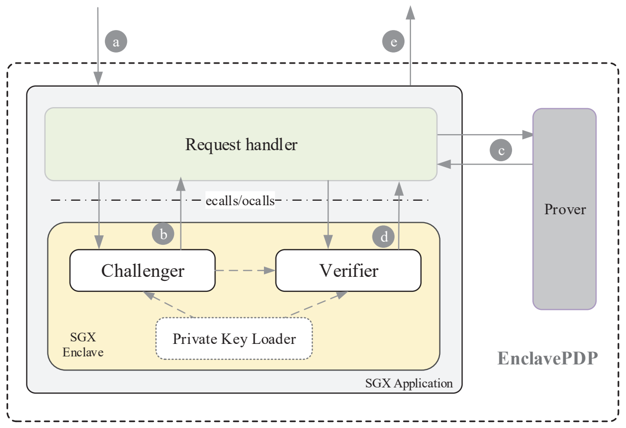

Figure 1: The Architecture of EnclavePDP

- *Verify*. A received proof will be verified against the challenge using the private key or public key depending on the specific PDP schemes.

KeyGen and *Tag* are running on the client side, so the integrity of these two functionalities depends on the client. *Proof* is conducted by the prover, i.e., the cloud storage server. For existing PDP schemes, *Challenge* and *Verify* functionalities are performed by either the client or the TPA. To ensure the correctness of the data integrity verification, the private key used to generate challenges and verify proofs should never be exposed to the cloud platforms. In addition, challenge and verification results must be protected against being tampered with or forged. Therefore, these security-sensitive operations (i.e., *Challenge* and *Verify*) should be executed in the enclave and the security-sensitive data (e.g., the private key) should be placed in the enclave as well.

Figure 1 shows the architecture of EnclavePDP. The *Challenger* running inside the enclave generates challenges for PDP schemes. EnclavePDP also makes a backup of the challenge to defeat potential rollback attacks[2]. The *Verifier* inside the enclave is responsible for verifying proofs generated by the *Prover*, a regular (non-SGX) application running on the cloud storage server, entitled to access the data to be audited to generate proofs of data possession on behalf of the cloud storage services. The *Private Key Loader* is designed to securely load private keys into the enclave (see Section 4.3.2). *Request handler* is a SGX application (as described in Section 2.2), which is responsible for receiving/sending messages (e.g., verification request, challenge, proof, etc.) from/to the other modules. It invokes `ecalls` provided by the *Challenger* or the *Verifier* to generate challenges or verify proofs.

4.2 The Workflow of EnclavePDP

As shown in Figure 2, the workflow of EnclavePDP mainly consists of two phases: an **initialization** phase and a **verification** phase.

[2]For example, an untrusted cloud storage server may provide a fake proof based on an outdated challenge as the response to the current challenge.

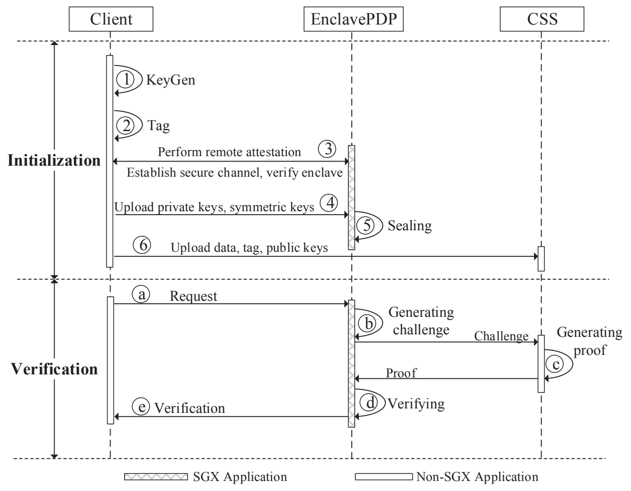

Figure 2: The workflow of EnclavePDP

EnclavePDP Initialization. In the initialization phase, clients (data owners) cooperate with EnclavePDP to complete the necessary setup:

(1) *Key Generation*: Several keys will be generated by the client, i.e., a pair of public key (`pub_k`) and private key (`pri_k`), as well as a symmetric key (`sk`). The public key is available to the cloud storage services and will be used to generate proofs, while the private key, owned by the client and EnclavePDP, is used to generate tags as well as challenges, and verify proofs. EnclavePDP uses the symmetric key to encrypt the verification result and sends it back to the client.

(2) *Data Tagging*: The client generates tags for the original data using the private key. For some PDP schemes (e.g., DHT-PA [18]), the client also generates additional data structures (e.g., Dynamic Hash Table) to record extra information, e.g., data version, timestamp, etc., to support dynamic auditing.

(3) *Remote Attestation*: The client can upload the EnclavePDP executable to the cloud server running on Intel SGX, and start the EnclavePDP remotely. After starting EnclavePDP, the client attests it using Intel SGX remote attestation to verify the code integrity of EnclavePDP, and subsequently performs ECDH [38] protocol to create a secure communication channel for the following operations.

(4) *Secrets Uploading*: The private key and the symmetric key will be sent to EnclavePDP via the secure channel established in Step (3). Other security-sensitive data (e.g., dynamic hash table of DHT-PA schemes [18]) used to support dynamic auditing will also be uploaded to EnclavePDP.

(5) *Sealing*: When EnclavePDP receives the private key and the symmetric key, it encrypts them using Intel SGX Sealing technique and stores the sealed (i.e., encrypted) keys on disk. Other security-sensitive data, (e.g., dynamic hash table of DHT-PA [18]), should be sealed in the same way.

(6) *Data Uploading*: The client uploads the public key, the data and all the tags to the cloud storage services (CSS).

EnclavePDP Verification. In the verification phase, the client issues a PDP request (e.g., via HTTP) to the cloud storage service, which forwards the request (e.g., via TCP sockets) to the *Request handler* (step ⓐ in Figure 1). Note that the client can also directly issue PDP requests to EnclavePDP. The request then will be forwarded to the *Challenger* via ecall interface, and the *Challenger* generates a challenge (step ⓑ in Figure 1) for the file indicated in the request (typically via file name or file path). The *Request handler* will transmit the challenge to the *Prover*, a regular application (i.e., non-SGX application) that can access the data stored in the cloud storage servers. Note that the cloud storage services can entitle the *Prover* to access data directly or provide APIs for the *Prover* to access data indirectly. The *Prover* reads the data from the cloud storage servers, and uses such data (i.e., file blocks, tags) and the challenge to generate a proof of data possession. The proof will be sent back to the *Request handler* (step ⓒ in Figure 1) and then forwarded to the *Verifier* (step ⓓ in Figure 1). The *Verifier* uses the proof to verify whether the cloud storage servers actually possess the correct data, and returns a verification response encrypted using the symmetric key `sk` to the *Request handler*. Finally, the client will receive (step ⓔ in Figure 1) and decrypt the verification response using the same `sk` to ensure the confidentiality and integrity of it. Note that the *Request handler* is also designed to be able to generate verification requests periodically on behalf of the client, and forward these requests to the *Challenger*, followed by other steps mentioned as above. Finally, EnclavePDP will create a tamper-free (encrypted by the enclave) verification log that will be forwarded to the client if necessary.

4.3 Key Management

4.3.1 Private Key Protecting

The private key used by the PDP schemes can never be exposed to the cloud storage services. In the initialization phase, the client establishes a secure channel to upload the private key and EnclavePDP encrypts the private key using SGX Sealing technique. The confidentiality and integrity of the keys are guaranteed by two conditions: (i) the ECDH protocol is executed in the enclave, which guarantees the confidentiality and integrity of the ECDH computation; (ii) The sealed private key is bound to a signing authority (developer), so only the enclave signed by the same authority can unseal it.

4.3.2 Private Key Loading

Before performing challenge or verification operations, the *request handler* (running outside of the enclave) firstly reads the sealed private key from the disk and invokes `ecalls` provided by *Private Key Loader* to unseal the private key inside the enclave. Note that the symmetric key `sk` is also loaded and unsealed in the enclave. When generating challenge (or verification), the *Challenger* (or the *Verifier*) uses the unsealed private key to generate challenges (or verify the proof). To reduce enclave transitions caused by `ecalls`, the private keys used recently are stored in the private key buffer. Thus, the

Table 1: A brief comparison of the ten PDP schemes

	Dynamic	Retrievability	Public	Encryption
MACPDP	X	PDP	X	Sym.
APDP [9]	X	PDP	X	Asym.
MRPDP [11]	X	PDP	X	Asym.
SEPDP [10]	X	PDP	X	Sym.
CPOR [12]	X	**POR**	X	Sym.
DPDP [40]	✓	PDP	X	Asym.
FlexDPDP [8]	✓	PDP	X	Asym.
PPPAS [19]	X	PDP	✓	Asym.
SEPAP [17]	✓	PDP	✓	Asym.
DHT-PA [18]	✓	PDP	✓	Asym.

* Note: "✓" means "support"; "X" means "not support"; "Sym." means symmetric encryption; "Asym." means asymmetric encryption.

Private Key Loader will firstly check if the required private key already exists in the private key buffer. If so, it returns the private key directly. Otherwise, it will demand the *request handler* to load the sealed private key from the disk and unseal the private key into the private key buffer. To save the enclave memory, the LRU (Least Recently Used) strategy is utilized to refresh the private key buffer.

5 Implementation

We implemented a prototype of EnclavePDP on a Linux platform, based on Intel SGX SDK 2.4, Intel SGX Driver 1.0, Intel SGX SSL library integrated with OpenSSL 1.1.0i, Intel SGX GMP library and an enclave-supported PBC library trimmed based on pbc-0.5.14. For generality and scalability, the *Request handler* utilizes Linux epoll [39] mechanism to provide support for multi-thread execution and concurrent responses. The requests to verify data integrity are encapsulated into TCP sockets and forwarded to EnclavePDP, which eases the deployment of EnclavePDP on third party cloud services.

5.1 Porting PDP Schemes

PDP Implementation. We chose 10 representative PDP schemes, which cover the taxonomy described in Section 2.1 as in Table 1. Most of the PDP schemes can be implemented in Intel SGX quite straightforward, but the following issues need to be addressed for other PDP schemes.

(1) For MAC-PDP, to avoid frequent I/O operations from the enclave and reduce the EPC memory consumption. the MAC of the file blocks to be verified is not re-computed inside the enclave. Instead, the prover (running on the cloud storage server) re-computes the MAC and also loads the encrypted tags (i.e., MACs of file blocks encrypted by the private key and uploaded to the cloud storage server during the initialization phase) associated with these file blocks into non-EPC memory. Then the verifier (inside the EnclavePDP) decrypts the tags to get the original MAC and compares it with the MAC computed by the prover.

(2) Some PDP schemes (e.g., SEPAP and DHT-PA) design an extra data structure to record the data property information (e.g., timestamp, version) used to perform dynamic auditing [3]. Such additional data structures should be uploaded to the TPA (when involved), protecting their integrity from the cloud storage server. In contrast, EnclavePDP encrypts these data structures using the private key and upload them to the remote cloud server during the initialization phase. In the verification phase, EnclavePDP decrypts them in the enclave and uses them to verify the proofs.

(3) During the initialization phase, DPDP generates root metadata based on the Rank-Based Authenticated Skiplist (RBASL) to verify its integrity, while FlexDPDP generates root metadata based on FlexList to verify its integrity. Similar as (2), EnclavePDP also encrypts the root metadta and uploads it to the cloud storage server, and then decrypts it inside the enclave in the verification phase.

Trimming Intel SGX SSL library. Intel SGX SSL [21] is to provide cryptographic service for enclave applications based on OpenSSL library. It includes lots of functionalities that are unnecessary to implement PDP schemes, e.g., *des*, *rc2* and *md4*, etc. To save the enclave memory consumption, we trimmed the native implementation of SGX SSL by removing those unnecessary modules from the configuration file at the compilation time. Finally the size of the trimmed SGX SSL library decreases 26.1% (from 4.6MB to 3.4MB).

Porting PBC library. Public auditing schemes (e.g., PPPAS [19], SEPAP [17] and DHT-PA [18]) are all based on the BLS signature cryptographic primitive implemented in the PBC library [23], which is not supported by Intel SGX yet. Therefore, we ported the PBC library into SGX to make it easy to port other existing or develop new BLS-based schemes in EnclavePDP. We only ported those functions required by the public auditing schemes into SGX to provide a lightweight PBC library, thus reducing the memory consumption of EnclavePDP. Note that some of those functions need a bit tuning. For instance, generating random numbers is a quite frequent operation for most PDP schemes, the PBC library generates random numbers using the */dev/urandom* pseudo file on Linux platform. However, code running in the enclave cannot perform I/O operations directly. Hence, we use Intel RDRAND instruction [41] when porting the random number generation function in the PBC library.

5.2 Protecting Enclave Binary Integrity

The implementation of the PDP schemes inside the enclave is essentially an executable binary running on the untrusted cloud platform. Hence, the adversaries may reverse-engineer the binary enclave shared object to extract the code logic. We utilized Intel SGX PCL technique [42] to encrypt the enclave shared object (.so) at build time and decrypt it at enclave load time. Moreover, the untrusted cloud providers may create a fake enclave to perform ECDH [38] protocol with the data

[3]In particular, when generating tags for the original data, SEPAP will create a doubly linked info table (DLIT), while DHT-PA scheme will create a dynamic hash table (DHT).

owner to steal the private keys. To defeat such threat, the data owner periodically requests enclave to return its enclave measurement (constructed by invoking the *EREPORT* instruction, which can only be executed inside the enclave), and compares it with local backup measurement. The successive operations can only be continued upon a match of the measurements. Note that malicious cloud providers may create a copy of EnclavePDP and execute this copy, but they cannot reveal any secret data inside the enclave. The copy of EnclavePDP may cause DoS attack, which is out of scope of this work.

5.3 Integration with Cloud Storage Service

In order to deploy EnclavePDP on existing cloud storage services easily, we exposed high-level interfaces (e.g., TCP sockets) for users or cloud storage services to submit/return PDP requests/responses. [4]

We deployed the prototype of EnclavePDP on FastDFS [24], an open source high performance distributed file system (DFS). FastDFS has two major functionalities: tracker and storage. The former conducts scheduling and load balancing for file access. The latter performs file management including: file storing, file syncing, providing file access interface. We extended the *fastdfs-nginx-module* of FastDFS for user to easily submit integrity verification requests, e.g., issuing a *http get* request. When receiving requests submitted by users, the *fastdfs-nginx-module* forwards the requests to EnclavePDP (runs as a daemon on the storage servers) and waits for the verification result returned by EnclavePDP. The implementation of integrating EnclavePDP with FastDFS is less than *300 lines of C code*. Note that the *Prover* runs on the storage server of FastDFS, so it can access the outsourced data directly and generate proofs on behalf of FastDFS. As for the closed source cloud storage services (e.g., Amazon S3), EnclavePDP can only invoke the public APIs exposed by those cloud storage services to access the outsourced data. Current implementation of EnclavePDP supports the integrity check of the data stored on Amazon S3 using AWS C++ SDK, with around *70 lines of C++ code* added into EnclavePDP and without any changes to Amazon S3 platform. However, EnclavePDP needs to download all the data to local disk and performs verification, because Amazon S3 does not support random access to different data blocks.

[4]The cloud storage service needs to: (1) allow users to submit PDP requests and forward the PDP requests to EnclavePDP; (2) allow the *Prover* process to access the outsourced data directly, or provide APIs for the *Prover* to access the data indirectly. Recall that the *Prover* is a non-SGX application designed to generate proofs on behalf of the cloud storage services, which makes it possible to integrate EnclavePDP with existing cloud storage services with as few changes as possible.

6 Evaluation

6.1 Experimental Setup

We deployed EnclavePDP and FastDFS on Microsoft Azure Confidential Computing (ACC) [43] VMs supporting Intel SGX. Each VM runs Ubuntu 16.04.1 LTS with kernel version 4.15.0-1036 on a platform with an Intel(R) Xeon(R) E-2176G CPU (4 cores, 3.70 GHz, and 12 MB cache) and 16 GB RAM. We ran FastDFS v5.12 on four VMs, one VM as the tracker server and the others as storage servers. The tracker server takes charge of scheduling and load balancing for file access, and is also extended to dispatch PDP requests to other storage servers. In particular, FastDFS utilizes its Nginx module (i.e., *fastdfs-nginx-module* that is built on nginx-1.15.4) to interact with the user, thus we extend this module to handle the PDP requests submitted by the user. EnclavePDP runs as a daemon on the storage servers. When the tracker server receives PDP request, it dispatches the PDP request to the EnclavePDP running on the corresponding storage servers. To evaluate the throughout of EnclavePDP when handling concurrent requests, we used a popular workload testing tool, Apache JMeter, to simultaneously issue integrity verification requests to EnclavePDP at different speed (requests/second). Apache JMeter runs on a local computer with Ubuntu 16.04.1 LTS equipped with Intel(R) Core(TM) i7-7700HQ CPU.

6.2 Analysis of TCB

We measured the change of the TCB code base after porting the *Challenge* and *Verify* operations into Intel SGX, as shown in Table 2. We only focus on the core part of the implementation of those PDP schemes when measuring the SLOC (Source line of code), and ignore other code like I/O operations, sockets, etc. All the PDP schemes include *Challenge* and *Verify* operations, which are two security-sensitive functions. To guarantee the confidentiality of private keys used to generate challenges or verify proofs, loading private keys into enclave is the third security-sensitive function. For DPDP and FlexDPDP, there is an extra verification against the integrity of the Rank-based Authenticated SkipList and FlexList respectively. Therefore, there exists the fourth security-sensitive function for those two schemes. Accordingly, each security-sensitive function is associated with an `ecall` interface. Hence, each PDP request will conduct three or four `ecall` crossings (i.e., traps into enclave) depending on the specific schemes.

As in Table 2, the security-sensitive SLOC of native PDP varies from 7% to 33%, while the security-sensitive SLOC after porting them into enclave varies from 8% to 36%. Take APDP as an example. Its native implementation totally contains 1348 SLOC, among which 300 SLOC is security-sensitive (account for 22% of the total). After porting it into enclave, the security-sensitive SLOC increases to 350 SLOC

Table 2: TCB size of EnclavePDP

Schemes	SLOC	Security -sensitive SLOC	Security -sensitive functions	SGX -enabled SLOC
MACPDP	1483	115 (7%)	3	121 (8%)
APDP [9]	1348	300 (22%)	3	350 (25%)
MRPDP [11]	1440	476 (33%)	3	624 (43%)
SEPDP [10]	1259	106 (8%)	3	153 (12%)
CPOR [12]	1057	167 (15%)	3	210 (19%)
DPDP [7]	950	117 (12%)	4	145 (15%)
FlexDPDP [8]	945	139 (14%)	4	158 (16%)
PPPAS [19]	1012	199 (20%)	3	249 (24%)
SEPAP [17]	620	162 (26%)	3	225 (36%)
DHT-PA [18]	720	187 (26%)	3	255 (35%)

(25% of the total). Such increase mainly results from extra functionalities, such as private key loading, challenges backup/destroy, decrypting other security-sensitive data (e.g., doubly linked info table, dynamic hash table), etc. Additionally, we also quantitatively measured the SLOC of those three enclave-supported libraries: Intel SGX SSL library contains about 138.4K SLOC; Intel SGX GMP contains 163.4K SLOC; PBC library contains 29.9K SLOC.

6.3 Evaluation of Challenger and Verifier

Given the amount of data outsourced on the cloud, it is inadvisable to challenge all data blocks at once to verify the integrity. Instead, the sampling verification is used by most PDP schemes, that is, to achieve high-accuracy verification by only checking a portion of the data at once. In particular, [9, 11, 18, 19] demonstrated that if t fraction of data is corrupted, randomly sampling c blocks will detect such corruption with the probability $P = 1-(1-t)^c$. When $t = 1\%$, the verifier only needs to verify 460 randomly chosen blocks to detect such corruption with the probability larger than 99%. Hence, in all the following experiments, we choose 460 as the maximum number of challenge blocks[5], even for large files with much more file blocks. When the number of the total file blocks is less than 460, the verifier challenges all the file blocks instead. We measured the performance of performing the *Challenge* and *Verify* operations inside the enclave, and compared it with the native implementation below. Note that the time involved in sending challenges/proofs and reading data is ignored for both of the two cases.

6.3.1 Overhead of Challenge Operation

Figure 3 depicts the time (in μs) of generating challenges for both enclave-enabled and native implementation with varying file sizes. For all the 10 PDP schemes, both enclave and native implementation demonstrate similar changes over different file sizes. The challenge operation time of APDP, MRPDP and SEPDP is relatively constant regardless of file size, because their challenge operations just produce a random seed used to generate the random block set to be verified, which is independent of the file size. For the other seven PDP schemes, as the file size increases, the challenge operation time first increases and then becomes constant. This is because those schemes generate a random `n`-element set for the challenge, whose size increases as the file size increases. It reaches a constant (i.e., the maximum number of challenge blocks) when the number of file blocks exceeds the maximum number of challenge blocks (460 as described above).

Comparing with the native PDP schemes, APDP and MRPDP saw an increase of 18.2% and 18.1% of the challenge operation time respectively. MAC-PDP, DPDP, FlexDPDP and CPOR incurred 62%, 50%, 41.5% and 180% overhead when their challenge operation time reaches a constant. The three BLS-based schemes, i.e., PPPAS, DHT-PA, SEPAP, imposed similar overhead, 89.7%, 85.3% and 84.3% respectively. The challenge operation time of SEPDP increased nearly 1.9 times. Actually the difference of overhead results from the challenge operation time of each PDP scheme. For instance, the challenge operation for native SEPDP is below 4 μs for varying file sizes, which magnifies the impact of `ecall` overhead, thus causing nearly 1.9 times overhead. In contrast, the challenge operation for native APDP is from 250 μs to 300 μs for varying file sizes, thus causing merely 18.2% overhead.

Observation 1. The overhead of the challenge operation is not proportional to the security-sensitive SLOC. PDP schemes in the same category introduce similar overhead. Enclave-enabled challenge operation time is still in the scale of microsecond (μs), which should have little impact on practical applications.

6.3.2 Overhead of Verify Operation

Figure 4 depicts the time of executing the verify operation for both enclave-enabled and native implementation with varying file sizes. The verify operation time of native PDP schemes varies significantly. In particular, the verify operation time of SEPDP, MAC-PDP and CPOR is in the scale of microsecond (μs), but in the scale of millisecond (ms) for APDP, MRPDP, DPDP and FlexDPDP (FDPDP). For the other three BLS-based schemes (PPPAS, DHT-PA and SEPAP), their verify operation time is in the scale of second (s).

Observation 2. RSA-based schemes (ms) are an order of magnitude slower than symmetric-based schemes (μs), because the RSA-based modular exponential operations are complicated and expensive. BLS-based schemes (s) is another order of magnitude slower, probably due to the inherent drawback of the complicated and slow computation of BLS signatures (e.g., curves pairing) [44].

Regarding enclave-enabled implementation of PDP schemes, executing the verify operation inside the enclave imposed 17.1%, 12.7% and 24.7% overhead for APDP, MRPDP

[5]The block size is 16KB for APDP, and 4KB for other PDP schemes.

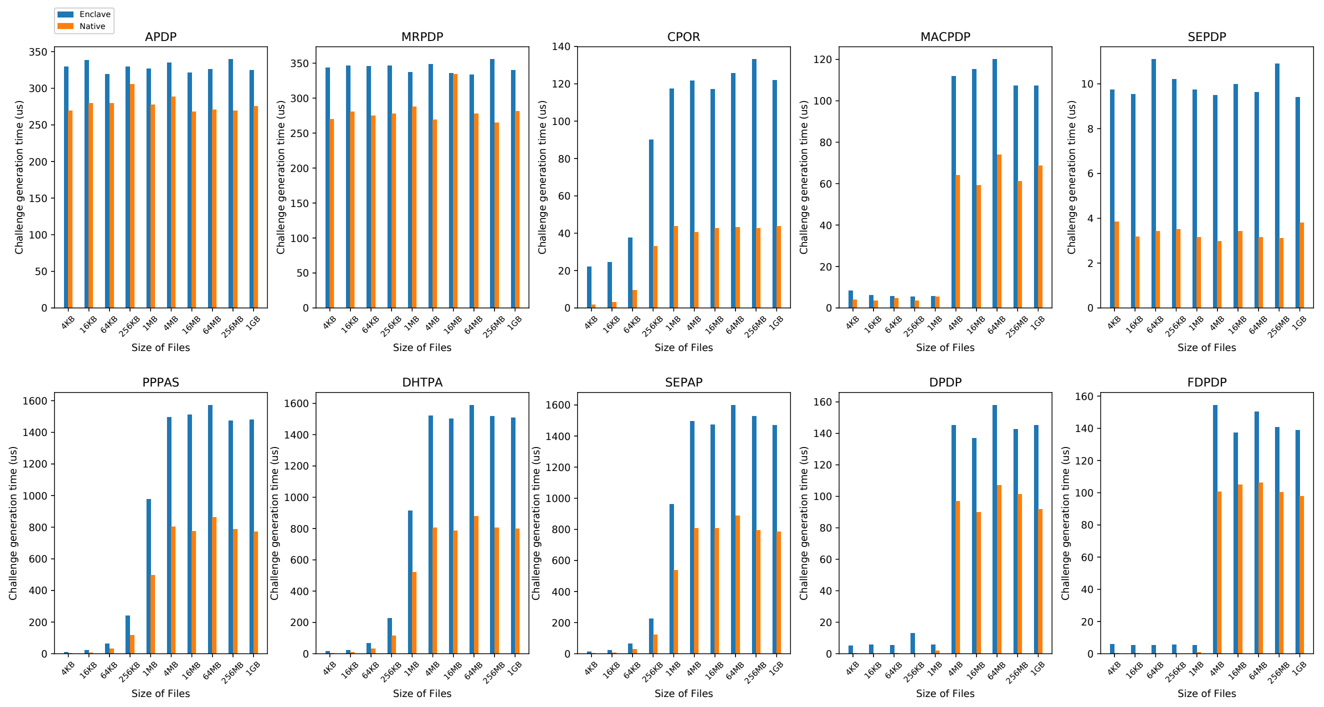

Figure 3: Overhead of Challenge Operations

and MAC-PDP, respectively. The three BLS-based schemes (PPPAS, DHT-PA and SEPAP) saw similar performance degradation, i.e., 34.9% for PPPAS, 36.5% for DHT-PA, and 35.2% for SEPAP, respectively. DPDP and FlexDPDP introduced 47% and 37% overhead respectively, while SEPDP and CPOR experienced 82.0% and 92.2% increase of the verify operation time respectively. The reason for such overhead is similar to that of the overhead of the challenge time described above, the low-overhead operation (the verify operation) is affected more significantly by the `ecall` execution context switch. Though up to 92% runtime overhead, the following experiments (Section 6.4) will demonstrate that such microsecond-range or millisecond-range overhead makes acceptable, or even negligible impact on the throughput for the practical deployment.

Observation 3. Running the verification operation inside the enclave introduces less overhead (12.70%–92.2%) compared with the challenge operation (18.10%–190%), because the challenge operation is relatively "lightweight" compared with the verification operation in terms of computation.

6.4 Evaluation of PDP Request

We measured the response time and throughput of the 10 native PDP schemes and their EnclavePDP implementation, by verifying the integrity of files with different sizes. The response time includes the time of network communication and all operations (i.e., *Challenge*, *Proof*, *Verify*) in the verification phase. Since we set the maximum number of challenge blocks as 460, we intend to choose 1GB file (larger than 460 blocks) and 16KB file (smaller than 460 blocks) to conduct the following experiments.

The right of Table 3 shows the average response time under the condition of the maximum throughout of both native PDP and EnclavePDP on verifying the integrity of 1GB file. "Thr" indicates the number of concurrent threads (imitating multiple users) used to trigger the maximum throughput. As shown in Table 3, the average response time for most of the PDP schemes (including CPOR, SEPDP, MACPDP, APDP, MRPDP, DPDP and FlexDPDP), when implemented in Intel SGX, is almost negligible, with the overhead from 1.0% to 5.4%. In contrast, the overhead of the three BLS-based schemes, i.e., PPPAS, DHT-PA and SEPAP, is 24.5%, 23.4%, and 10.9% respectively. Recall the verification operation for the BLS-based schemes takes significantly longer than that of other PDP schemes, in the scale of second, but the overhead incurred by EnclavePDP is still reasonable.

We conducted an experiment to measure the proportion of challenge/verify time to the total response time, when launching only one thread to issue one PDP request each time. As shown in Table 4, the verification time of the BLS-based schemes accounts for much more proportion than that of other schemes, which well explains why running BLS-based schemes in enclave introduces more overhead compared with other PDP schemes. However, the response time also includes the network communication latency and the time of the proof operation, thus the overhead per PDP request for these PDP schemes is diluted. For most of these 10 PDP schemes, the runtime of the challenge operation and the verify operation accounts for a quite small proportion of the total response time, which is in line with the fact that although per challenge

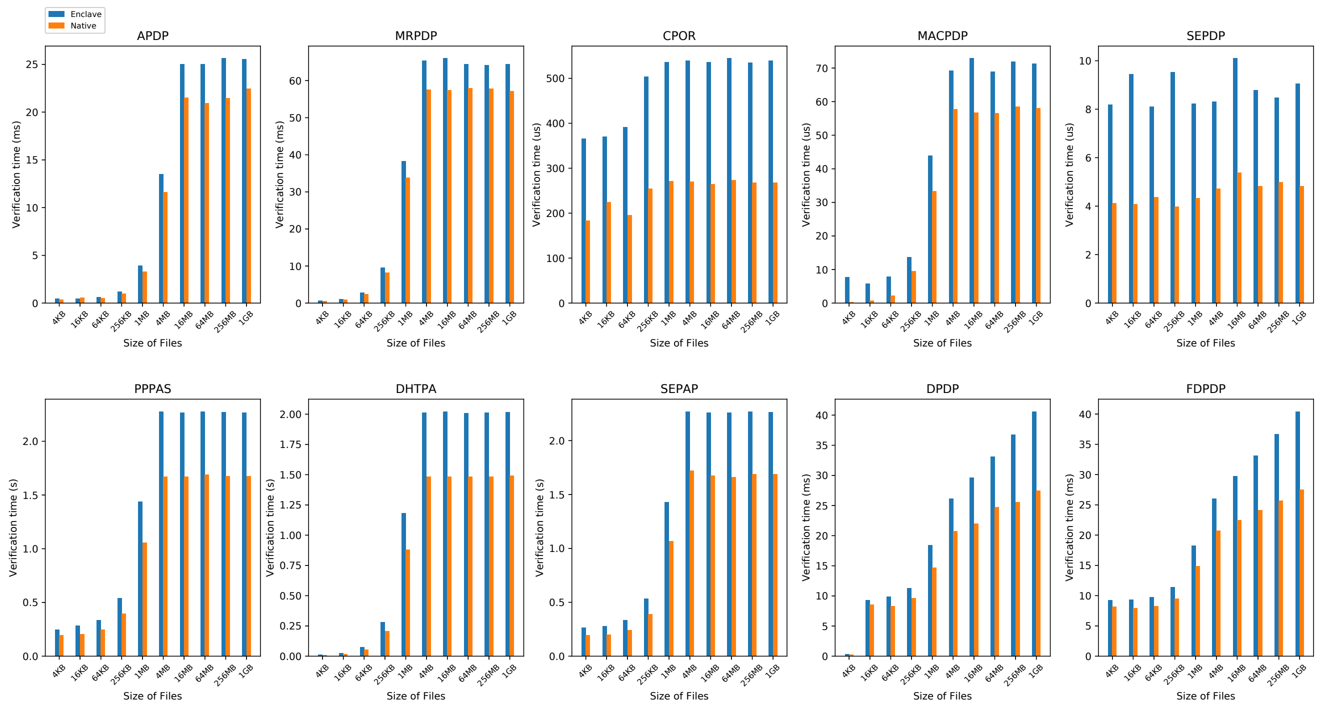

Figure 4: Overhead of Verify Operations

Table 3: Evaluation of PDP Request on 16KB and 1GB File.

Schemes	16KB file				1GB file			
	EnclavePDP	Native PDP	Overhead	Thr.	EnclavePDP	Native PDP	Overhead	Thr.
MACPDP	515 ms (403.2 req/s)	500 ms (418.1 req/s)	3.0% (3.6%)	220	987 ms (239.9 req/s)	971 ms (244.7 req/s)	1.6% (2.0%)	200
APDP [9]	1154 ms (33.8 req/s)	1110 ms (35.1 req/s)	3.9% (3.7%)	40	2164 ms (8.9 req/s)	2079 ms (9.3 req/s)	4.1% (4.5%)	40
MRPDP [11]	957 ms (89.8 req/s)	936 ms (91.1 req/s)	2.2% (1.4%)	100	3115 ms (6.2 req/s)	2976 ms (6.5 req/s)	4.7% (4.6%)	40
SEPDP [10]	642 ms (410.2 req/s)	601 ms (436.7 req/s)	6.8% (6.0%)	275	725 ms (327.6 req/s)	718 ms (334.0 req/s)	1.0% (1.9%)	250
CPOR [12]	539 ms (389.7 req/s)	520 ms (414.2 req/s)	3.6% (6.0%)	250	1140 ms (84.8 req/s)	1131 ms (85.7 req/s)	1.0% (1.1%)	100
DPDP [7]	1095 ms (90.4 req/s)	939 ms (103.4 req/s)	16.6% (12.6%)	120	24655 ms (0.0405 req/s)	23814 ms (0.0418 req/s)	3.4% (3.5%)	5
FlexDPDP [8]	1075 ms (90.7 req/s)	934 ms (104.7 req/s)	15.1% (13.3%)	120	50698 ms (0.052 req/s)	48100 ms (0.0552 req/s)	5.4% (5.5%)	5
PPPAS [19]	8391 ms (3.3 req/s)	6318 ms (4.5 req/s)	32.8% (24.4%)	30	46034 ms (0.363 req/s)	36886 ms (0.465 req/s)	24.5% (21.9%)	20
SEPAP [17]	5552 ms (3.5 req/s)	4162 ms (4.6 req/s)	33.3% (24.0%)	30	41700 ms (0.365 req/s)	37591 ms (0.458 req/s)	10.9% (20.4%)	20
DHT-PA [18]	1311 ms (22.3 req/s)	1110 ms (26.3 req/s)	18.1% (15.2%)	30	34207 ms (0.487 req/s)	27709 ms (0.64 req/s)	23.4% (24.0%)	30

* Note: the value in the "( )" is the maximum throughput (req/s) associated with corresponding response time.
* Thr. : Threads indicating concurrent users.

or verify operation introduces relatively high overhead, the impact to per PDP request is almost negligible.

In addition, we find that the proportion of challenge/verify time for most enclave-enabled PDP schemes is in the same order of magnitude as that of native PDP schemes, slightly higher than the latter. For DPDP and FlexDPDP schemes, the higher overhead on challenge time (2.3 times and 1.5 times respectively) might be explained by a loop function (an expensive operation) used to generate non-negative random integers in the challenge generation function of the two enclave-enabled schemes. The 1.6 times overhead on challenge time for the enclave-enabled CPOR scheme is probably due to the extra operations (e.g., private keys loading, challenge backup), which has significant impact on the originally small challenge operation time of the CPOR scheme.

Observation 4. The impact incurred by EnclavePDP to the entire response time, a complete challenge-verify procedure, is acceptable for practical deployment.

From the perspective of maximum throughput, SEPDP, MAC-PDP and CPOR perform much better than the other schemes. In particular, the maximum throughput of SEPDP and MAC-PDP is one order of magnitude higher than CPOR and two orders of magnitude higher than APDP and MRPDP. This can be attributed to the fact that symmetric encryption (SEPDP and MAC-PDP) is of higher efficiency than asymmetric encryption (e.g., APDP). Meanwhile, the maximum throughput of those three BLS-based schemes (i.e., PPPAS, DHT-PA and SEPAP) is one or several orders of magnitude slower than the above five schemes, since they utilize the BLS signatures primitive to support public auditing at the expense of low efficiency inherited from BLS signatures. The maximum throughput of DPDP and FlexDPDP is another one order of magnitude smaller than the three BLS-based PDP schemes, because building the Rank-Based Authenticated Skiplist (RBASL) or FlexList data structures is not efficient and quite memory-consuming.

Figure 4 shows that the verification time of DPDP and FlexDPDP is about one order of magnitude shorter than that

Table 4: Proportion of Challenge and Verify Time in a PDP Request

		MACPDP	APDP [9]	MRPDP [11]	SEPDP [10]	CPOR [12]	DPDP [7]	FlexDPDP [8]	PPPAS [19]	SEPAP [17]	DHT-PA [18]
Challenge	E	0.015%	0.037%	0.030%	0.001%	0.016%	0.001%	0.001%	0.040%	0.040%	0.050%
	N	0.010%	0.032%	0.025%	0.001%	0.006%	0.0003%	0.0004%	0.026%	0.027%	0.034%
Verify	E	0.011%	2.881%	5.541%	0.001%	0.072%	0.154%	0.210%	62.870%	68.083%	71.942%
	N	0.009%	2.574%	5.104%	0.001%	0.037%	0.108%	0.140%	57.180%	58.753%	63.338%

* Note: "E" means "EnclavePDP"; "N" means "Native PDP".

of the three BLS-based schemes, which seems contradictory to fact that the maximum throughput of the former is about one order of magnitude smaller than that of the latter when verifying the integrity of 1GB file as in the right of Table 3. We conducted another experiment to evaluate the proof generation time of those five schemes to generate proofs on 1GB file. We find that the proof generation time of DPDP is 10s, nearly 5 times of those three BLS-based schemes, i.e., 2.5s for PPPAS, 1.8s for DHTP-A and 2.3s for SEPAP, respectively. The proof generation time of FlexDPDP is also about 3 times of those BLS-based schemes. In fact, the proof generation of DPDP and FlexDPDP spends a large amount of time to build RBASL and FlexList, and the property information of the blocks to be checked needs to be sent back to the verifier, which introduces much more communication overhead than that of those BLS-based schemes. Moreover, the size of RBASL and FlexList also depends on the size of the file to be verified. The left part of Table 3 shows that DPDP and FLexDPDP perform better than the BLS-based schemes when verifying the integrity of smaller files.

Observation 5. To support dynamic auditing, the performance of PDP schemes like DPDP and FlexDPDP downgrades significantly, due to the expense of building and managing memory-consuming data structures.

Finally, we also conducted an experiment to evaluate the overhead incurred by EncalvePDP when performing integrity verification on a smaller file, i.e., 16KB. As shown in Table 3, when the number of concurrent threads is the same for 16KB file and 1GB file, verifying 16KB file by EnclavePDP introduces less overhead than 1GB file. For example, with the same 40 concurrent threads, enclave-enabled APDP imposed 3.9% overhead on 16KB file and 4.1% overhead on 1GB file. With the same 30 concurrent threads, enclave-enabled DHT-PA imposed 18.1% overhead on 16KB file, and 23.4% overhead on 1GB file. However, for other schemes, we cannot simply compare the overhead on 16KB file and 1GB file directly, because the number of concurrent threads launched to evaluate the maximum throughput can be quite different, e.g., 120 for DPDP and FlexDPDP on 16KB file and 5 on 1 GB file. Overall, when verifying 16KB file, the maximum throughput of EnclavePDP is still in the same order of magnitude as the native PDP, which indicates the overhead caused by EncalvePDP is still acceptable for practical deployment.

7 Related Works

Provable Data Possession Schemes. Many data integrity verification schemes [7–16], [30, 45, 46] have been proposed. Among them, SEPDP [10], DPDP [7], and FlexDPDP [8]) provided support to verify dynamic data. Mirror [16], CPOR [12] and Iris [31] extended PDP schemes to provide data integrity verification with data recovery if any data corruption is identified, i.e., proof of retrievability (POR) schemes. [11, 30] designed the integrity check of static data for multiple copies. PPPAS [19], DHTPA [18] and Qruta [14] proposed privacy-preserving auditing schemes using third parties. Many literature surveys (e.g., [25–29]) presented comprehensive summaries and comparison of the existing PDP scheme by defining a taxonomy of existing PDP schemes. However, these surveys primarily focus on a summary of the existing PDP schemes, without any practical implementation or evaluation of them on real-world cloud storage servers.

Securing Cloud Storage Systems. DEPSKY [47] proposed a cloud-of-clouds storage system, storing data on several cloud services to improve the data integrity and retrievability. Depot [48] designed a cloud storage system to guarantee the consistency of operations on data. It also protects the integrity of data by preventing unauthorized nodes from accessing the data objects. DEPSKY [47] still trusts the cloud storage platforms, while Depot [48] mainly focuses on the consistency and availability of the data. CloudProof [49] used cryptographic keys to create access control policies, which allow users to detect violations of integrity and also prove those violations to a third party. CloudProof mainly aims to provide security guarantees for the SLA (Service Level Agreement) to ensure that users will receive a certain compensation in case of cloud misbehavior.

Intel SGX-based Approaches. LibSEAL [37] presented a secure audit library to detect service integrity violations (e.g., committing operations of Git) by creating non-repudiable audit logs protected by Intel SGX. LibSEAL is implemented as a TLS library, which is not applicable to verify data integrity. EnclaveDB [50] is a secure database that guarantees the confidentiality and integrity of data and queries by placing sensitive data (tables, indexes and other metadata) in Intel SGX enclave. DelegaTEE [51] designed a brokered delegation scheme, which utilizes SGX for users to securely delegate their credentials of service providers to others. Ohrimenko et al [52] rely on SGX to perform privacy-preserving machine learning on collaborative data owned by multi-parties.

8 Conclusion

In order to enable users to independently and confidentially verify the integrity of their outsourced data on cloud storage servers, we present EnclavePDP, a general framework that utilizes Intel SGX to perform data integrity verification. We tailored Intel SGX SSL library and ported PBC libraries into Intel SGX. Then 10 representative PDP schemes are implemented based on EnclavePDP framework. We deployed EnclavePDP on a real-world cloud application (FastDFS) to evaluate its practicality. The experimental results show that EnclavePDP introduced a reasonable runtime overhead for different sizes of files, thus feasible to be deployed with existing cloud storage services via its convenient interfaces.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by Strategic Priority Research Program of Chinese Academy of Sciences (No.XDC02010900), National Key Research and Development Program of China (No.2016QY04W0903), Beijing Municipal Science and Technology Commission (No.Z191100007119010) and National Natural Science Foundation of China (No.61772078). Peng Liu is supported by NSF CNS-1814679.

References

[1] Dropbox bug wipes some users' files from the cloud. `https://www.engadget.com/2014/10/13/dropbox-selective-sync-bug/`, 2014.

[2] Amazon's cloud crash disaster permanently destroyed many customers' data. `https://www.businessinsider.com/amazon-lost-data-2011-4`, 2011.

[3] Tencent cloud says 'improper operations' led to data loss. `https://www.scmp.com/tech/article/2158785/tencent-cloud-says-improper-operations-led-data-loss-client-it-seeks-implement`, 2018.

[4] Christian Priebe, Divya Muthukumaran, Dan O' Keeffe, David Eyers, Brian Shand, Ruediger Kapitza, and Peter Pietzuch. Cloudsafetynet: Detecting data leakage between cloud tenants. In *Proc. of ACM CCSW*, 2014.

[5] A look back: U.s. healthcare data breach trends. `https://infosec.uthscsa.edu/sites/default/files/HITRUST_Report-US_Healthcare_Data_Breach_Trends.pdf`, 2012.

[6] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, May 2011.

[7] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proc. of ACM CCS*, 2009.

[8] Ertem Esiner, Adilet Kachkeev, Samuel Braunfeld, Alptekin Kupcu, and Oznur Ozkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. *ACM Transactions on Storage*, 12(4):23, 2016.

[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *Proc. of ACM CCS*, 2007.

[10] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *Proc. of ACM SecureComm*, 2008.

[11] Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. Mr-pdp: Multiple-replica provable data possession. In *Proc. of IEEE ICDCS*, 2008.

[12] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Proc. of ASIACRYPT*, 2008.

[13] Łukasz Krzywiecki and Mirosław Kutyłowski. Proof of possession for cloud storage via lagrangian interpolation techniques. In *Proc. of NSS*, 2012.

[14] B. Wang, B. Li, and H. Li. Oruta: privacy-preserving public auditing for shared data in the cloud. *IEEE Transactions on Cloud Computing*, 2(1):43–56, Jan 2014.

[15] Boyang Wang, Baochun Li, and Hui Li. Knox: privacy-preserving auditing for shared data with large groups in the cloud. In *Proc. of ACNS*, 2012.

[16] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *Proc. of USENIX Security*, 2016.

[17] J. Shen, J. Shen, X. Chen, X. Huang, and W. Susilo. An efficient public auditing protocol with novel dynamic structure for cloud data. *IEEE Transactions on Information Forensics and Security*, 12(10):2402–2415, Oct 2017.

[18] H. Tian, Y. Chen, C. Chang, H. Jiang, Y. Huang, Y. Chen, and J. Liu. Dynamic-hash-table based public auditing for secure cloud storage. *IEEE Transactions on Services Computing*, 10(5):701–714, Sep. 2017.

[19] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for secure cloud storage. *IEEE Transactions on Computers*, 62(2):362–375, Feb 2013.

[20] Intel® software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[21] Intel® software guard extensions ssl. https://github.com/intel/intel-sgx-ssl, 2019.

[22] Gnu multiple precision arithmetic trusted library for intel® software guard extensions. https://github.com/intel/sgx-gmp, 2019.

[23] Pbc library. https://crypto.stanford.edu/pbc/, 2019.

[24] Fastdfs. https://github.com/happyfish100/fastdfs, 2013.

[25] Faheem Zafar, Abid Khan, Saif Ur Rehman Malik, Mansoor Ahmed, Adeel Anjum, Majid Iqbal Khan, Nadeem Javed, Masoom Alam, and Fuzel Jamil. A survey of cloud computing data integrity schemes: Design challenges, taxonomy and future trends. *Computers & Security*, 65:29 – 49, 2017.

[26] S. G. Worku, Z. Ting, and Q. Zhi-Guang. Survey on cloud data integrity proof techniques. In *Proc. of IEEE AsiaJCIS*, 2012.

[27] Nouha Oualha, Jean Leneutre, and Yves Roudier. Verifying remote data integrity in peer-to-peer data storage: A comprehensive survey of protocols. *Peer-to-Peer Networking and Applications*, 5(3):231–243, Sep 2012.

[28] Mehdi Sookhak, Hamid Talebian, Ejaz Ahmed, Abdullah Gani, and Muhammad Khurram Khan. A review on remote data auditing in single cloud server: Taxonomy and open issues. *Journal of Network and Computer Applications*, 43:121 – 141, 2014.

[29] Lei Zhou, Anmin Fu, Shui Yu, Mang Su, and Boyu Kuang. Data integrity verification of the outsourced big data in the cloud environment: A survey. *Journal of Network and Computer Applications*, 122:1 – 15, 2018.

[30] Ayad F Barsoum and M Anwar Hasan. Integrity verification of multiple data copies over untrusted cloud servers. In *Proc. of IEEE Computer Society CCGRID*, 2012.

[31] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *Proc. of ACM ACSAC*, 2012.

[32] Giuseppe Ateniese, Seny Kamara, and Jonathan Katz. Proofs of storage from homomorphic identification protocols. In *Proc. of ASIACRYPT*, 2009.

[33] Boneh–lynn–shacham. Accessed on April 10, 2019.

[34] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proc. of NDSS*, 2017.

[35] S. Gueron. Memory encryption for general-purpose processors. *IEEE Security Privacy*, 14(6):54–62, Nov 2016.

[36] Innovative technology for cpu based attestation and sealing. https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing, 2013.

[37] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. Libseal: Revealing service integrity violations using trusted execution. In *Proc. of ACM EuroSys*, 2018.

[38] Elliptic-curve diffie–hellman. https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman, 2019.

[39] epoll. https://en.wikipedia.org/wiki/Epoll, 2019.

[40] Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 213–222. ACM, 2009.

[41] Intel® digital random number generator (drng) software implementation guide. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide, 2014.

[42] Intel(r) software guard extensions (sgx) protected code loader (pcl) for linux* os. https://github.com/intel/linux-sgx-pcl, 2017.

[43] Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2019.

[44] Bls signatures: better than schnorr. https://medium.com/cryptoadvance/bls-signatures-better-than-schnorr-5a7fe30ea716, 2018.

[45] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Proc. of Theory of Cryptography Conference*, 2009.

[46] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proc. of ACM CCSW*, 2009.

[47] Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *TOS*, 9(4):12:1–12:33, 2013.

[48] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proc. of UNSENIX OSDI*, 2010.

[49] David Molnar, Jay Lorch, , and Raluca Ada and Popa. Enabling security in cloud storage slas with cloudproof. Technical report, May 2010.

[50] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *Proc. of IEEE SP*, 2018.

[51] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. Delegatee: Brokered delegation using trusted execution environments. In *Proc. of USENIX Security*, 2018.

[52] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *Proc. of USENIX Security*, 2016.