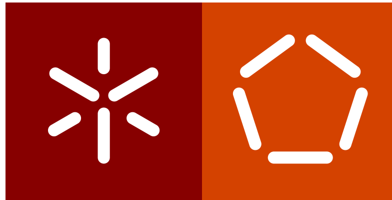
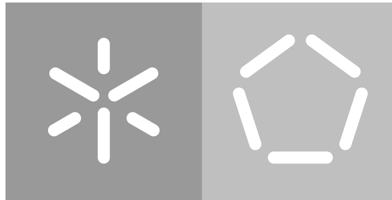


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mariana Martins de Sá Miranda

S2Dedup:
SGX-enabled Secure Deduplication System

December 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Mariana Martins de Sá Miranda

S2Dedup: SGX-enabled Secure Deduplication System

Master dissertation
Integrated Master in Informatics Engineering

Dissertation supervised by
João Tiago Medeiros Paulo
Rui Carlos Oliveira

December 2020

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Atribuição
CC BY

https://creativecommons.org/licenses/by/4.0/

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

AGRADECIMENTOS

A realização desta dissertação não seria possível sem o contributo e apoio de algumas pessoas, pelo que gostaria de expressar a minha gratidão a todos os que me ajudaram a atingir os objetivos e metas estabelecidas.

Em primeiro lugar, ao meu orientador, Doutor João Paulo, por toda a motivação, disponibilidade e incansável paciência e dedicação em todas as etapas desta dissertação.

Ao meu co-orientador, Doutor Rui Oliveira pelo apoio no decorrer deste projeto.

Ao meu supervisor, Bernardo Portela que esteve sempre disponível para todas as minhas dúvidas e por pôr ao dispor a sua experiência e conhecimentos de segurança.

A todos meus colegas do HASLab, em especial à Tânia Araújo Esteves, pelo seu apoio e disponibilidade a ajudar em qualquer situação que necessitasse.

Por último, aos meu pais, irmãos e amigos, por todo o apoio e incentivo durante, não apenas a realização desta dissertação, mas toda a minha vida académica.

RESUMO

Com os atuais avanços tecnológicos, cada vez mais informação está a ser digitalizada, resultando assim num aumento exponencial nos dados guardados em formato digital. Este crescimento sem precedentes tem levantado preocupações acerca do espaço e custo dos sistemas de armazenamento, criando uma necessidade de explorar mecanismos que visem mitigar este problema.

Uma estratégia que se foca neste problema é a técnica conhecida por deduplicação, que se baseia no facto que dados idênticos estão a ser gerados e armazenados repetidamente, resultando assim num consumo desnecessário de espaço de armazenamento em disco. Deste modo, a deduplicação propõe uma análise dos dados armazenados e, subsequentemente, a eliminação de cópias redundantes, economizando espaço e custos de armazenamento.

Serviços como DropBox e Google Drive aplicam essa estratégia, contudo, o processamento de dados que pertencem a vários ulilizadores fomenta preocupações de privacidade e segurança, especialmente quando este é realizado em fornecedores de serviços de armazenamentos terceiros. A abordagem tradicional para resolver estes problemas é os utilizadores enviarem os seus dados já cifrados. Contudo, usar uma cifra probabilística implica que dados idênticos podem resultar em textos cifrados diferentes, o que torna impossível encontrar cópias redundantes e, consequentemente, aplicar a deduplicação.

Deste modo, propomos o S2Dedup, um sistema de deduplicação seguro que explora tecnologias emergentes de segurança assistida por hardware. Mais especificamente, a solução proposta recorre ao Intel SGX (*Software Guard Extensions*), de forma a permitir a deduplicação de dados entre utilizadores em infraestruturas de armazenamento de terceiros, sem descuidar da segurança e privacidade dos seus dados. Além disto, o S2Dedup foi projetado para oferecer suporte a vários esquemas de segurança, cada um oferecendo diferentes níveis de espaço poupado, desempenho e privacidade. Esta característica é fundamental para garantir a aplicabilidade do S2Dedup a uma gama ampla de sistemas com requisitos diferentes.

Um protótipo do S2Dedup é implementado e avaliado com cargas de trabalho sintéticas e realistas, assim como comparado com as soluções alternativas de deduplicação seguras do estado da arte. Os resultados mostram que é possível implementar técnicas de segurança mais robustas e ao mesmo tempo manter bons resultados de desempenho e até mesmo alcançar, em alguns casos, uma melhoria na eficácia da deduplicação em comparação com as soluções do estado da arte.

Keywords: Deduplicação, Segurança, Armazenamento, Hardware Confiável, SGX

ABSTRACT

With the current advancements in computer technologies, more and more information is being digitized, resulting in an exponential increase of digital data. This unprecedented growth raises concerns about the space and cost of data storage, creating the need to explore mechanisms that strive to mitigate this ever-increasing data problem.

A strategy that addresses this issue is the technique known as deduplication, which leverages from the fact that identical data is being generated and stored repeatedly, consuming unnecessary storage space. Therefore, deduplication proposes an analysis of the stored data and subsequently the elimination of redundant copies, thus saving storage space and costs.

Services like DropBox and Google Drive support deduplication, however, eliminating redundant information across data belonging to multiple users raises privacy and security concerns, specially when this is done at third-party untrusted infrastructures. The conventional approach to ensure data privacy is for the users to outsource their data in an encrypted format. However, using standard probabilistic encryption implies that identical data will result in different ciphertexts, which makes it impossible to find redundant copies, and consequently apply deduplication.

Therefore we propose S2Dedup, a secure deduplication system that explores emergent hardware-assisted security technologies. In more detail, the proposed solution leverages Intel *Software Guard Extensions* to enable cross-user privacy-preserving deduplication at third-party storage infrastructures. Furthermore, S2Dedup is designed to support multiple security schemes, each providing different trade-offs in terms of deduplication space savings, storage performance, and privacy. Such feature is key to improve S2Dedup's applicability to a wider range of applications with different requirements.

A prototype of S2Dedup is implemented and evaluated with both synthetic and realistic workloads whilst being compared to the state of the art secure deduplication solutions. The results show that it is possible to implement more robust security techniques, while maintaining overall interesting performance results and even achieve, in some cases, an improvement of deduplication effectiveness when compared to the state of the art solutions.

Keywords: Deduplication, Security, Storage, Trusted Hardware, SGX

CONTENTS

1 INTRODUCTION 1
- 1.1 Problem Statement 2
- 1.2 Objectives and Contributions 3
- 1.3 Outline 3

2 STATE OF THE ART 5
- 2.1 Trusted Hardware 5
  - 2.1.1 Intel SGX 6
- 2.2 Deduplication 8
  - 2.2.1 Secure deduplication 10
  - 2.2.2 Secure deduplication with trusted hardware 13
- 2.3 Discussion 15

3 S2DEDUP ARCHITECTURE 17
- 3.1 General Architecture 17
  - 3.1.1 Flow of Requests 18
- 3.2 Deduplication Engine 20
- 3.3 Security Solutions 23
  - 3.3.1 Enclave Based Scheme 24
  - 3.3.2 Epoch Based Scheme 25
  - 3.3.3 Epoch and Exact Frequency Based Scheme 26

4 PROTOTYPE 28
- 4.1 Server Implementation 28
  - 4.1.1 Deduplication engine 29
  - 4.1.2 Enclave Based Scheme 30
  - 4.1.3 Epoch Based Scheme 32
  - 4.1.4 Epoch and Exact Frequency Based Scheme 32
  - 4.1.5 Estimated Frequency Based Scheme 33
- 4.2 Client Implementation 34

5 EVALUATION 35
- 5.1 Methodology 35
- 5.2 Workloads 36
- 5.3 Setups 36
- 5.4 Environment Testing 38
- 5.5 Prototype Testing 39

5.5.1 Synthetic experiments 40
5.5.2 Realistic experiments 47
5.6 Discussion 52
6 conclusion 53
6.1 Future Work 53

a appendix 61
a.1 Resource Usage 61
a.1.1 Environment Results 61
a.1.2 Prototype Results 63

LIST OF FIGURES

Figure 1	Secure remote computation.	6
Figure 2	*Count-Min Sketch* algorithm.	13
Figure 3	Dang and Chang proposed three-tier architecture.	14
Figure 4	System architecture overview.	18
Figure 5	Deduplication system architecture.	21
Figure 6	Server implementation.Server implementation.	29
Figure 7	Example of the "Epoch and Exact Frequency Based" setup for t=15.	33
Figure 8	Client implementation.	34
Figure 9	Latency for *sequentials writes* for *dist_highperf*.	42

LIST OF TABLES

Table 1 Comparative study of security properties of hardware-based trusted computing architectures. 7
Table 2 Environment results. 38
Table 3 Variation of throughput and deduplication ratio depending on the epoch duration. 40
Table 4 Synthetic tests results for the distribution *dist_highperf* for the in-memory implementation. 43
Table 5 Synthetic tests results for the distribution *dist_highperf* for the persistent implementation. 44
Table 6 Synthetic tests results for the distribution *dist_kernels* for the in-memory implementation.. 45
Table 7 Synthetic tests results for the distribution *dist_kernels* for the persistent implementation. 46
Table 8 Realistic tests results for the trace *mail*. 49
Table 9 Realistic tests results for the trace *homes*. 50
Table 10 Realistic tests results for the trace *web-vm*. 51
Table 11 Resource usage by the *read* operations of the environment tests. 61
Table 12 Resource usage by the *write* operations of the environment tests. 62
Table 13 Resource usage by the *read* operations of the synthetic tests for the distribution *dist_highperf*. 64
Table 14 Resource usage by the *write* operations of the synthetic tests for the distribution *dist_highperf*. 66
Table 15 Resource usage by the *read* operations of the synthetic tests for the distribution *dist_kernels*. 68
Table 16 Resource usage by the *write* operations of the synthetic tests for the distribution *dist_kernels*. 70
Table 17 Resource usage by the realistic tests for the trace *mail*. 71
Table 18 Resource usage by the realistic tests for the trace *homes*. 72
Table 19 Resource usage by the realistic tests for the trace *web-vm*. 73

LIST OF LISTINGS

2.1 Example of an EDL file syntax. 8
3.1 Extract from the deduplication engine interface. 20
4.1 Extract from the EDL file. 31

ACRONYMS

API Application Programming Interface.

CE Convergent Encryption.

EDL Enclave Description Language.

IOPS (Input/Output Operations Per Second.

ISCSI Internet Small Computer System Interface.

MLE Message-Locked Encryption.

NBD Network Block Device.

OS Operating System.

PSW Intel SGX Platform Software.

SGX Intel Software Guard Extensions.

SPDK Storage Performance Development Kit.

1

INTRODUCTION

Over the last few years, due to advancements in computer and storage technologies, there has been an exponential growth in the amount of digital information. For instance, according to projections provided by the International Data Corporation (IDC) [4], by 2025 the stored digital information will reach 175 zettabytes, which compared to the 33 zettabytes of 2018 and 16.1 zettabytes of 2016, makes the discussion and implementation of mechanisms to efficiently store the ever-increasing volumes of data of the uttermost importance.

A recurrent behaviour noticeable in storage services is that identical data is being stored repeatedly while consuming unnecessary storage space. Deduplication strives at eliminating these redundant copies, thus enabling savings in storage space and costs, as well as, in network bandwidth. A study conducted by Meyer and Bolosky revealed that deduplication can reduce stored data by up to 83% in backup storage and 68% in primary storage systems.

Cloud storage providers like Dropbox and Google Drive employ this mechanism in their infrastructures [16, 36, 50]. Additionally, to achieve higher deduplication gains these services perform cross-user deduplication, which implies that if one or more users upload the same data, the cloud provider will only store one copy. This approach is viable when security and privacy are not a main concern. However, from the moment that users outsource their data to a third-party service, they no longer have sole control over their information which can be exploited by malicious adversaries. As an example, Dropbox in 2012 suffered a major data breach where more than 68 million user accounts were leaked on to the internet [5]. Or more recently, in 2020 a vpnMentor's research team found a serious breach in an open Amazon S3 bucket owned by Data Deposit Box, a secure cloud storage provider [14]. This leak exposed detailed information about 270,000 private files uploaded by customers through the company's secure cloud storage service. Namely, this attack revealed personally identifiable information (PII) of customers, such as unencrypted administrator usernames, passwords, and users' local computer name and globally unique identifier (GUID), which could have severe consequences for those affected.

In order to avoid this kind of exploits, users should encrypt their data, with their own encryption keys, before outsourcing it to third-party storage services. However, this leads to the scenario where identical data, owned by different users, results in ciphertexts with

distinct content, thus making it impossible to apply deduplication and leverage its space saving benefits.

1.1 PROBLEM STATEMENT

Traditionally, convergent encryption (CE) is used as the mean to enable cross-user deduplication over encrypted data [30]. Namely, data is encrypted at the users' premises with a deterministic encryption scheme in which the secret key is generated from the data content itself. This means that identical content encrypted by different users will generate matching ciphertexts thus enabling deduplication. However, employing a deterministic scheme comes with the cost of being susceptible to numerous attacks, namely leakage of information where a server, or even a client, can detect if a certain data is a duplicate.

This leads, for example, to the "confirmation of a file attack", where an adversary can deduce whether a file was previously sent to a storage provider by simply uploading the same file and verifying if deduplication occurred. This can be easily achieved in a client-side deduplication solution since it is first sent to the server a content's hash signature and, only if is not duplicated, that is sent the actual content. Therefore, an adversary can check if a file exists in a storage provider by verifying if that file is sent to the server. It is also possible to verify if deduplication occurred by observing the time differences between the upload of files, given that if a new file is being stored it would require a longer time to finish the operation than for a duplicate file that is already stored and does not need to be persisted [46].

Another example of the consequences of employing a deterministic scheme is it being susceptible to the "learn the remaining information attack", which is done by performing a brute-force attack, where an adversary repeatedly tries every possible combination that could make up the content that is trying to obtain, until finding one that matches the ciphertext. This attack allows an attacker to recover files falling into a known set by applying CE to each candidate message and comparing the ciphertext of the sampled message with the target ciphertext. For example, letters from the bank usually follow a certain structure, containing parts of boilerplate legal text plus a few with critical information, such as a user's bank account number and password. An attacker who knows the boilerplate content might be able to infer a file original content by trying every possible value for the unknown parts, encrypting it using convergent encryption and then comparing it with the ciphertext that is trying to determine [3].

Lately, there has been a rise of hardware-assisted security technologies (e.g. Intel SGX [11, 25]), which provide a secure trusted execution environment to perform critical operations. This feature can be explored to aid the process of secure deduplication since the secure execution environment provided by these technologies allows for the data to be handle in

its original form in an untrusted storage server. This way, before sending their data to the storage service, a user can protect it using a probabilistic encryption scheme instead, and once it reaches the storage service, thanks to the trusted execution environment, it is possible to securely decrypt it and apply deduplication. However, this approach is vaguely explored by the literature and could provide novel trade-offs in terms of security and performance when compared to traditional CE solutions.

1.2 OBJECTIVES AND CONTRIBUTIONS

The main goal of this dissertation is then to develop a system that performs secure deduplication based on trusted hardware, without neglecting the storage performance and deduplication effectiveness. Namely, we propose the following contributions:

- The design and implementation of S2Dedup, a secure deduplication prototype that takes advantage of the functionalities provided by trusted hardware (specifically Intel SGX), as well as, state of the art frameworks that allow the development of efficient storage solutions (namely SPDK [12, 13]).
- The proposal of different secure deduplication schemes integrated within S2Dedup, that balance security guarantees with deduplication performance and effectiveness. Namely, we propose a novel approach that combines the concept of epochs with the idea of limiting the number of duplicates per chunk. This solution is based on the notion of masking the real number of duplicate copies so that an adversary cannot correctly infer about the content that was previously stored.
- A detailed and extensive evaluation of S2Dedup prototype resorting to both synthetic and realistic workloads. The evaluation contemplates more than 500 hours of experiments and 60 TB of data read/written. Finally, we also compare our prototype with the state of the art approaches for secure deduplication [27, 40]. The results show that is possible to implement more robust security techniques, while maintaining overall interesting performance results and even achieve, in some cases, an improvement of deduplication effectiveness when compared to the state of the art solutions.

1.3 OUTLINE

The document is organized into 5 different chapters: State of the Art (2), Architecture (3), Prototype (4), Evaluation (5) and Conclusion (6).

Chapter 2 introduces the concept of trusted hardware while presenting widely-used hardware-based trusted computing architectures and their security properties, paying special attention to how Intel SGX operates and the functionalities that it offers. Then, a definition for deduplication is presented along with the main features to consider when designing a deduplication engine. Finally, existing secure deduplication techniques and solutions based on trusted hardware are also detailed. Chapter 3 introduces S2Dedup architecture and design, while Chapter 4 describes its implementation. Chapter 5 details the conducted experimental evaluation and the results obtained for our prototype. Lastly, Chapter 6 concludes the document.

2

STATE OF THE ART

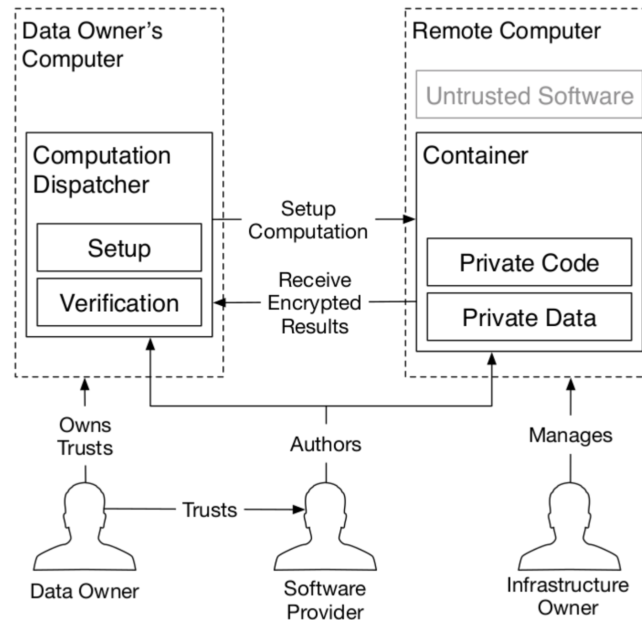
As explained previously, traditional encryption schemes limit the possibility of performing deduplication at third-party storage services. Therefore, alternative procedures must be explored. In this work, we do so by contemplating hardware-assisted security technologies, specifically the functionalities provided by Intel SGX.

In this chapter, we start by introducing the concept of trusted hardware and discussing existing solutions, paying special attention to how Intel SGX operates and the functionalities that it offers (Section 2.1). Then, in Section 2.2, we describe the key concepts behind deduplication, while detailing existing secure deduplication schemes, as well as, current solutions based on trusted hardware.

2.1 TRUSTED HARDWARE

Nowadays, with the rise of hardware-assisted security technologies, cloud service providers are progressively employing trusted hardware, especially Intel SGX [11], at their security infrastructures (e.g. IBM Cloud Data Guard [8] and Azure Confidential Computing [2]). This technology is key for enabling secure remote computation (Figure 1), in other words, for being able to securely execute programs in an untrusted software environment [26].

Even though there are alternative software-based trusted computing architectures, those solutions are limited and cannot achieve the same security guarantees as hardware-based architectures. The reason for this is that in a software-based solution an attacker can always manipulate software if the Operating System (OS) is not trusted. Whereas hardware-based architectures can protect applications against attackers even when they have full control over the system, namely when an application or even the OS have been compromised. This happens because it is much harder for an attacker to modify hardware functionalities, provided that the hardware is considered to be immutable [43].



Source: Intel sgx explained [25].

Figure 1: Secure remote computation.

Throughout the years, several hardware-based architectures have been proposed. Maene et al. [43] conducted a survey exploring architectures proposed by both academia and industry. The study's conclusions concerning security properties are shown in Table 1.

Although there are several options, it was chosen to work with Intel *Software Guard Extensions* (SGX) [11], due to the reliable computing features it makes available, the security and confidentiality it offers and that its wide use across both academia [26, 34, 37, 53, 64] and industry [2, 8]. Moreover, it is available in commodity hardware, which facilitates its practical application.

2.1.1 *Intel SGX*

Intel *Software Guard Extensions* (SGX) [11] provides a set of instructions in recent Intel's processors that ensure integrity and confidentiality. This is achieved by assuming that any layer in the computer's software stack (firmware, hypervisor, OS) is potentially malicious, thus SGX only trusts the CPU's microcode and a few privileged containers, known as enclaves. Enclaves are private regions of memory whose contents are protected and unable to be accessed by any process outside of it, including processes running at higher privilege levels.

Architecture	Security Properties						
	Isolation	Attestation	Sealing	Dynamic RoT	Code Confidentiality	Side-Channel Resistance	Memory Protection
AEGIS	●	●	●	●	●	○	●
TPM	○	●	●	○	●	—	◐
TXT	●	●	●	●	●	●	◐
TrustZone	●	○	○	●	○	○	○
Bastion	●	○	●	●	●	○	●
SMART	○	●	○	●	○	○	○
Sancus	●	●	○	●	○	○	○
Soteria	●	●	○	●	●	○	○
SecureBlue++	●	○	●	●	●	○	●
SGX	●	●	●	●	●	○	●
Iso-X	●	●	○	●	○	○	●
TrustLite	●	●	○	○	○	○	○
TyTAN	●	●	●	●	○	○	○
Sanctum	●	●	●	●	●	●	○

● = Yes; ◐ = Partial; ○ = No; — = Not Applicable

Source: Hardware-based trusted computing architectures for isolationand attestation [43].

Table 1: Comparative study of security properties of hardware-based trusted computing architectures.

The memory region reserved for the enclave is called *Enclave Page Cache* (EPC), which consists of 4 kB pages that store enclave code and data. Their content is only decrypted when entering the CPU package, given that it is protected by the enclave mode, and then re-encrypted when leaving to the EPC memory region.

Interaction. From an outsider perspective, the enclave can be characterized as a black-box, whose only method to interact with is through communication channels that bridge the trusted and untrusted environments.

To define these channels, Intel provides an interface definition language called *Enclave Description Language* (EDL), which enables the specification of ECALLs (*Enclave Calls*) and OCALLs (*Enclave Calls*). ECALLs enables an untrusted application to invoke a predefined set of functions inside of the enclave and OCALLs allows for the opposite, to call out to the untrusted application from inside the enclave.

Listing 2.1 demonstrates an example for the EDL file syntax. The `trusted` section must contain the functions to communicate with the enclave and the `untrusted` section, the functions to call out of the enclave.

```
enclave {
    trusted {
        /* define ECALLs here. */
    };

    untrusted {
        /* define OCALLs here. */
    };
};
```

Listing 2.1: Example of an EDL file syntax.

Sealing. When the enclave is closed, either by the application itself or a power cut, its contents are typically lost. To avoid this, the secrets that need to be preserved when an enclave is closed must be stored outside of the enclave boundary. However, simply storing the enclave secrets outside of it, without any encryption, can lead to a major security breach. To prevent this, there is a mechanism that is known as sealing that is responsible for the process of encrypting enclave secrets before being stored persistently.

The encryption is done by using a private *Seal Key* that is unique to a particular SGX enabled platform and enclave, and is unknown to any other entity. That key is used to encrypt data to that particular platform or to decrypt data already on the platform. The process of these encrypt and decrypt operations is referred to as sealing and unsealing, respectively.

Remote and Local Attestation. Another interesting security property of SGX is that it provides local and remote attestation. Local attestation can be very useful, because sometimes an application might require the usage of two or more enclaves that need to collaborate with each other and, therefore, the enclaves must prove to each other that can be trusted. The process of remote attestation is similar but occurs between an enclave and a third-party application/service running at a distinct server. Attestation allows assessing the identity of an enclave, its structure, the integrity of its code and ensuring that this code is running on a genuine SGX processor.

2.2 DEDUPLICATION

Deduplication is a technique that allows eliminating redundant copies of data within a dataset, thus reducing the required storage space and consequently lowering storage

costs [46]. It is similar to compression, but instead of only identifying intra-file or block redundancy, deduplication can also eliminate redundancy between files or blocks stored at different times or even at servers in different locations.

Although there are many variations on how deduplication operates, it generally follows the following steps:

1. Separate incoming data into fixed or variable size chunks;
2. Calculate an hash sum for each chunk using a cryptographic hash function;
3. Compare the returned hash values with the already existing ones in an index:
   a) If an hash value already exists, this means that a duplicate chunk has already been stored. Thus the new chunk does not need to be stored again and, whenever requested (*read* operation), the content of the previously stored chunk is returned instead;
   b) If not, a new chunk is stored and the index updated.

There are several categories for classifying deduplication solutions, such as their granularity, that is, how data is partitioned into the chunks that are going to be compared during deduplication. If it occurs at the file level, the deduplication engine takes into account the entire file, eliminating duplicate copies of the same file, which leads to fewer chunks to index and process. However, this also implies that the smallest alteration to a file requires storing a full copy of it, thereby decreasing deduplication space savings (ratio). An alternative approach is partitioning chunks at the block level, which means that data is partitioned into fixed-sized or variable-sized chunks. This approach offers higher deduplication ratios given that smaller chunks make it more probable to find redundant information. However, this approach leads to increased processing and metadata size (*e.g.*, index) overhead.

Deduplication can further be categorized based on the point in time that it occurs, in other words, if the process happens before or after data is written to disk. In case it happens before, it is called inline deduplication and I/O requests are intercepted and processed (*i.e.*, deduplicated) before being persisted at the storage medium. This approach can introduce major overhead for write requests latency, but avoids writing redundant content to disk. With the latter approach, known as offline deduplication, data is first written to the storage system and only later (in background) checked for duplicates. Although this alternative solves the overhead introduced at the critical I/O path of write requests, it also entails that is necessary to temporarily store redudant data at the storage medium, thus requiring more storage space than inline deduplication.

Another important decision to take when implementing deduplication is its location, particularly, if it will happen at server-side or client-side. Regarding the first approach, the

client sends the data (*e.g.*, entire file) to the server and only there is checked for duplicates. In the case of client-side deduplication, data partitioning and hashing are done at the client that then sends the hash signatures to the server. The server checks for duplicate hash signatures at the index and only requests unique chunks from the client, thereby enabling network bandwidth savings.

2.2.1 *Secure deduplication*

Throughout the years, different solutions have been proposed in an effort to ensure secure deduplication. A ground-breaking approach from which many other solutions developed from is the concept of convergent encryption (CE), introduced by Douceur et al. [30]. Convergent encryption is a deterministic symmetric encryption scheme that not only provides confidentiality but also enables deduplication across ciphertexts generated by different users. This is accomplished because in CE the encryption key is generated from the cryptographic hash value of the plaintext content, which leads to identical plaintexts producing the same keys and, consequently, the same ciphertexts. In other words, a user derives the key K from message M with a cryptographic hash function H and then encrypts M with K resulting in a ciphertext C, which can be translated to K = H(M), C = E(K,M), where E is a block cipher.

Furthermore, the convergent encryption scheme also includes a phase where a tag T is generated that can be used to detect duplicates. This is possible because the tag also results from the plaintext message itself. This way, for two identical messages, the generated tags will be the same. It is worth taking note that this tag is derived independently from the convergent key, thus not being possible to deduce the convergent key from it and compromising data confidentially.

However, CE due to its deterministic nature, it introduces frequency leakage, so that an adversary can infer if certain data was deduplicated or not, thus leading to the "confirmation of a file attack". Moreover, it is also susceptible to offline brute-force attacks, where a adversary can recover original plaintext data falling into a known set by applying CE to each candidate message and comparing the ciphertext of the sampled message with the target ciphertext, also known as the"learn the remaining information attack".

Bellare et al. proposed a file-level encryption solution known as *Message-Locked Encryption* (MLE) [20], wherein the key used for encryption and decryption, much like in convergent encryption, is derived from the message itself. MLE suggests four variants of the convergent encryption approach - CE, HCE1 (*Hash Convergent Encryption 1)*, HCE2 (*Hash Convergent Encryption 2)* and RCE (*Randomized Convergent Encryption*). Of these schemes, it is worth pointing-out RCE, since it is the most efficient, being able to generate the key, encrypt the

message and produce the tag in a single pass over the plaintext data. This is only possible because, unlike CE, HCE1 and HCE2 that first derive its key K from the message and only later encrypt it, requiring at least two passes over the data, the RCE approach first picks a random symmetric encryption key L and then, in one pass, encrypts the message using L and generates the MLE key K simultaneously. Finally, it encrypts L using K and derives the tag from K, resulting in a single pass over the data.

Unfortunately, since the encryption key is generated from the message itself, MLE is again susceptible to brute-force attacks and frequency leakage. Also, RCE is vulnerable to duplicate faking attacks, meaning that a user might not be able to retrieve their original message because it can be replaced by a fake one without being noticed. In response to this problem, Bellare and Keelveedhi proposed an interactive version of RCE, called interactive randomized convergent encryption (IRCE) [19]. In IRCE, whenever data is uploaded, the server replies back with the necessary metadata for integrity validation, making it possible for an honest user to interact with the server in order to verify if indeed the original message is stored. Furthermore, Chen et al. presented a solution named *Block-level Message-Locked Encryption* (BL-MLE) [23], that extends MLE in order to support both file-level and block-level deduplication over encrypted data.

Since MLE is susceptible to brute-force attacks, Keelveedhi et al. [36] proposed a new architecture called DupLESS (*Duplicateless Encryption for Simple Storage*). To withstand this type of attack, DupLESS converts a predictable message into an unpredictable one with the aid of a key server (KS) that is separate from the storage service. In this system, a client requests from the KS a convergent key to encrypt their message via an oblivious pseudorandom function (OPRF) protocol. This key is generated by the KS based on the hash of a message and a system-wide key. To stop external attackers, KS requires the client to previously authenticate, as well as, sets a limit on the number of requests that a client can make during a fixed time interval. As long as the KS remains inaccessible to the attackers, high security is achieved. On the other hand, DupLESS can deliver at least the same security as MLE, even if both the key server and storage service are compromised.

2.2.1.1 *Optimizations*

Besides security, efficiency and data availability also play a very important role in deduplication. In fact, traditional solutions based on convergent encryption must manage a large number of convergent keys, which must be stored in a resilient fashion at the client premises. Dekey [39] is an efficient and reliable key management scheme for block-level deduplication, that removes the responsability of key management from the users and instead builds upon secret shares of the original convergent keys while distributing these across multiple KM-CSPs (*Key-Management Cloud Service Providers*). The secret shares are created by making use of the RSSS (*Ramp Secret Sharing Scheme*) [22, 29], which allows the key management to

adapt to different reliability, confidentiality, storage overhead and performance levels. This is possible because RSSS allows tuning a variety of parameters, specifically:

- the number of shares to generate from a secret (n), which translates to the number of KM-CSPs;
- the minimum number of shares necessary to recover information from the secret (k), meaning that the higher is k, the lower is the reliability level;
- the confidentiality level (r), which ensures that no information about the secret can be deduced from any r shares.

Another solution that seeks to address the problem of security and efficiency is SecDep [66], whose aim is to resist brute-force attacks and to reduce large key space and computation overheads. This is accomplished because SecDep employs User Aware Convergent Encryption (UACE) and Multi-Level Key management (MLK). By applying UACE, SecDep resists brute-force attacks, since file-level CE keys are generated using a server-aided HCE (*Hash Convergent Encryption)*, whereas for chunk-level applies an efficient user-aided CE approach, which also reduces computation overheads. Regarding MLK, it lowers key space overhead by encrypting chunk-level keys using file-level keys, thus an increase in the number of users, will not linearly increase the key space overheads. Moreover, MLK splits a file-level key into share-level keys via *Shamir secret sharing scheme* [54] and sends them to multiple key servers, which eliminates the chance of having single-point-of-failures.

TED (*Tunnable Encrypted Deduplication*) [40] is solution proposed by Li et al. that also relies on a server-aided key manager. TED introduces a new way to compute a block's hash, which bases the key derivation on both the chunk content and the number of chunk copies, thus relaxing the deterministic encryption nature. This property allows controlling the maximum number of duplicate copies for a given chunk, parameter that is referred to as t. By choosing a bigger or smaller t value, a user can configure the deduplication effectiveness, and, consequently, control the storage blowup factor and information leakage. Namely, if t is defined as 1, it means that it is only possible to have one copy per chunk, leading to blocks with the same content always having different hashes, thus giving the perception that the system is operating on data with no duplicates. On the opposite end, if t tends to ∞, it enables an infinite number of copies per block, which provides the same security guarantees as not having this control mechanism.

The counter that maintains the number of copies per chunk is implemented with the *Count-Min Sketch* [24] algorithm (Figure 2), which allows obtaining a frequency estimation for each unique chunk, whilst protecting the chunk identities and fixing the memory footprint for the corresponding metadata structure. This algorithm is implemented by having a

two-dimensional array with r rows and w counters per row. It is also necessary to define a hash function ($H1$,$H2$,..) per row that returns a value from 1 to w, which represents the counter's index for its respective row. This way, when it is necessary to provide the number of duplicates for a chunk (c), the short hashes $H1(c)$...$Hr(c)$ are calculated and the minimum counter indexed by (x,$Hx(c)$), being x each one of the rows, corresponds to the estimated number of duplicates for that chunk. However, it is worth noting that w is generally smaller than the number of different chunks, which leads to hash collisions, and, consequently to overestimated values. On the other hand, it has its perks in terms of security, the approximate counting protects the chunk information from an attacker since it is not able to infer a chunk's content from its short hashes.

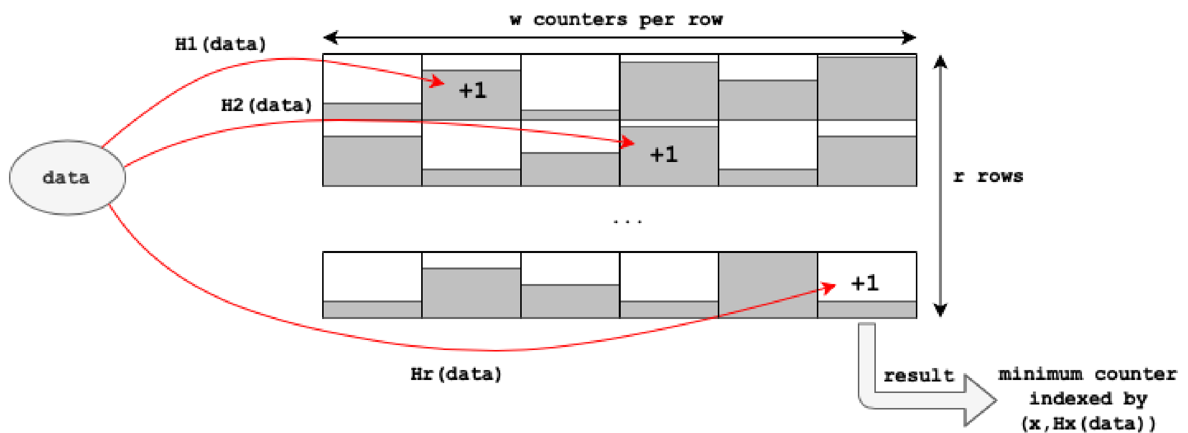


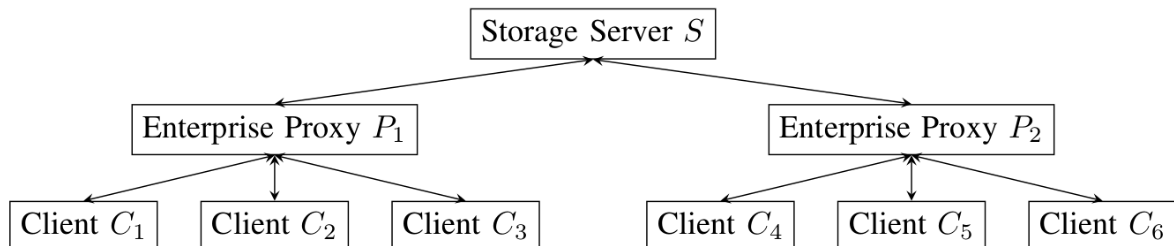
Figure 2: *Count-Min Sketch* algorithm.

To resist offline brute-force attacks, the mentioned solutions DupLESS [36], Dekey [39], SecDep [66], SecDep [66], TED [40] and others like ClouDedup [51] or [55], rely on the aid of additional independent key servers, which is not always possible, especially in a commercial context. That is why Liu et al. proposed the first single-server scheme that supports client-side encryption [41], where by exploiting the PAKE (*password authenticated key exchange*) [21] protocol, two parties can privately compare their secrets and share their encryption keys, not needing additional key servers.

2.2.2 *Secure deduplication with trusted hardware*

Deduplication with trusted hardware is a very recent concept, hence why the literature for this topic is still limited. Dang and Chang [27] introduced a solution that achieves the

bandwidth savings of client-side deduplication while preventing side-channel attacks. To this end, they proposed a three-tier architecture composed of three parties: the storage server S, enterprise proxies Ps and clients Cs (Figure 3). In this system, each client C uploads their files to a proxy P where it is performed an intermediary deduplication step. The chunks are then sent to the storage server S to perform cross-proxy deduplication.



Source: Privacy-Preserving Data Deduplication on Trusted Processors [27].

Figure 3: Dang and Chang proposed three-tier architecture.

To achieve data confidentiality, the solution relies on two encryption layers, accomplished by leveraging trusted SGX-enabled processors that offer a protected execution environment. The first layer is implemented at the client that encrypts data with a deterministic message-derived key that was received from the proxy enclave via a blind signature protocol. The second layer is performed by both the proxy and storage server and introduces randomness. This randomness is computed using a semantically secure encryption scheme and achieves protection from equality of information leakages while data is stored, that is, masks if two records represent the same file.

The proxies and storage server work in epochs, so deduplication is only performed at the end of an epoch, which protects from an adversary detecting equality of information leakages for data stored in different epochs. To further prevent this type of leakages, which could be deduced from the access patterns that the SGX-enabled processor performs during an epoch, the deduplication is done using a privacy-preserving compaction [28] that can mask any correspondence between files. Also, the traffic between the proxies and storage server is padded to prevent traffic analysis from revealing sensitive information or metadata.

Another solution that implements storage functionalities with trusted hardware is TrustFS [32]. TrustFS is a modular stackable file system framework that executes critical operations in a secure trusted hardware environment provided by Intel SGX. To support secure deduplication TrustFS employs two enclaves, one is used to re-encrypt incoming data since it was encrypted with a user's own key and to be able to find duplicates all data must be consistent. Therefore, the incoming data must first be decrypted with their owner's key and then encrypted with the server's secure key. The other enclave is used to calculate a chunk's cryptographic hash value so that when it is indexed in the metadata of deduplication, the original content cannot be easily deduced. In this solution to reduce side-channel attacks the

deduplication is performed in epochs as well, which prevents from an adversary finding out if a chunk was already stored in previous epoch, but it comes with the cost of lowered storage savings.

2.3 DISCUSSION

As described in Section 2.1, there are several platforms that provide secure and isolated execution environments, which allow for the processing and storage of sensitive data. Of these, it was chosen Intel SGX [11], since it offers useful features and is widely used across both academia [26, 34, 37, 53, 64] and industry [2, 8]. Moreover, it is also available in commodity hardware, which facilitates its practical application.

When it comes to deduplication, the traditional solutions that aim to achieve secure deduplication usually apply a deterministic encryption scheme, like convergent encryption or some sort of variation that developed from this concept (Section 2.2.1). However, these solutions have some security flaws or rely on the use of a server-aided key manager, which requires additional resources.

One of these flaws is centered around the fact that traditional approaches allow for the storage server to be able to recognize when a block is duplicate or not, without any control mechanism. This is inherently a requirement for deduplication since this is how the storage server controls if certain data must be written to the storage device or not. However, it can also lead to information leakage, such as equality of information, so additional measures must be put into place to try to mitigate this issue. A possible approach is to perform deduplication in epochs, which protects from an adversary finding out if a chunk was already stored in a previous epoch, thus making it so that the server is only able to infer duplicates from the same epoch. Obviously, this scheme leads to lower storage savings, which can be mitigated if one leverages the concept of temporal locality, which states that a significant percentage of duplicate blocks are expected to be written in a small time-window. Although this property may not be present in all storage workloads, it is highly explored in the deduplication field to introduce caching optimizations and improve deduplication performance [46].

However, applying deduplication in epochs raises some issues for the traditional approach, since CE is not designed in a way that allows for the clients to synchronously change the epoch. Thus it is required to depend on an external key server, a secure proxy or employ trusted hardware on the storage server to control the change of epoch.

Dang and Chang [27] achieved secure deduplication between epochs by relying on the use of trusted hardware and enterprise proxies, which validated the idea of combining deduplication with hardware-assisted security technologies and paved a way for future

solutions (Section 2.2.2). However, there are still some drawbacks associated with the proposed approach. For instance, it assumes the existence of reliable proxies between clients, which is not possible in many use cases. In addition, the several levels of processing lead to performance overhead in the critical I/O path as well as requires additional resources (*e.g.*, additional servers).

Like Dang and Chang, TrustFS [32] also exploits temporal locality and only deduplicates within an epoch. However, TrustFS was developed using FUSE (*Filesystem in Userspace*), which leads to a relatively big overhead and was only tested using a synthetic benchmark, so it was not possible to get a perception of the real trade-offs of performing deduplication in epochs.

Evidently, performing deduplication in epochs does not protect from the server inferring duplicates from the same epoch, so additional measures must be put in place. Of the state of the art solutions that applied epochs, the approach proposed by Dang and Chang [27] to protect against leakage in the same epoch assumed an anonymous communication service in each enterprise service within their local networks. However, in their words, that is "an aggressive and unreasonable requirement". In addition, TrustFS did not tackle this issue. So other solutions must be considered.

Of the state of the art solutions, TED [40] stood out. This security scheme does not apply epochs but it offers some interesting security guarantees. Of these, it is worth noting that it is able to mask the real number of duplicates by basing the computation of a block's hash on the number of chunk copies. In addition, it also offers a way to control the trade-off between storage efficiency and data confidentially. However, although TED approach is efficient on not disclosing the frequency of duplicates per chunk, their estimated frequency counter can lead to the speculation that a block has a higher number of duplicates than in reality, which can lead to lower deduplication gains. Nevertheless, the idea of limiting the number of duplicates for a chunk can be applied to mitigate in-epoch leakage.

Thus, the goal of this thesis is to design a secure solution that solves the aforementioned problems while achieving low storage performance overhead and high deduplication gains. This is attained by further exploring Intel SGX functionalities and proposing a novel epoch-based secure deduplication scheme that is able to hide the real number of duplicates per block from malicious attackers.

3

S2DEDUP ARCHITECTURE

This chapter details S2Dedup design and architecture, as well as, the different secure deduplication schemes supported by our solution.

Our architecture assumes a server that deals with several clients and is equipped with trusted hardware, which from its perspective is like a black box after the code is loaded. S2Dedup uses this as an advantage to implement a secure deduplication scheme that handles sensitive information on the server-side that is protected from a potentially unreliable server or other clients.

Threat Model. Our design assumes that the client application is running on trusted premises. On the other hand, the communication channels and server are not trusted and susceptible to honest-but-curious attackers [49].

Namely, an adversary might take control over the operating system or other software deployed at the storage server, but, it is unable to compromise trusted hardware (SGX). Moreover, since deduplication related metadata is maintained outside of a trusted environment, we take into consideration that an adversary might exploit this information to infer the number of duplicates per block and check what blocks are duplicated or not. It would be possible to prevent the latter attack, namely for an attacker to know if a block has duplicates or not, by using Oblivious RAM [56], and designing our deduplication engine to behave in constant time to prevent side-channel leakage. However, relying on Oblivious RAM would take a considerable toll in the performance of our storage system [52, 63], so we prioritize feasibility, leaving the support for these two types of attack for future work.

Finally the communication channels between the clients and server can be eavesdropped but, since we are considering honest-but-curious attackers, the content of data sent through these channels cannot be tempered with.

3.1 GENERAL ARCHITECTURE

The main goal of this dissertation is to develop a secure deduplication system that takes advantage of the functionalities provided by trusted hardware. To this end, we developed

a general system architecture for S2Dedup, depicted in Figure 4, that demonstrates how trusted hardware can be integrated with the deduplication workflow.

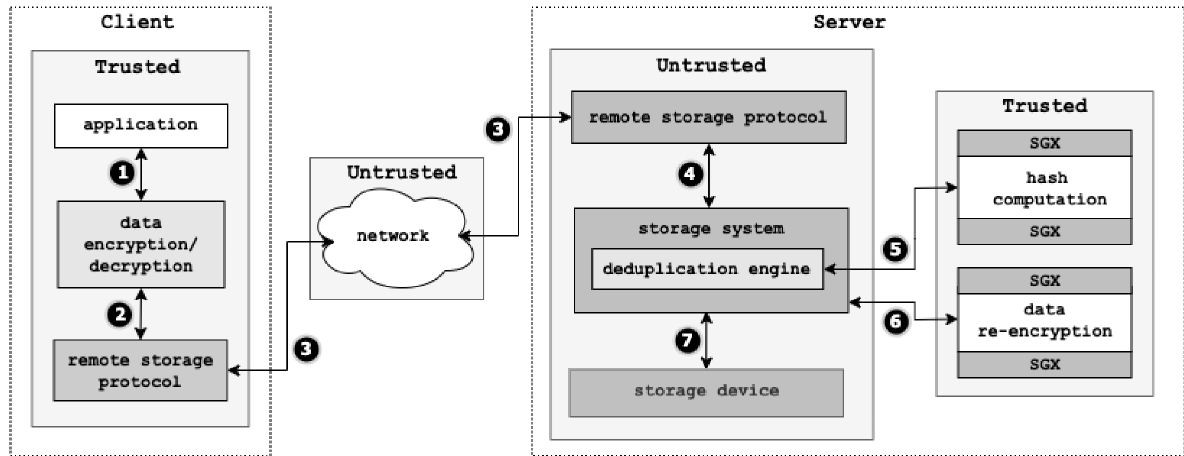


Figure 4: System architecture overview.

The design follows the classic client-storage server configuration, where a client sends data to a third-party remote server to be stored persistently.

Concerning the deduplication component, there are three main design choices to take into account, which are the granularity, timing and location. When it comes to granularity and timing, our solution works at the block-level and follows an inline approach as these can achieve higher deduplication gains. Also, deduplication is done at the server-side since any approach that changes the client-side behaviour towards deduplication will reveal duplicate information [17].

S2Dedup design is compliant with the architecture of hardware-assisted security technologies similar to Intel SGX. This choice was done due to SGX's availability in commodity hardware and popularity in academia [26, 34, 37, 53, 64] and industry [2, 8]. The goal of the trusted hardware module is to ensure that computations done by the storage system over plaintext are conducted in a privacy-preserving setup as well as to store critical data, such as encryption keys.

3.1.1 *Flow of Requests*

When a client wants to send their data to a remote storage server, it begins the process by encrypting it with its own key using a standard probabilistic encryption scheme (Figure 4-❶), to protect against attackers targeting the network channels and the untrusted remote server.

Then, the encrypted data is sent over the network through a remote storage protocol (e.g. iSCSI) (Figure 4-❷❸). Once it reaches the storage server, the client's encrypted data is forwarded to a storage system (Figure 4-❹) to be processed. This component integrates a deduplication engine that analyzes the data, block by block, in order to detect duplicates. To accomplish this, the deduplication engine must calculate a hash digest for each block (Figure 4-❺) and compare it with the hashes of previously stored blocks.

Note that when data is encrypted at the client level, identical blocks will result in different ciphertexts since it is used a probabilistic encryption scheme and each client has a different encryption key. Therefore, the hash calculation needs to be done over the plaintext message in order to find duplicate blocks. Accessing the data in its original form outside of a secure execution environment would reveal sensitive information to an attacker at the untrusted server, thus the hash calculation requires the use of SGX. Moreover, for one to able to decrypt the client's data it is necessary to use their key, thus is assumed that S2Dedup has brief bootstrap stage, where a secure channel is initialized between the client and the remote enclave. This secure channel is used to exchange critical data between the client and the enclave, such as the symmetric key used by the client to encrypt their data. Instrumenting enclave code in this way is a common requirement for these systems, and has been shown to be achievable with minimal performance overhead [18, 53]. The establishment of a secure channel and the exchange of keys is not the focus of this dissertation, but it would be possible resorting to existing protocols, such as the ones presented by Bahmani et al. [18] and Machida et al. [42]. The enclave can be seen as an extension of the client on the server, which allows us to do these processing.

If a duplicate block is found, the redundant content does not need to be written to the storage medium. If the block is unique, then the block's content is re-encrypted using a universal *encryption key*, that is only accessible by the enclave (Figure 4-❻), and then forwarded to the storage device (Figure 4-❼) . The re-encryption process is again accomplished with the aid of SGX since it requires decrypting the client's encrypted data and then encrypting it with the universal *encryption key*. Since plaintext data will be disclosed temporarily, this needs to be done at a secure enclave. This re-encryption process is required because the storage server is used by multiple clients, thus the data needs to be stored in a uniform manner so that when it is necessary to access it, it is not necessary to control which of the client's keys was used to encrypt each of the blocks.

Re-encryption and hash calculation steps are done in two separate enclave invocations. The alternative approach would be to do both in a single enclave call, while always outputting to the untrusted environment the corresponding hash and encrypted block. In a deduplication ecosystem, one expects to find a high percentage of duplicates, therefore most write operations will not be persisted into the server's storage medium. Being this the case, doing the re-encryption step for duplicate blocks that will not be stored is unneces-

sarily increasing the processing done at the enclave, as well as, the data being transferred between the enclave and untrusted environment. Thus, our approach only performs data re-encryption when a block is found to have unique content.

Moreover, due to performance reasons, our design ensures that only a block (e.g., 4kB of data) is sent to the enclave at each time. This is important for trusted hardware solutions, such as Intel SGX, as several studies show that transferring large amounts of data (*i.e.*, in the order of few MiB or more) to enclaves has a negative performance impact [33, 57, 58].

When a client wants to retrieve data from the remote server, the request follows a similar process. Once it reaches the storage system 4-❹), the deduplication engine is consulted, block by block, in order to figure out where in the storage device is stored the requested data. When it obtains the addresses, the data is read from storage (Figure 4-❼), decrypted with the *encryption key* and then re-encrypted with the corresponding client's key (Figure 4-❻). The latter two steps are done inside a secure enclave. Afterwards, the encrypted data is sent over the network to the client (Figure 4-❹❸), where is then decrypted and read by the client (Figure 4-❷❶).

3.2 DEDUPLICATION ENGINE

As one can infer from Section 3.1, one of the key elements of the proposed system is the deduplication engine. This engine is integrated in the storage system by a simple interface, of which is worth pointing out the `read_block` and `write_block` operations (Listing 3.1).

```
uint64_t read_block(uint64_t laddr);
uint64_t write_block(uint64_t laddr, char *content);
```

Listing 3.1: Extract from the deduplication engine interface.

In a system that performs deduplication the address that the client perceives to store a certain block, also known as logical address (`laddr`), usually does not correspond to its physical address (`paddr`), that is, the address in the storage device that the block is actually stored at. This happens because when a block already exists in disk is not written again and, when is new, the address that is stored at is controlled by the deduplication engine. Therefore, functions like the `read_block` and `write_block` are required.

The `read_block` operation is used by the storage system to figure out where to read in disk a certain block. It receives as an input the logical address (`laddr`) of where the client is trying to read and returns the physical address (`paddr`) of where that block is actually stored.

Likewise, the `write_block` operation is called by the storage system to figure out if a block is new and if that is the case, where to write it in the disk. It receives the block's

laddr and content to detect for duplicates and returns, in the case of a new block, the paddr at where the operation should be performed.

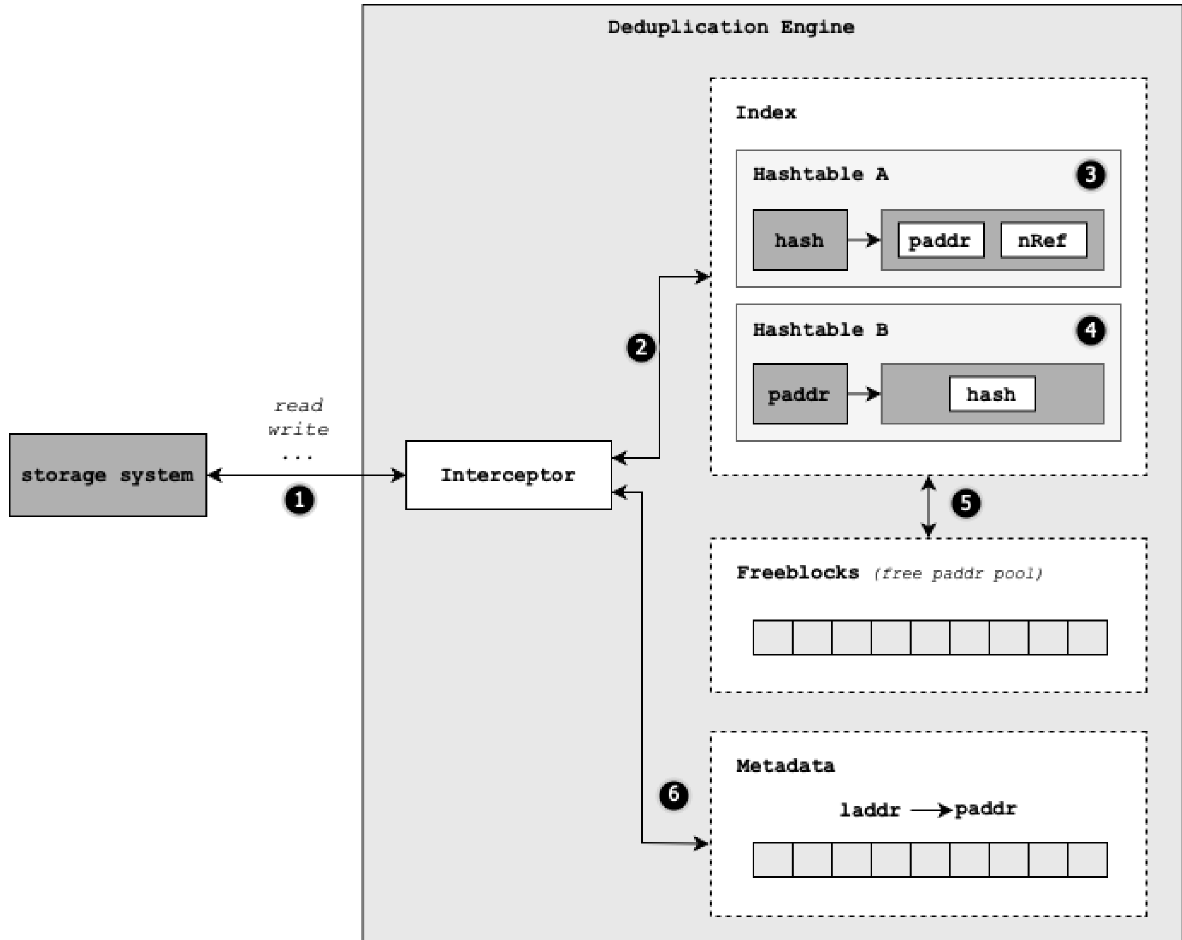


Figure 5: Deduplication system architecture.

Since S2Dedup is following an inline deduplication approach, this engine is composed by four modules (Figure 5) - interceptor, index, freeblocks, metadata - each one playing a critical role in the correct execution of the deduplication algorithm:

- interceptor - receives external requests, such as read_block and write_block operations, and maps the logical address (laddr) to the corresponding physical address (paddr) at which the operation should be executed.
- index - this module is responsible for keeping track of a block's physical address (paddr) and number of references (number of duplicates) (nRef). Since using a block's content as a key for search purposes would be inefficient, its corresponding hash sum is used instead.

- `metadata` - manages the mapping of a logical address to the physical address. Essentially maintains the correspondence between the address that a client perceives to store a certain block and the address in the storage device that is actually stored at. Therefore, when we want to access a certain address, it is first required to consult this module in order to determine where to read from in the storage device.

- `freeblocks` - is a pool of free physical addresses that are available to be written to. A physical address can become available when the data is replaced or deleted and no `laddr` maps to the content that the physical address is storing. Thus it is not necessary to preserve it and that physical address is considered free for new content to be written to. In order words, a physical address is available when its number of duplicates, that is, the correspondent `nRef` value maintained by the `index` module, reaches zero.

Considering a scenario that a client wants to read some data from the storage device. When the storage system receives that request, it invokes the `read_block` operation (Figure 5-❶), provided by the `interceptor` module, for each of the blocks that is trying to read.

The `interceptor` based on the logical address (`laddr`) that it receives as an input, consults the `metadata` module to find its correspondent physical address (`paddr`) (Figure 5-❻), that is, the address in the storage device that the block's content should be read from. Once the `interceptor` obtains the `paddr`, returns it back to the storage system (Figure 5-❶).

In the case of a *write* request, the storage system calls the `write_block` method in the `interceptor` module for each of the blocks that is trying to write (Figure 5-❶). The `interceptor` receives as an input not only the block's `laddr`, but also its `content`, in order to be able to detect duplicates. However, directly using its content as a key to compare to previously stored blocks would be inefficient, thus the `interceptor` begins its operation by calculating its corresponding `hash`. As explained in 3.1.1, this is accomplished with the aid of SGX, because the hash calculation needs to be done over the plaintext message and handling it outside of a secure execution environment would reveal sensitive information to a possible attacker.

Once the hash is calculated, the `index` is consulted (Figure 5-❷), to check if the block is a duplicate and determine the physical address that is stored. In the case of a new block, an available address to write to is return instead. Thereby, the `index` receives the computed block's `hash` and tries to determine if it is a duplicated block by consulting `hashtable A` (Figure 5-❸). If it already exists, the number of references to that block is incremented, otherwise, a free physical address is requested from the `freeblocks` module (Figure 5-❺) and a new entry added to the `hashtable A` and `B` (Figure 5-❸❹).

Afterwards, based on the `paddr` returned from the `index` and the `laddr` received as an input in the `write_block` operation, the `interceptor` invokes a `put` operation to the `metadata` component (Figure 5-❻) . This module updates the physical address that

is mapped by the laddr that it received, so that when it is required to read from that laddr we know the current address in the storage device that the block's content should be read from. However, when this put operation is performed, it means that there is one less duplicate copy for the previous paddr that was mapped by that laddr. Thus the index module, more specifically, the nRef in the hashtable A for that paddr, should be updated.

Thereby, once the metadata completed its operation, the interceptor sends the previous paddr to the index module to update its nRef (Figure 5-❷). The requirement for the deduplication engine to be able to, based on a paddr, update its current number of duplicate copies, resulted in the need for the hashtable B. This hashtable B manages what content (hash) is stored at a certain paddr (Figure 5-❹) and, based on this, we can find its correspondent nRef in the hashtable A (Figure 5-❸) and, consequently, decrement the current number of duplicate copies for that paddr. If the nRef value happens to reach zero, this means that no laddr maps to that paddr, thus that paddr is free to store new content. Consequently, it is then added to the freeblocks pool and the respective entries in the hashtable A and B are deleted.

When all of this is completed, the interceptor can finally inform the storage system if the block is new or a duplicate and, in the case of a new block, where to write it in the storage device (Figure 5-❶).

3.3 SECURITY SOLUTIONS

Applying deduplication over encrypted data raises new challenges, given that a user cannot simply encrypt its data using their private key, since the same original data, owned by different users, may result in different ciphertexts, which makes it impossible to find duplicates. In fact, if a probabilistic encryption scheme is being used, even identical blocks from the same user will be encrypted to ciphertexts with different content.

Convergent encryption (CE) solves this problem by encrypting data with a key that is derived from the block's content. This way, identical plaintexts will generate the same ciphertexts, which makes it possible to perform deduplication over encrypted data. However, this technique is susceptible to numerous attacks, like when an adversary tries to deduce whether a file was previously sent to a storage system or when an adversary tries to figure out the original content of a file, by performing a brute-force attack, where repeatedly tries every possible combination until finding one that matches. Moreover, CE also requires the client to store the convergent keys generated, so it can later access the data that was encrypted using this approach.

Next, we detail three secure deduplication schemes, supported by S2Dedup, that offer different security and privacy guarantees.

3.3.1 *Enclave Based Scheme*

S2Dedup relies on hardware-assisted security technologies that grant access to a secure environment to perform deduplication and even store sensitive information, such as encryption keys or deduplication related metadata.

The first security scheme, named "Enclave Based", assumes that clients encrypt their data at trusted premises, with a probabilistic encryption scheme and a private key, before sending it through the network. Once the encrypted data reaches the storage server, it is partitioned and sent to the interceptor module, where it is processed within the SGX protected environment, block by block. This enclave stores the client's symmetric key, which allows for the system to securely decipher the encrypted data and process it in plaintext. Namely the block's hash sum is calculated over the plaintext, inside the enclave, using a specific *hash key* used only to compute a block's hash, that is also stored in the enclave.

This hash is used by the deduplication engine, which operates outside of the enclave, to verify at the index if this is an unique or duplicate block. In case that it is unique, the data is encrypted, again within the enclave, using the *encryption key*, which is kept in the enclave as well. This key is different from the *hash key* not only because it offers more security guarantees, but also because it makes this scheme adaptable to more robust security techniques, which are discussed in Section 3.3.2 and Section 3.3.3.

It is noteworthy that outside of the enclave it is only handled the client's data encrypted by their own key, the generated block's hash and the client's data encrypted by the enclave *encryption key*, thus at no point, the client's original data exists outside of a secure environment.

In the case of a *read* operation, it follows a similar process. The client sends the request to the storage system and this consults the deduplication engine, block by block, in order to figure out where to read the data from in the storage device. Once it obtains the addresses, the storage system reads the data from the disk. However, this data was previously encrypted using the server's *encryption key*, thus for the client to be able to read, it must be re-encrypted with the client's own key. Since this operation requires the data to be handle in its original form, this re-encryption process is also performed within the enclave. Once all the data is read and encrypted with the client's key, it is sent over the network to be then decrypted and read by the client.

S2Dedup by relying on hardware-assisted security technologies allows for the use of a probabilistic encryption scheme to encrypt the client's data, instead of a deterministic one proposed by the convergent encryption approach. This is possible because our solution

offers a secure environment in the server for the data to be handled securely in its plaintext form. Moreover, our solution also avoids the need for the client to store in a resilient fashion the convergent keys generated.

3.3.2 *Epoch Based Scheme*

The "Enclave Based" scheme can be improved in terms of security guarantees by trading off some of the achievable deduplication space savings.

Although "Enclave Based" scheme manages to perform secure deduplication to some degree, it still suffers from some of the same security issues as the convergent encryption approach, meaning that an attacker is still able to detect if a block is duplicate. For example, an adversary can send a file to the storage server and verify how the deduplication related metadata is affected since it is stored outside of a secure environment.

To protect against this type of attack, we developed an alternative scheme named "Epoch based" supported by the work of Dang and Chang [27] and Esteves et al. [32]. In S2Dedup we leverage the concept of temporal locality, that states that a significant percentage of duplicate blocks are expected to be written in a small time-window [46], to increase the provided security guarantees, whilst still ensuring its deduplication effectiveness.

In more detail, the proposed scheme operates in epochs, that change according to a pre-defined number of operations or time period. In each epoch, a new *hash key* is generated, therefore making it impossible to detect duplicates from different epochs. Namely, identical data processed at the same epoch will produce the same hash and will be deduplicated. However, if identical data is processed at a different epoch, it will not be deduplicated since different keys were used to calculate their hashes, thus from an outsider perspective, it will be as if the deduplication system is processing a block with distinct content. Since the *hash key* changes for each of the epochs, we cannot use the same key to encrypt the data (*encryption key*) as the one to compute its hash (*hash key*), since to be able to decrypt the data after that epoch ends, it would require the control of which key was used to encrypt each block, which introduces unnecessary complexity to a problem that can be easily solved by simply using different keys.

If storage workloads exhibit strong temporal locality, most duplicates will be written at the same epoch, thus not affecting significantly the achievable deduplication gain. In Chapter 5, we show that indeeed this is valid for different realistic workloads. Nevertheless, the duration of an epoch can have a big influence on the deduplication gains and level of security. A smaller epoch offers higher security but also leads to fewer duplicates being detected, while a larger epoch allows finding more duplicates at the cost of leaking more information about the number of duplicates of a given workload.

In conclusion, with this scheme, an attacker will not be able to infer the number of duplicates per block across different epochs, whilst still enabling deduplication space savings. However, it does not protect from the server inferring duplicates from the same epoch.

3.3.3 *Epoch and Exact Frequency Based Scheme*

Li et al. [40] proposed a solution that determines a block's hash sum not only on its content but also on the current number of blocks that were found to have the same content. This allows controlling the maximum number of duplicate copies for a given chunk, parameter that was referred to as *t* in Li et al. work. The smaller the *t* parameter, the higher the security offered, but it comes with the trade-off of a lower deduplication effectiveness. Evidently, the higher this parameter, the opposite behaviour occurs.

This solution bases the frequency counter on a *Count-Min Sketch* algorithm, which only provides approximate counters in order to reduce the memory footprint for the metadata where these counters are stored. This algorithm is known to lead to inaccurate counters that overestimate blocks frequency. In some cases, such behaviour results in conjecturing that a block has a higher number of duplicates than it has in reality and, therefore, limits its number of copies, which leads to a lower deduplication effectiveness. For example, imagining a scenario that defined *t* as 5 and the counter indicates that a certain chunk had already 5 duplicates, when in reality only had 3. If that chunk were to be written again it would be treated as if the system is dealing with a new chunk, thus will not be deduplicated and, consequently, reducing the deduplication effectiveness. However, this approach comes with an advantage, the way that is implemented protects the deduplication system from disclosing the frequency of duplicates per chunk to an attacker.

With the secure environment provided by trusted hardware, an alternative scheme can be devised. Instead of using an estimated counter, we can base our calculations on an exact counter, that is, a counter that accurately provides how many duplicate copies for a block were previously stored. This way, we can ensure that the expected deduplication ratio is obtained. Revisiting the example mentioned above, if *t* is 5 and the system already handled 3 copies of a chunk, the exact counter will indicate correctly 3 copies. Thus if the same chunk is written again the maximum value still was not reached ($3<5$), so that chunk will be deduplicated.

Furthermore, by using SGX enclave to store this counter, the security guarantees offered by an estimated counter are no longer relevant, since an attacker will not be able to access this counter.

However, since the secure memory space offered by SGX is limited, when all the memory is used by the exact counter, it is necessary to clear it and calculate a new *hash key*. This is similar to the "Epoch based" setup, but now the epoch is not limited to the number of *write* operations performed, but by the enclave's memory.

Based on this, one might perceive that with an exact counter the deduplication gain is more affected than with an estimated counter, but that is not the case. It is necessary to take into account the temporal locality, that an epoch only changes after many operations and that the t parameter, based on TED, is expected to be relatively small, therefore making almost imperceptible the change of epoch in the deduplication ratio. Moreover, the integration of epochs into this solution adds the same security guarantees that the "Epoch based" scheme, therefore resulting in an extra layer of security, since an attacker is also not able to detect duplicates from different epochs.

In conclusion, with this scheme, we are able to mask the number of duplicates per block across different epochs and within each of the epochs, as well.

4

PROTOTYPE

We now describe the S2Dedup prototype that is evaluated and validated in Chapter 5. In this chapter we begin by discussing how the server components were implemented, namely, the frameworks, libraries, data structures and tools used (Section 4.1) and, afterwards, in Section 4.2, we present how the client component was developed.

4.1 SERVER IMPLEMENTATION

As previously mentioned, this project's main goal is to create a system that performs secure deduplication without neglecting its performance and efficiency in saving storage space. It is also crucial that it be up to date with today's progress in storage technologies. This way, S2Dedup implementation uses the *Storage Performance Development Kit* (SPDK) [12, 13].

Nowadays, with fast storage devices (e.g. NVMe SSDs) becoming increasingly popular, it is possible to write applications that take better advantage of the speed and resources of these storage disks. This can be done with SPDK since it provides a set of tools and libraries for writing high performance, scalable, user-mode storage applications. SPDK-based applications are able to achieve high performance by eliminating the kernel I/O stack overhead and alternatively moving all the necessary drivers into userspace and operating in a polled mode instead of interrupted mode, which avoids kernel context switches, eliminates interrupt handling overhead and also provides lockless resource access. According to Yang et al., an SPDK NVMe device driver can reach 6X to 10X better performance per cpu core (IOPS (*Input/Output Operations Per Second*)/core) than the kernel space NVMe driver.

By taking advantage of the tools provided by SPDK, S2Dedup secure deduplication engine is integrated in the form of a virtual block device, as shown in Figure 6. SPDK enables the implementation of virtual block devices, with user-defined logic, that follow a standard block-device interface. Therefore, by following the same interfaces, these virtual devices can be stacked and their functionalities can be aggregated to enable richer and more flexible user-space storage solutions. In this work, the deduplication virtual block device intercepts

incoming I/O requests, performs secure deduplication and only then sends them to the NVMe block device or to another virtual processing layer, depending on the targeted SPDK deployment. These requests will eventually reach the NVMe driver and storage device, unless intermediate processing eliminates this need, as it can happen for duplicate *writes* when using deduplication. Moreover, SPDK also provides a set of storage protocols that can be stacked over the block device abstraction layer. Of these, our work uses iSCSI (*Internet Small Computer System Interface*) to enable our clients to use the storage server from a remote machine.

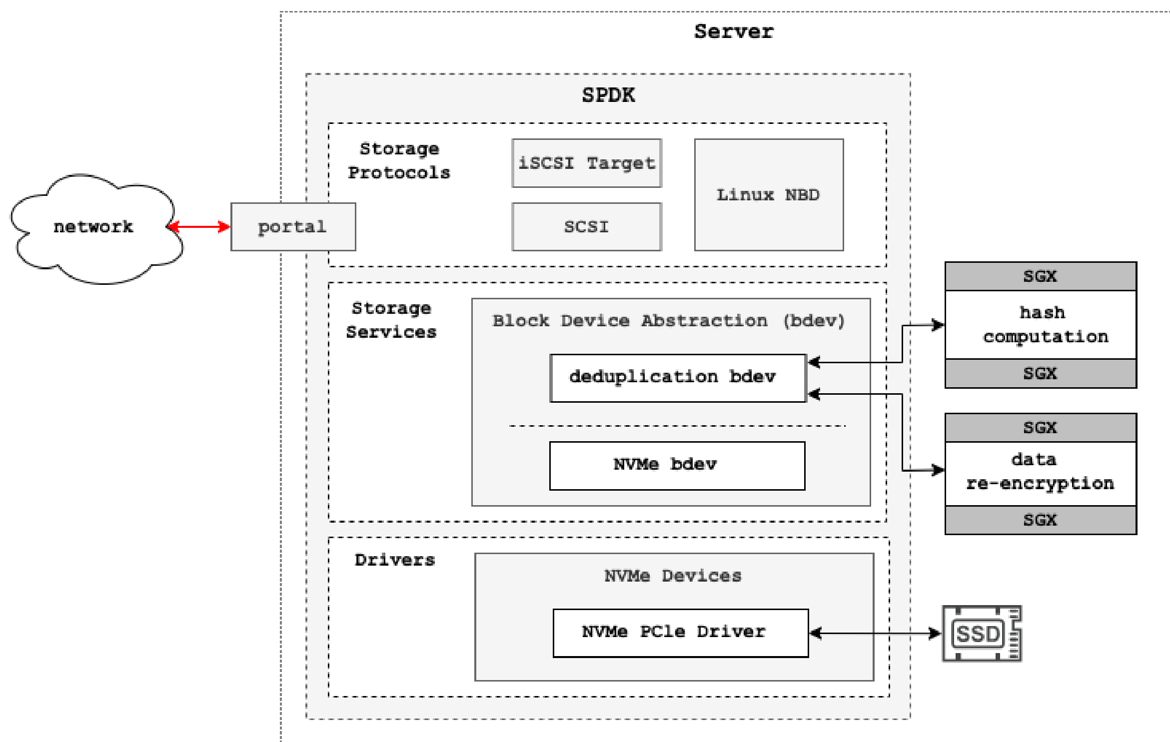


Figure 6: Server implementation.Server implementation.

4.1.1 *Deduplication engine*

Two alternatives were developed for the implementation of the deduplication engine, one that stores most of the deduplication related metadata in-memory and other that does so persistently. By comparing the first and latter alternatives one can observe the overhead of providing fault-tolerance support.

As described in 3.2 the deduplication engine is composed of four elements, the `interceptor`, `index`, `metadata` and `freeblocks`. The `interceptor` acts as a coordinator between the three other components, thus no data or metadata is required to be stored.

The `index` and `metadata` for the in-memory alternative were implemented by using the GLib [7] library to provide the two hashtables of the `index` and a simple array to manage the `metadata` module.

Regarding the persistent solution, each component was implemented using the on-disk key-value store LevelDB [1]. However, this presented itself as a challenge, given that when SPDK connects to the NVMe disk, it firsts unbinds the kernel driver from the device and then rebinds the driver to a "dummy" driver so that the operating system won't automatically try to re-bind the default driver. This makes the device no longer accessible outside of the SPDK environment, meaning that it is not possible to simply partition a disk and dedicate a part to run SPDK over it and the other to store the deduplication related metadata. Nevertheless, SPDK allows for the creation of partitions within its environment, so it was possible to mount the deduplication virtual block device over part of the disk and run a Linux NBD (*Network Block Device*) over the other, which is also provided by the SPDK, allowing for the data to be stored in the same disk as the deduplication related metadata.

As an alternative, one could simply use another disk to store the deduplication related metadata, however, we wanted a solution that did not require more than one disk and benefited from the fast storage of NVMe disks.

Concerning the `freeblocks` module, it was found that the best solution, for both alternatives, was to store the free addresses through the SPDK toolkit, combined with a *cache* system. Namely, in the same way that blocks of data are read and written to disk, we used blocks of data to store freeblocks addresses. For example, if a disk block has 4096 bytes and a freeblock address has 8 bytes, we can store 512 addresses in that block. So, when a new address is required, our solution, for a matter of efficiency, reads several blocks at once and stores its addresses in a memory *cache*, to be used in the next operations. When an address is no longer required, it is maintained in a different *cache* until a certain number of block addresses are released, and then, are flushed into the NVMe disk.

4.1.2 *Enclave Based Scheme*

The main reason why S2Dedup is able to perform secure deduplication is due to the fact that critical operations, like the block's hash computation and re-encryption of data, occur within the SGX enclaves [11, 25]. As explained in Section 2.1.1, an untrusted application can execute a set of functions inside of the enclave that were previously defined in a EDL

(*Enclave Description Language*) file. Therefore, ours prototype EDL file can be defined as follows:

```
enclave {
    trusted {
        public int trusted_compute_hash(...);
        public int trusted_reencrypt(...);
        public int trusted_reencrypt_reverse(...);
        ...
    };

    untrusted {
       ...
    };
};
```

Listing 4.1: Extract from the EDL file.

The `trusted_computed_hash` function allows for the computation of a block's hash, while `trusted_reencrypt` and `trusted_reencrypt_reverse` enables an application to re-encrypt incoming and outgoing data, respectively.

For the computation of a block's hash, it is generated an HMAC with SHA256 as the keyed cryptographic hash function, using the *hash key* as the key. This *hash key* is created when the SGX is initiated and maintained in the enclave.

When it comes to the re-encryption of data, for both client's and server's encryption schemes, we use the AES-XTS block cipher mode, which is standardized by the IEEE [15] and NIST [31]. It is the ideal solution for this deduplication system because it is length-preserving, thus the length of the ciphertext is the same as of the plaintext and does not apply chaining, supporting random access to the encrypted data, thereby not requiring reading several data units to decrypt a single unit.

It is noteworthy that in the case of a system reset or failure, one is still able to decrypt the data once the system is restarted since the *encryption key* used to encrypt incoming data is stored on disk, through the sealing mechanism provided by SGX, which is only accessible by the enclave.

Also, as explained in Section 3.1.1, our current prototype assumes that, during the application initialization, it was established a secure channel between the client's application and the server's SGX enclave, so that the client can securely exchange its key with the enclave. Although this functionality is not implemented, Bahmani et al. [18] and Machida et al. [42] demonstrated in previous work that is possible. For the purpose of evaluating and validating the prototype, the client's symmetric keys currently are stored statically in the enclave.

4.1.3 *Epoch Based Scheme*

This scheme's implementation is very similar to the "Enclave Based" scheme, but it differs by the introduction of the concept of epochs, which requires the *hash key* to be updated at every epoch.

As described in Section 3.3.2, an epoch is changed when a certain time has passed or a limit of operations is reached. We chose to define this limit by the number of *write* operations that the system has processed, thus when a certain number is reached, the current epoch ends and a new one is introduced. To achieve this, our solution simply maintains a counter of how many *write* operations were conducted until the moment and, when the limit is reached, this counter is cleared. Next, the function `sgx_read_rand`, provided by SGX, is used to generate a new key.

4.1.4 *Epoch and Exact Frequency Based Scheme*

The "Epoch and Exact Frequency Based" alternative is also based on the same foundations as the "Enclave Based" scheme, however, it differs in how a block's hash is computed.

As explained in Section 3.3.3 this solution bases its block's hash computation on not only its content but also on its current number of duplicate copies and a maximum number of allowed duplicate copies per chunk (t).

To accomplish this, it is first generated a simple hash from the block's content, that does not consider the number of copies previously found. Then, it is consulted a hashtable that returns the number of duplicates for that simple hash. This hashtable is maintained in the enclave so that it is protected from malicious attackers.

Once known the number of copies (n), that number is divided by t and the obtained number (n/t truncated towards zero) is used to calculate the new hash for the block. By dividing n by t, we guarantee that in intervals of t duplicate copies for a block, it will generate different hashes, thus from an outsider perspective, it will be as if we are dealing with a new block.

Figure 7 shows an example of this scheme. Let us consider a scenario where S2Dedup is configured with t as 15 and already found 13 copies for the block with the hash `hash1`. Then, another duplicate block is written with the same hash, which results in 14 duplicates for that chunk. Dividing 14 by t results in zero, which leads to `hash1a` like the previous *write* operations for that block, since, until 15 copies are found, every number when divided by 15 returns zero.

Later, the same block is written again, but this time the hashtable returns 15, which divided by t results in one. This leads to a different hash from `hash1a`, which is represented as `hash1b`, thus, for an attacker observing the content outside of the enclave, this hash will be inferred as a new block.

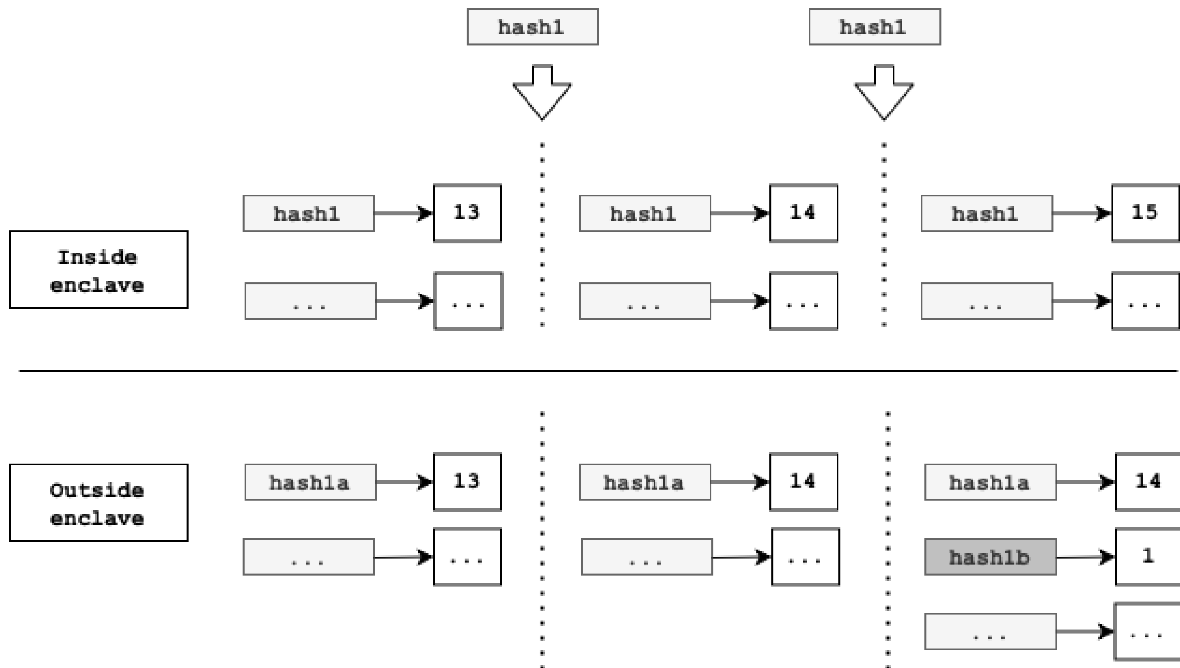


Figure 7: Example of the "Epoch and Exact Frequency Based" setup for t=15.

Since this hashtable is maintained in the enclave, which has limited available memory, it is necessary to clear the hashtable and calculate a new *hash key* when the memory limit is reached. We could not simply continue using the same key, because it could lead to a block having more than t duplicates.

4.1.5 *Estimated Frequency Based Scheme*

To be able to test our exact frequency approach against the estimated frequency one proposed by Li et al., we also implemented a setup that is based on an estimated frequency counter. This scheme is implemented in the same way as the "Epoch and Exact Frequency Based", but instead of using a hashtable as a counter, it is used the *Count-Min Sketch* algorithm as a counter. Moreover, since this algorithm uses a fixed memory size, it is not required to change of epoch once all the memory is used. Therefore the implementation of this scheme, and as proposed in the original work, does not resort to epochs.

To implement this algorithm, it was used a matrix with 4 rows and 2^{20} counters per row. These values were based on the ones used in TED, which were 4 rows and 2^{25} counters per row. It was decided to use 2^{20} counters per row, instead of 2^{25}, because it was found that for 2^{20} counters the deduplication effectiveness was not affected by the number of counters, and, since this matrix is maintained in the enclave, which has limited memory, it was unnecessary to spend important resources.

4.2 CLIENT IMPLEMENTATION

The client and the server communicate through the iSCSI protocol. However, since security is one of the main concerns, a client cannot just send data over a network to a remote server without first encrypting it, thus it is also necessary a mechanism that enables clients to transparently secure their data.

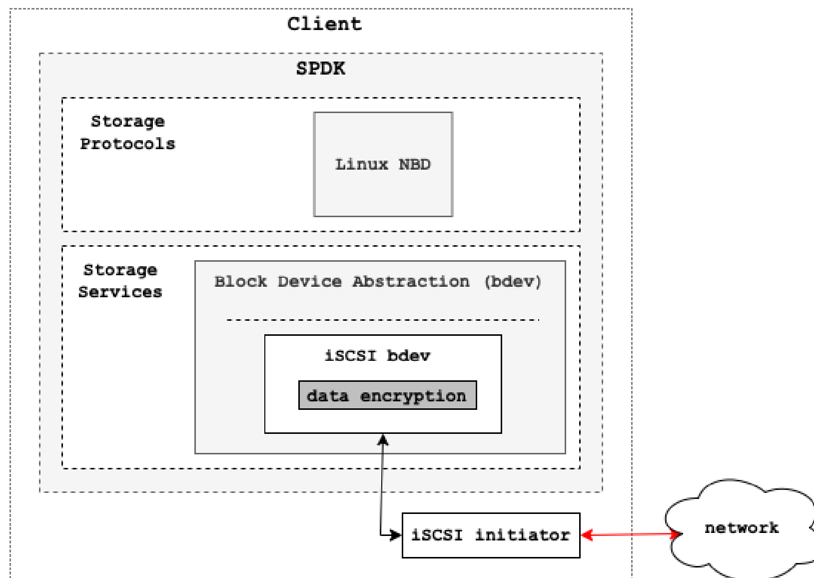


Figure 8: Client implementation.

To equip the client's side with transparent encryption, several alternatives were explored, but it was found that the straightforward solution was to operate on the data using the functionalities offered by SPDK (Figure 8). Namely, SPDK features an iSCSI client virtual block device that integrates the library *libiscsi* [9] into its configuration, which makes it possible to connect to a remote storage system by iSCSI. Based on this, one can easily integrate transparent encryption to a request that is sent through an iSCSI protocol. Again, the AES-XTS block cipher mode was used to encrypt client's data. Moreover, by stacking an Linux NBD over the iSCSI virtual block device, our solution can be used by any application relying on a standard block-device interface.

5 EVALUATION

Evaluation plays a pivotal role in the development of any solution since it allows us to critically examine and better understand what impact different settings have on a program.

Considering that our system is based on two main features, deduplication and security, we intend with our evaluation to examine:

- What is the impact of deduplication on applications performance?
- What are the trade-offs in terms of deduplication throughput, latency and resources (*i.e.*, RAM, CPU, network) usage, when applying different levels of security?
- How do different levels of security affect deduplication gains?

In this chapter, we detail both how the evaluation of our system was conducted as well as the obtained results. We begin by introducing which methodology (Section 5.1), workloads (Section 5.2) and setups (Section 5.3) were used. Then, in the Sections 5.4 and 5.5 we present the tests and the obtained results and, lastly, we discuss them in Section 5.6.

5.1 METHODOLOGY

All conducted experiments were based on the same general principles, these being that 4kB blocks (the most commonly used size in deduplication systems) of data were read or written for a certain amount of time, or, in some cases, until it totalled a specific volume of data. To guarantee the accuracy of the results, each test was repeated at least three times, making sure to collect the latency and throughput at the end of its run and, when appropriate, the deduplication gain as well.

Additionally, it was used *dstat* [59], a commonly used tool for generating Linux system resource statistics, which allowed us to collect relevant metrics, like the CPU, RAM and network usage. Moreover, to ensure the consistency between the different runs, all tests needed to be performed under the same conditions. Hence, we made sure to clean the page cache and the deduplication system prototype between each run and to, before any *read* operation testing, first populate the system with the same data.

Two types of tests were conducted, first the ones we labelled as environment testing, which had the goal of evaluating the experimental environment and each component that is independent from our implementation. The second type was prototype testing and was used to assess our prototype and its different alternative setups.

The servers used at the evaluation, that whether played the role of client or server, had the following specifications: a hexa-core 3.00 GHz CPU (Intel Core i5-9500), 16 GB DDR4 RAM, a 250GB Samsung SSD 970 EVO Plus and were connected by a 10Gb/s network link. Both servers ran Ubuntu Server 18.04.4 LTS with Linux kernel version 4.15.0-99-generic. Also, it was used the release of SPDK v20.04, and the 2.10.100.2 version of SGX SDK and Intel SGX Platform Software (PSW).

5.2 WORKLOADS

Both synthetic and realistic workloads were used at the evaluation. The first type of workload was provided by DEDISbench (commit #0956b9d [10, 48]), which was previously used in several scientific publications [47, 60, 62, 65]. DEDISbench is a disk I/O block-based benchmarking tool for deduplication systems that generates a synthetic workload that follows a realistic content distribution. This allows us to test our system with basic operations like *reads* or *writes* and control what access pattern is followed, which can be whether *sequential*, *uniform* or *zipfian*. The latter access pattern simulates a scenario where a small group of blocks is more accessed than the remaining blocks. Moreover, we can also choose the number of concurrent processes and the content distribution to be generated. We chose the distributions *dist_highperf* and *dist_kernels*, that have respectively 25% and 72% duplicated blocks, so that it is possible to assess the system with workloads exhibiting distinct levels of redundancy.

The second workload consisted of three real traces [6], that also have been used previously to evaluate other systems [38, 47]. These traces were collected for a duration of three weeks from three production systems at the Florida International University (FIU) Computer Science department and have different I/O workloads that consist of a virtual machine running two web-servers (*web-vm* workload), an email server (*mail* workload), and a file server (*homes* workload). Since these traces were collect over multiple weeks, for a matter of efficiency, the traces were replayed at different speedups to show how S2Dedup behaves under different I/O stressing conditions.

5.3 SETUPS

In order to isolate the performance of the different components and secure schemes used by S2Dedup prototype the following setups were considered at the evaluation:

1º - Local disk (NVMe). In this setup, the workloads executed I/O operations directly to a local NVMe disk in a single server.

2º - Local SPDK (NBD + SPDK + NVMe). This setup was established to assess SPDK's impact on the workloads performance. SPDK was configure to export a local NBD interface to the workloads while storage requests were forwarded, by the SPDK, to the NVMe disk. Note that this is also a local setup as the workloads ran in the same server as the SPDK storage stack.

3º - Server-based SPDK (iSCSI + SPDK + NVMe). This setup was designed to assess the network overhead of a remote storage protocol. The server configuration was similar to the previous one, however, SPDK exported an iSCSI interface to remote clients. Therefore, the local NBD interface was not used at the server. Additionaly, the workloads ran in another server (client machine) and used the standard iSCSI library, included in Linux, to communicate with the iSCSI server.

4º - Client and server-based SPDK (NBD + SPDK + iSCSI + SPDK + NVMe). This setup maintained the same server configurations of the previous one. At the client machine, the standard iSCSI deployment was replaced by an SPDK stack composed by an iSCSI client and an NBD virtual layer. The first layer enabled communication with the server through the iSCSI protocol, while the latter exposed the SPDK stack as a local block-device to the workloads.

5º - Only deduplication and no security (NBD + SPDK + iSCSI + SPDK + NVMe). This setup followed the same configuration as the previous while offering deduplication, without any in-place security measures. For this setup, as well as the following ones, there were used two types of implementations, one that stored deduplication related metadata in-memory and another that stored it in a persistent manner.

6º - Enclave based secure deduplication (NBD + SPDK + iSCSI + SPDK + NVMe). This was the first setup to perform secure deduplication, by following the "Enclave Based" security scheme.

7º - Epoch based secure deduplication (NBD + SPDK + iSCSI + SPDK + NVMe). This setup used instead the "Epoch Based" secure deduplication scheme.

8º - Estimated frequency based secure deduplication (NBD + SPDK + iSCSI + SPDK + NVMe). This setup used the "Estimated frequency based" secure deduplication scheme proposed by Li et al.[40].

9º Epoch and exact frequency based secure deduplication (NBD + SPDK + iSCSI + SPDK + NVMe). This setup used the "Epoch and exact frequency based" secure deduplication scheme.

5.4 ENVIRONMENT TESTING

Performing requests over complex remote I/O stacks can have a toll on systems performance. Therefore, before conducting any test on a solution with deduplication or security enabled, it is pertinent to observe and analyse the impact of the storage stack used by S2Dedup prototype. To accomplish this goal, the first four setups presented in Section 5.3 were validated with the DEDISbench benchmarking tool. The experiments were configured with a single benchmarking process and resorted to the *dist_highperf* distribution. For each experiment, 4kB blocks of data were read or written for 20 minutes or until the aggregated I/O totalled 64GB, following a *sequential* or *uniform* access pattern.

The obtained results are shown in Table 2. Also, the resource usage for these tests can be consulted at A.1.1.

		dist_highperf + 1 process							
		Throughput (MiB/s)				Latency(ms)			
		Local disk	Local SPDK	Server-based SPDK	Client and server-based SPDK	Local disk	Local SPDK	Server-based SPDK	Client and server-based SPDK
seq-read	AVG	1729.8	1606.45	471.01	670.65	0.002	0.002	0.008	0.006
	DEV	0.08	1.35	12.58	1.28	0.0	0.0	0.0	0.0
uni-read	AVG	47.05	50.75	30.88	29.54	0.082	0.077	0.126	0.132
	DEV	0.06	0.01	0.29	0.01	0.0	0.0	0.001	0.0
seq-write	AVG	470.31	458.36	453.02	451.21	0.006	0.007	0.007	0.007
	DEV	12.9	8.66	4.75	3.98	0.0	0.0	0.0	0.0
uni-write	AVG	350.43	381.25	304.42	246.01	0.008	0.009	0.01	0.014
	DEV	15.01	6.82	14.73	1.0	0.001	0.001	0.001	0.0

Table 2: Environment results.

As one can infer from Table 2 when running SPDK over an NVMe disk, which is represented by the "Local SPDK" setup, the system does not suffer much impact in either the *write* or *read* tests. However, once the network comes into play, in the "Server-based SPDK" setup, one can easily infer that although there is a noticeable difference in the *write* test, especially when the access pattern is *uniform*, the most drastic change is for the *read* test, diminishing the throughput almost to a quarter in the *sequential* test and to 60% in the *uniform*.

Lastly, for the "Client and server-based SPDK" setup, the *sequential writes* do not suffer almost any change, while the *uniform writes* experience a decrease of almost 50 MiB/s. An interesting behaviour occurs when observing the results of the *read* tests for this setup, that is, there is an increase of 200 MiB/s in the throughput for the *sequential* access when compared to the previous one, although the *uniform* one maintains essentially the same performance. This result can be caused by prefetching mechanisms at the SPDK stack deployed on the client machine.

5.5 PROTOTYPE TESTING

Once established what to expect from the environment and the frameworks used by our solution, one can proceed to evaluate the prototype and its different alternative setups. In the next experimens, six different configurations were used namely: the 4º, 5º, 6º, 7º, 8º and 9º setups presented in Section 5.3, these being referred onwards as "No deduplication", "Only deduplication", "Enclave based", "Epoch based", "Estimated frequency based" and "Epoch and exact frequency based", respectively. Both synthetic and realistic workloads were used to validate each of these setups.

However, before conducting these tests there are some variables that need to be established, these being:

- The duration of an epoch in the "Epoch based" setup.
- The limit of deduplicates per block in the "Estimated frequency based" and "Epoch and exact frequency based" setups.

In our experiments, the duration of a given epoch is calculated by the number of *write* operations that were carried out. However, having bigger or smaller epochs can affect the system's performance, deduplication gains and security. Therefore, in a real system, the duration of an epoch will establish the level of security that is provided and how the performance and deduplication are affected. To establish a value we used DEDISbench and tested the throughput and deduplication ratio for the distribution *dist_highperf* and *dist_kernels* for a deployment without epochs and for deployments that change epochs for every 8, 4 or 2 million of *write* operations.

Based on the results shown at Table 3, we chose 4 million operations as the value for changing epochs. This choice ensures small overhead in terms of performance and space savings, while keeping the epoch duration small, thus increasing the provided security guarantees.

			Epoch			
			No epoch	8 000 000	4 000 000	2 000 000
dist_highperf	**Memory**	Throughput (MiB/s)	106.61	109.15	108.99	109.48
		Deduplication (%)	**17.96**	**17.46**	**17.16**	**17.02**
	Persistent	Throughput (MiB/s)	52.64	52.42	52.38	52.29
		Deduplication (%)	**17.92**	**17.45**	**17.17**	**17.01**
dist_kernels	**Memory**	Throughput (MiB/s)	153.95	149.50	139.75	128.72
		Deduplication (%)	**70.93**	**64.73**	**55.68**	**43.99**
	Persistent	Throughput (MiB/s)	72.85	69.52	65.51	60.85
		Deduplication (%)	**71.23**	**64.48**	**55.50**	**43.87**

Table 3: Variation of throughput and deduplication ratio depending on the epoch duration.

For the second configuration variable, the limit of duplicates per block, we based our choice on the values used by Li et al. [40], which were 5, 10, 15 and 20. For practicability, we chose only one of these values, which was 15 because, similarly to the epoch case, it provides a good balance in terms of security, performance and deduplication gains.

5.5.1 *Synthetic experiments*

For the synthetic benchmarking, two different distributions provided by DEDISbench were used: *dist_highperf* and *dist_kernels*, which have 25% and 72% of duplicate blocks, respectively. For each test 4kB blocks of data were read or written for 20 minutes or until the aggregated I/O totalled 64GB. Moreover, each test followed a *sequential* (seq), *uniform* (uni) or *zipfian* (zip) access pattern and used whether 1 (p1) or 4 (p4) concurrent processes. The obtained results are shown in Tables 4, 5, 6 and 7 while the resource usage results are available at A.1.2.1.

Results Analysis. From Table 4 and 6, one can see that when deduplication related metadata is stored in-memory ("Only deduplication" setup), there is not a detrimental impact on the throughput of the *write* requests. Our solution is even able to reach a higher throughput for the *uniform* and *zipfian* access pattern, which can be justified by the fact that deduplication avoided unnecessary *write* operations to the NVMe disk. However, if the metadata is stored persistently (Table 5 and 7), it suffers a drastic impact, decreasing, for example, from 429.3 MiB/s to only 113.09 MiB/s for the *dist_kernels* distribution with a *sequential* access pattern, which is to be expected from this scenario, since storing data persistently is more costly than in-memory. Namely, extra I/O operations must be done to the persistent metadata structures thus requiring more disk accesses and sharing the disk bandwidth with the workloads operations.

For the *read* test is evident a considerable decrease in the throughput when deduplication is introduced, especially in the *sequential* test for the *dist_highperf* distribution, since it decreases from 673.01 MiB/s to only 254.89 MiB/s in the in-memory solution and to 154.46 MiB/s in the persistent alternative. This behaviour can be justified by the fact that deduplication introduces data fragmentation, thus turning some sequential accesses into random ones. Since the latter pattern is more costly in terms of performance, the throughput decrease observed for sequential reads is to be expected and well documented in the literature [35, 45].

When introducing security ("Enclave based" setup), the system exhibits a decrease in performance, which is to be expected since operating in the SGX enclave is known to be costly. The major differences happened in the *write* tests for the in-memory implementation, which showed, in some cases, a reduction of throughput to almost a quarter, when compared to the setup without security. This occurred because it had such a high throughput previously, that it lead to the security mechanisms not being able to keep up. On the other hand, given that the persistent implementation did not have a similar level of throughput, the differences are not as drastic, being able to maintain the throughput at around 60%, when compared to the baseline deduplication solution. The *read* results followed a similar pattern, reducing its throughput overall, however, the differences are less severe since these requests require less SGX related operations.

In the "Epoch based" setup, there is a minimal difference for the throughput, latency and deduplication ratio from the 'Enclave based" configuration for the distribution *dist_highperf*. This occurred because although the key used for generating a block's hash changed after an epoch, it had minor repercussions in the deduplication gains since it was already relatively small before applying epochs. Therefore, it had less duplicates to be impacted by these alterations.

The same is not observed for the distribution *dist_kernels*, which with the change of epochs suffered a reduction in the deduplication gain, decreasing, for example, from 71.46% to 56.1%, when compared to the non-epoch secure deduplication deployment. This decrease is most noticeable when there is a higher deduplication ratio, than when dealing with a smaller one, and is also affected by the temporal locality properties of the workload. DEDISbench presents a worst-case scenario since content is written uniformly across time and thus, it does not exhibit any temporal locality properties.

Moreover, the decrease on the deduplication ratio meant more *write* operations to disk, thus leading to a lower throughput, which is also observed. The changes in this setup only affect the *write* operation, hence why the results for the *read* tests are similar to the "Enclave based" setup.

The "Estimated frequency based" alternative displayed a slight difference in performance from the "Enclave based" setup for the distribution *dist_highperf*. Considering that there is a limit to the number of deduplicates per block, it is discernible a reduction of around 2%

from the base deduplication gain. Similarly to what happened for the "Epoch based" setup, the *dist_kernels* workload in this setup is even more affected, reducing space savings from 71.46% to 38.77%. Once again, the changes in this setup only affect the *write* operation, so the *read* results have similar performance.

Lastly, we found that the "Epoch and exact frequency based" configuration suffered a minimal difference of about 2% from the "Enclave based" for the *dist_highperf* distribution as well, which is predictable since the previous setup and this one are based on the same principles of limiting the number of deduplicates per block. The distribution *dist_kernels* also experienced a drastic reduction in the deduplication ratio from 71.46% to 40.52%. However, it is noteworthy that it has a slight improvement when compared to the "Estimated frequency based" setup, which only achieved around 38.77%. This happens because, as previously mentioned, when using an estimative, it can lead to inaccurate numbers, and thus to a lower number of deduplicates. There are also no significant changes to the *read* tests, since this setup did not alter the *read* operation. It is also noticeable a minimal decrease in the throughput from the "Estimated frequency based" setup to the "Epoch and exact frequency based". This is happens because although there is a higher deduplication, an exact counter comes with an higher computational cost that an estimated one. But nonetheless, the differences are negligible.

Figure 9 shows a comparison of what to expect in terms of latency when sending a *write* request to each of the setups. From this figure, one can easily grasp a notion on how the deduplication, basic security and its different alternatives affect the system's performance.

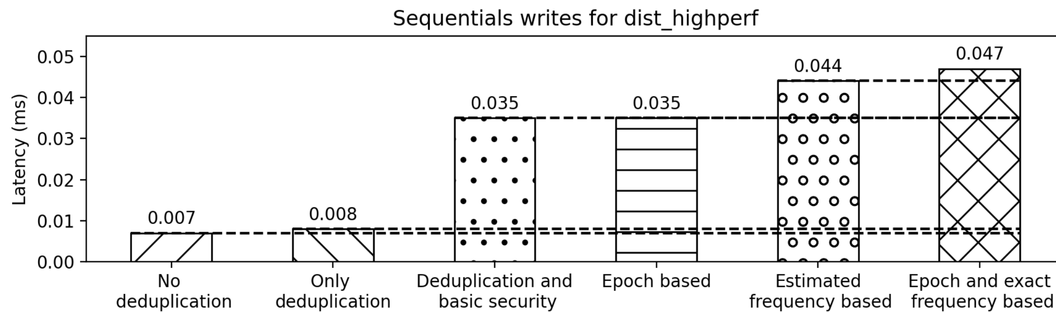


Figure 9: Latency for *sequentials writes* for *dist_highperf*.

Memory			dist_highperf																
			Troughput(MiB/s)						Latency(ms)						Deduplication(%)				
			No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based
seq-read	p1	AVG	673.01	254.89	169.72	169.42	177.72	174.76	0.006	0.015	0.023	0.023	0.022	0.022	-	-	-	-	-
		DEV	0.71	0.15	0.03	0.16	0.14	0.05	0.0	0.0	0.0	0.0	0.0	0.0	-	-	-	-	-
	p4	AVG	1033.7	240.95	162.28	182.17	182.96	182.44	0.06	0.259	0.384	0.343	0.341	0.343	-	-	-	-	-
		DEV	0.36	1.67	0.18	2.14	0.82	0.17	0.0	0.002	0.0	0.005	0.002	0.002	-	-	-	-	-
uni-read	p1	AVG	29.6	26.31	22.81	22.63	23.03	22.6	0.132	0.148	0.171	0.172	0.169	0.173	-	-	-	-	-
		DEV	0.03	0.04	0.09	0.02	0.01	0.01	0.0	0.0	0.001	0.0	0.0	0.001	-	-	-	-	-
	p4	AVG	114.8	100.24	78.63	78.69	79.17	78.25	0.544	0.622	0.792	0.793	0.788	0.797	-	-	-	-	-
		DEV	0.12	0.09	0.05	0.11	0.01	0.08	0.0	0.002	0.001	0.001	0.0	0.002	-	-	-	-	-
zip-read	p1	AVG	122.21	69.52	59.31	59.32	60.44	59.33	0.032	0.056	0.065	0.066	0.064	0.066	-	-	-	-	-
		DEV	0.12	0.18	0.2	0.1	0.07	0.17	0.0	0.0	0.001	0.001	0.0	0.001	-	-	-	-	-
	p4	AVG	454.93	236.98	179.7	179.19	180.94	178.99	0.136	0.264	0.348	0.348	0.344	0.348	-	-	-	-	-
		DEV	0.89	0.16	0.18	0.17	0.2	0.11	0.0	0.0	0.001	0.001	0.0	0.0	-	-	-	-	-
seq-write	p1	AVG	454.6	421.72	105.78	105.9	85.63	80.58	0.007	0.008	0.035	0.035	0.044	0.047	17.81	17.95	17.17	15.93	15.83
		DEV	8.7	0.67	0.29	0.15	0.11	0.06	0.0	0.0	0.001	0.0	0.0	0.0	0.0	0.02	0.01	0.01	0.0
	p4	AVG	446.51	421.6	103.89	105.91	85.35	80.45	0.133	0.141	0.596	0.584	0.725	0.77	17.8	17.95	17.16	15.93	15.83
		DEV	2.13	0.04	2.52	0.41	0.09	0.15	0.002	0.001	0.014	0.004	0.002	0.002	0.01	0.0	0.0	0.01	0.01
uni-write	p1	AVG	249.24	271.35	101.59	101.79	82.53	78.86	0.014	0.013	0.037	0.037	0.046	0.048	17.8	17.87	17.17	15.92	15.83
		DEV	1.24	0.7	0.29	0.16	0.31	0.07	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.01	0.01
	p4	AVG	248.37	267.51	100.9	101.56	82.5	78.24	0.244	0.228	0.613	0.609	0.751	0.792	17.79	17.87	17.17	15.93	15.83
		DEV	0.41	0.04	1.33	0.17	0.42	0.23	0.0	0.0	0.009	0.001	0.005	0.004	0.01	0.01	0.01	0.01	0.01
zip-write	p1	AVG	324.65	457.11	124.2	124.75	101.31	96.59	0.01	0.007	0.03	0.03	0.037	0.039	17.33	17.36	17.06	15.86	15.81
		DEV	4.68	1.12	0.25	0.21	0.38	0.19	0.001	0.0	0.0	0.0	0.0	0.0	0.01	0.01	0.01	0.01	0.0
	p4	AVG	337.57	433.11	118.05	118.32	96.44	91.26	0.179	0.137	0.524	0.521	0.641	0.679	17.33	17.38	17.09	15.86	15.82
		DEV	1.68	1.56	0.16	0.11	0.35	0.96	0.001	0.001	0.001	0.002	0.002	0.008	0.0	0.01	0.01	0.01	0.01

Table 4: Synthetic tests results for the distribution *dist_highperf* for the in-memory implementation.

			dist_highperf																
			Troughput(MiB/s)						Latency(ms)						Deduplication(%)				
			No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based
Persistent	seq-read	p1	AVG	673.01	154.46	151.98	152.4	153.2	151.67	0.006	0.026	0.026	0.026	0.026	0.026	-	-	-	-	-
			DEV	0.71	0.15	1.46	1.32	1.37	1.22	0.0	0.0	0.001	0.001	0.001	0.001	-	-	-	-	-
		p4	AVG	1033.7	240.95	151.3	152.02	151.5	151.83	0.06	0.259	0.412	0.412	0.412	0.412	-	-	-	-	-
			DEV	0.36	1.67	0.22	0.12	0.03	0.05	0.0	0.002	0.0	0.0	0.0	0.0	-	-	-	-	-
	uni-read	p1	AVG	29.6	18.88	22.12	21.96	21.94	21.94	0.132	0.207	0.177	0.178	0.178	0.178	-	-	-	-	-
			DEV	0.03	0.04	0.06	0.03	0.06	0.01	0.0	0.0	0.001	0.001	0.001	0.0	-	-	-	-	-
		p4	AVG	114.8	100.24	73.79	74.14	73.95	73.61	0.544	0.622	0.845	0.841	0.844	0.848	-	-	-	-	-
			DEV	0.12	0.09	0.05	0.02	0.06	0.01	0.0	0.002	0.002	0.0	0.0	0.0	-	-	-	-	-
	zip-read	p1	AVG	122.21	50.21	57.61	57.64	57.64	57.26	0.032	0.078	0.068	0.067	0.067	0.068	-	-	-	-	-
			DEV	0.12	0.18	0.01	0.04	0.08	0.11	0.0	0.0	0.001	0.001	0.001	0.0	-	-	-	-	-
		p4	AVG	454.93	236.98	166.74	167.62	167.29	166.21	0.136	0.264	0.373	0.372	0.372	0.376	-	-	-	-	-
			DEV	0.89	0.16	0.3	0.07	0.21	0.1	0.0	0.0	0.002	0.001	0.0	0.0	-	-	-	-	-
	seq-write	p1	AVG	454.6	88.24	50.84	50.66	45.14	44.39	0.007	0.043	0.075	0.076	0.085	0.086	17.81	17.9	17.18	15.93	15.82
			DEV	8.7	0.67	0.09	0.14	0.35	0.23	0.0	0.0	0.001	0.001	0.001	0.001	0.0	0.01	0.0	0.0	0.0
		p4	AVG	446.51	421.6	50.74	50.5	44.79	44.43	0.133	0.141	1.225	1.231	1.389	1.401	17.8	17.91	17.16	15.93	15.83
			DEV	2.13	0.04	0.18	0.08	0.08	0.27	0.002	0.001	0.005	0.002	0.002	0.008	0.01	0.01	0.01	0.01	0.01
	uni-write	p1	AVG	249.24	84.28	49.71	49.71	44.04	43.36	0.014	0.045	0.077	0.077	0.087	0.089	17.8	17.8	17.16	15.93	15.85
			DEV	1.24	0.7	0.23	0.2	0.23	0.17	0.0	0.0	0.0	0.0	0.0	0.001	0.0	0.01	0.01	0.0	0.0
		p4	AVG	248.37	267.51	49.46	49.18	43.7	43.1	0.244	0.228	1.258	1.265	1.423	1.444	17.79	17.8	17.16	15.92	15.83
			DEV	0.41	0.04	0.1	0.08	0.28	0.09	0.0	0.0	0.003	0.002	0.009	0.004	0.01	0.01	0.0	0.01	0.01
	zip-write	p1	AVG	324.65	135.84	63.49	63.27	55.78	54.63	0.01	0.027	0.06	0.06	0.069	0.07	17.33	17.35	17.08	15.87	15.82
			DEV	4.68	1.12	0.53	0.12	0.32	0.14	0.001	0.0	0.001	0.0	0.001	0.0	0.01	0.0	0.01	0.0	0.0
		p4	AVG	337.57	433.11	59.71	59.92	52.65	51.89	0.179	0.137	1.039	1.037	1.18	1.197	17.33	17.37	17.08	15.88	15.83
			DEV	1.68	1.56	0.12	0.1	0.52	0.04	0.001	0.001	0.002	0.002	0.012	0.002	0.0	0.01	0.01	0.01	0.01

Table 5: Synthetic tests results for the distribution *dist_highperf* for the persistent implementation.

				dist_kernels																
				Troughput(MiB/s)						Latency(ms)						Deduplication(%)				
				No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based
Memory	seq-read	p1	AVG	672.73	339.76	156.46	153.76	172.95	171.5	0.006	0.011	0.025	0.025	0.022	0.023	-	-	-	-	-
			DEV	1.02	3.23	2.57	0.79	0.19	0.15	0.0	0.0	0.001	0.0	0.0	0.0	-	-	-	-	-
		p4	AVG	1035.44	486.59	183.94	183.47	184.21	182.31	0.06	0.128	0.34	0.34	0.34	0.344	-	-	-	-	-
			DEV	0.66	0.14	0.06	0.01	0.14	0.03	0.0	0.0	0.0	0.0	0.0	0.0	-	-	-	-	-
	uni-read	p1	AVG	29.65	26.42	22.95	22.91	22.97	22.64	0.132	0.148	0.17	0.17	0.17	0.172	-	-	-	-	-
			DEV	0.06	0.02	0.1	0.08	0.02	0.03	0.001	0.0	0.001	0.001	0.0	0.0	-	-	-	-	-
		p4	AVG	115.17	102.67	78.88	78.93	79.27	78.2	0.54	0.608	0.792	0.791	0.787	0.797	-	-	-	-	-
			DEV	0.05	0.01	0.03	0.05	0.12	0.05	0.0	0.0	0.0	0.002	0.002	0.002	-	-	-	-	-
	zip-read	p1	AVG	122.84	70.95	60.65	60.71	60.94	59.94	0.032	0.055	0.064	0.064	0.064	0.065	-	-	-	-	-
			DEV	0.29	0.05	0.02	0.15	0.04	0.02	0.001	0.0	0.0	0.0	0.0	0.0	-	-	-	-	-
		p4	AVG	458.01	247.69	182.1	181.89	182.65	180.36	0.136	0.252	0.342	0.343	0.34	0.344	-	-	-	-	-
			DEV	0.61	0.09	0.2	0.19	0.24	0.12	0.0	0.0	0.001	0.002	0.0	0.0	-	-	-	-	-
	seq-write	p1	AVG	455.32	429.3	151.41	135.11	95.37	90.06	0.007	0.008	0.024	0.027	0.039	0.042	70.37	71.46	56.1	38.77	40.52
			DEV	6.34	0.35	0.18	0.32	0.11	0.35	0.0	0.0	0.0	0.0	0.0	0.0	0.01	0.02	0.04	0.0	0.24
		p4	AVG	445.24	430.22	148.09	134.46	95.13	89.76	0.134	0.14	0.416	0.457	0.649	0.689	70.4	71.47	56.09	38.8	40.49
			DEV	3.74	0.21	4.7	0.5	0.3	0.32	0.002	0.0	0.013	0.002	0.002	0.002	0.01	0.03	0.08	0.01	0.25
	uni-write	p1	AVG	251.14	271.87	144.17	127.89	91.78	87.57	0.014	0.013	0.025	0.029	0.041	0.043	70.55	70.89	55.95	38.68	40.39
			DEV	0.13	1.74	0.59	0.32	0.32	0.35	0.0	0.0	0.001	0.0	0.0	0.0	0.04	0.02	0.04	0.01	0.05
		p4	AVG	249.87	270.21	143.7	127.5	91.47	87.36	0.244	0.224	0.428	0.483	0.677	0.708	70.54	70.9	55.93	38.69	40.38
			DEV	0.08	0.15	0.3	0.48	0.29	0.12	0.0	0.0	0.0	0.002	0.002	0.0	0.0	0.01	0.05	0.01	0.03
	zip-write	p1	AVG	329.05	466.31	172.71	155.55	114.11	108.79	0.01	0.007	0.021	0.023	0.033	0.034	64.28	65.25	50.68	36.31	38.11
			DEV	4.87	1.85	0.28	0.41	0.39	0.08	0.0	0.0	0.0	0.001	0.001	0.0	0.03	0.03	0.05	0.02	0.0
		p4	AVG	343.36	445.7	166.16	147.11	108.5	102.87	0.175	0.133	0.368	0.419	0.568	0.601	64.4	64.94	50.11	36.11	38.2
			DEV	4.47	0.33	2.6	0.48	0.05	0.45	0.003	0.001	0.007	0.002	0.0	0.002	0.01	0.43	0.01	0.03	0.06

Table 6: Synthetic tests results for the distribution *dist_kernels* for the in-memory implementation..

				dist_kernels																
				Troughput(MiB/s)						Latency(ms)						Deduplication(%)				
				No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based
Persistent	seq-read	p1	AVG	672.73	221.55	141.66	140.64	149.82	148.74	0.006	0.019	0.028	0.028	0.026	0.026	-	-	-	-	-
			DEV	1.02	70.99	1.44	1.22	1.49	2.24	0.0	0.005	0.001	0.001	0.0	0.001	-	-	-	-	-
		p4	AVG	1035.44	245.82	156.02	151.77	151.78	150.62	0.06	0.254	0.4	0.412	0.412	0.415	-	-	-	-	-
			DEV	0.66	0.98	0.1	0.15	0.24	1.99	0.0	0.002	0.0	0.0	0.0	0.005	-	-	-	-	-
	uni-read	p1	AVG	29.65	22.03	22.22	21.96	21.93	15.63	0.132	0.177	0.175	0.177	0.178	0.549	-	-	-	-	-
			DEV	0.06	0.02	0.05	0.06	0.03	10.91	0.001	0.0	0.001	0.001	0.0	0.642	-	-	-	-	-
		p4	AVG	115.17	76.06	74.44	73.89	74.1	73.56	0.54	0.82	0.839	0.844	0.843	0.848	-	-	-	-	-
			DEV	0.05	0.15	0.07	0.05	0.1	0.53	0.0	0.001	0.002	0.0	0.002	0.007	-	-	-	-	-
	zip-read	p1	AVG	122.84	58.65	58.67	58.22	58.17	53.95	0.032	0.066	0.066	0.067	0.067	0.073	-	-	-	-	-
			DEV	0.29	0.25	0.09	0.18	0.04	6.83	0.001	0.001	0.0	0.0	0.0	0.01	-	-	-	-	-
		p4	AVG	458.01	137.97	170.6	168.56	168.55	167.98	0.136	0.452	0.364	0.368	0.369	0.372	-	-	-	-	-
			DEV	0.61	0.04	0.25	0.13	0.27	0.2	0.0	0.0	0.0	0.001	0.002	0.001	-	-	-	-	-
	seq-write	p1	AVG	455.32	113.09	69.87	63.04	49.91	49.71	0.007	0.033	0.054	0.06	0.077	0.077	71.22	71.39	55.6	38.74	40.65
			DEV	6.34	0.15	0.54	0.14	0.33	0.01	0.0	0.0	0.001	0.001	0.001	0.0	0.03	0.04	0.05	0.02	0.01
		p4	AVG	445.24	111.51	69.91	62.74	49.35	49.47	0.134	0.555	0.888	0.99	1.261	1.257	71.18	71.43	55.07	38.73	40.67
			DEV	3.74	0.84	0.07	0.22	0.23	0.06	0.002	0.005	0.0	0.004	0.005	0.002	0.04	0.01	0.02	0.02	0.04
	uni-write	p1	AVG	251.14	105.73	67.24	60.22	48.31	48.11	0.014	0.035	0.056	0.063	0.079	0.08	70.9	71.11	55.17	38.6	40.32
			DEV	0.13	0.32	0.09	0.04	0.29	0.15	0.0	0.0	0.001	0.0	0.001	0.001	0.02	0.02	0.0	0.03	0.05
		p4	AVG	249.87	105.11	66.27	59.91	48.6	47.89	0.244	0.587	0.936	1.037	1.279	1.299	70.87	71.24	55.11	38.61	40.3
			DEV	0.08	0.39	0.34	0.1	0.35	0.21	0.0	0.002	0.004	0.002	0.008	0.005	0.03	0.02	0.03	0.01	0.04
	zip-write	p1	AVG	329.05	154.94	80.46	74.49	61.17	60.0	0.01	0.024	0.047	0.051	0.062	0.063	64.88	65.27	50.33	36.31	38.32
			DEV	4.87	0.16	0.3	0.12	0.13	0.27	0.0	0.0	0.0	0.0	0.0	0.001	0.01	0.0	0.02	0.01	0.06
		p4	AVG	343.36	151.07	77.5	70.75	58.38	57.08	0.175	0.407	0.8	0.877	1.063	1.088	65.1	65.69	50.13	36.04	37.74
			DEV	4.47	0.46	0.31	0.19	0.11	0.08	0.003	0.001	0.004	0.003	0.002	0.0	0.01	0.02	0.03	0.02	0.01

Table 7: Synthetic tests results for the distribution *dist_kernels* for the persistent implementation.

5.5.2 *Realistic experiments*

To further assess the performance and deduplication gain of S2Dedup prototype we also resorted to realistic storage traces - *mail*, *homes* and *web-vm* - described in Section 5.2. Each run lasted for 40 minutes and I/O operations were issued with a 4kB block size.

We devised a simple tracing tool, written in C, that replayed the I/O events described at the trace files. However, since these traces were collected over a long period of time, for a matter of testing the storage system at different stressing levels, the traces were replayed at diverse speeds (S).

The obtained results can be consulted in the Tables 8, 9 and 10 and the resources consumption at A.1.2.2.

Results Analysis. When deduplication is introduced at the "Only deduplication" setup, for the trace *mail* (Table 8), at a speedup of 400, the throughput remained essentially the same for the in-memory implementation, but experienced a reduction for the persistent implementation, which is a behaviour that was also noticeable for the synthetic tests. Nevertheless, at other speedups and in the other traces, it remained essentially the same. This is justifiable because the peak saturation is reached at a speedup of 400 for the trace *mail*, much like what it happens for DEDISbench that is constantly stressing the storage system.

However, at other speedups, namely 1 and 200, and the other traces, the disk bandwidth peak saturation is not reached, thus it presents essentially no impact with the integration of the deduplication mechanism. For these tests, we should redirect our attention instead to the latency obtained. Although there is some variance between a few values of the same test, which can be justified by factors like the state of the machine and disk, cache mechanisms and the prefetching mechanisms preformed by the SPDK, it was observed that overall the latency for the *read* operation increased. This is understandable since this setup not only requires for the storage system to consult the deduplication engine to figure out where at the storage device to read, but also introduces data fragmentation that leads to an higher latency. For the *write* operations it shows a general decrease in the latency, for example, the trace *web-vm* in the persistent implementation and speedup of 700 lowers from 1.164μs to 0.841μs. This can again be explained by the deduplication employed, which avoided around 21.19% *writes* to disk, thus reducing the overall latency.

The same happens for the "Enclave based" setup results as in the "Only deduplication" setup, the traces *homes* and *web-vm* have almost no variation. However, it is perceptible that for the persistent implementation results, for the trace *mail*, not only the x400 speedup is affected by the changes, but also the x200 speedup. Also, it is evident that the in-memory solution reached peak saturation at x400 of speedup as well. Looking at the latency for the rest of the results, it is also observed some irregularity in the latency results, but it is possible

to infer overall an increase in latency for both the *read* and *write* tests. This is expected since the same was observed for the synthetic tests and results from the hash's computation and data re-encryption performed within the SGX enclave.

With the introduction of epochs in the "Epoch based" setup, the deduplication in all the traces results remained almost the same or had just some minor reduction. This behaviour did not happen for the *dist_kernels* distribution at the synthetic tests, which experienced a reduction from around 70% to 55% when the epochs were employed. This corroborates the idea of temporal locality when dealing with a real workload, which is the concept that is explored by this setup and was not possible to be reproduced with the DEDISbench tool. Moreover, since TrustFS only tested with synthetic workloads, the author was unable to get a perception of the real effect of performing deduplication in epochs. When it comes to the performance, it occurs the same behaviour as for the "Enclave based" setup, although there is a slight throughput reduction in the trace *mail* for the in-memory implementation at x400 speedup, due to a minor change in achievable space savings. Similarly to what happened for the other setups, the latency for some tests demonstrate some variance between runs, but looking at those with a relatively small standard deviation, it shows it is not negatively affected by the employment of epochs.

From the "Enclave based" setup to the "Estimated frequency based", the results show the biggest differences in terms of deduplication ratio. Mainly for the trace *mail* since, in some cases, it even reached a loss of around 14%. For the other traces, this value is around 2%. This inequality does not come as a surprise, since for the synthetic tests the distribution *dist_kernels*, that has the higher number of duplicates, also suffered a greater deduplication loss. Generally, this setup also shows a bigger latency, which can be justified by the decrease in duplicates and the processing required to maintain the approximate counter.

The "Estimated frequency based" and "Epoch and exact frequency based" setups have very similar results, but it is perceptible an overall slight improvement in the deduplication ratio at the "Epoch and exact frequency based" setup results when compared to the "Estimated frequency based". It even reached improvements of 4% for the trace *mail*, which supports the idea of using an exact frequency counter instead of an estimated one. This also shows that although this setup besides limiting the number of duplicates, it also incorporates epochs in its algorithm, the deduplication gains are not affected, whilst still offering an extra layer of security.

				Mail																
				Troughput(MiB/s)						**Latency(μs)**						**Deduplication(%)**				
				No deduplication	**Only deduplication**	**Enclave based**	**Epoch based**	**Estimated frequency based**	**Epoch and exact frequency based**	**No deduplication**	**Only deduplication**	**Enclave based**	**Epoch based**	**Estimated frequency based**	**Epoch and exact frequency based**	**Only deduplication**	**Enclave based**	**Epoch based**	**Estimated frequency based**	**Epoch and exact frequency based**
Memory	**Read**	**S1**	AVG	2.24	2.25	2.25	2.25	2.25	2.25	0.008	13.929	0.016	6.97	27.84	20.884	-	-	-	-	-
			DEV	0.02	0.0	0.0	0.0	0.0	0.0	0.0	12.049	0.002	12.049	31.878	20.872	-	-	-	-	-
		S200	AVG	23.64	23.64	23.64	23.63	23.59	23.56	22.358	48.663	60.907	72.886	96.497	98.675	-	-	-	-	-
			DEV	0.0	0.0	0.01	0.0	0.01	0.11	5.853	5.418	30.126	1.804	1.376	7.447	-	-	-	-	-
		S400	AVG	43.77	44.24	37.20	33.93	28.39	28.18	40.925	53.316	82.409	88.795	100.58	100.718	-	-	-	-	-
			DEV	0.02	0.37	1.02	0.95	0.47	0.27	0.805	6.555	3.817	2.9	5.104	4.628	-	-	-	-	-
	Write	**S1**	AVG	1.197	1.193	1.197	1.2	1.197	1.193	0.002	0.001	0.001	0.001	0.001	0.001	46.84	46.87	46.87	46.68	46.7
			DEV	0.01	0.01	0.01	0.0	0.01	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.02	0.03	0.0
		S200	AVG	181.12	181.12	181.14	181.19	180.45	179.82	0.926	0.623	0.973	1.5	2.62	2.527	84.32	83.5	79.28	71.26	74.77
			DEV	0.0	0.0	0.04	0.0	0.0	1.7	0.317	0.09	0.184	0.076	0.177	0.179	0.09	0.94	0.12	0.07	0.08
		S400	AVG	330.51	331.77	279.09	261.12	218.64	217.66	2.275	1.337	2.839	3.209	4.61	4.714	85.15	84.67	79.65	70.54	74.6
			DEV	0.08	0.95	17.48	3.06	8.83	7.99	0.276	0.121	0.174	0.096	0.246	0.287	0.23	0.16	0.16	0.05	0.07
Persistent	**Read**	**S1**	AVG	2.24	2.24	2.24	2.24	2.25	2.24	0.008	55.712	62.668	83.53	55.728	41.794	-	-	-	-	-
			DEV	0.02	0.02	0.02	0.02	0.0	0.02	0.0	52.52	41.74	20.87	24.104	20.867	-	-	-	-	-
		S200	AVG	23.64	23.61	20.14	21.72	18.12	18.21	22.358	97.907	139.41	126.27	148.22	143.148	-	-	-	-	-
			DEV	0.0	0.04	0.08	1.09	0.14	0.09	5.853	5.885	0.324	5.622	2.428	0.651	-	-	-	-	-
		S400	AVG	43.77	31.77	21.41	22.50	18.50	18.34	40.925	95.611	136.32	124.82	146.79	147.386	-	-	-	-	-
			DEV	0.02	0.39	0.05	0.07	0.32	0.26	0.805	1.656	0.838	0.868	2.808	3.46	-	-	-	-	-
	Write	**S1**	AVG	1.197	1.2	1.197	1.2	1.197	1.197	0.002	0.001	0.001	0.001	0.001	0.001	46.87	46.88	46.54	46.7	46.7
			DEV	0.01	0.0	0.01	0.0	0.01	0.01	0.0	0.0	0.0	0.0	0.0	0.0	0.02	0.03	0.61	0.0	0.0
		S200	AVG	181.12	180.56	166.44	168.61	148.05	149.05	0.926	1.94	5.436	5.632	7.476	7.915	83.48	82.32	78.23	70.47	74.24
			DEV	0.0	1.03	0.11	1.52	1.62	0.51	0.317	0.328	0.063	0.32	0.247	0.191	0.22	0.17	0.28	0.09	0.21
		S400	AVG	330.51	234.93	168.17	169.77	151.51	150.34	2.275	3.558	5.738	6.298	7.688	7.858	83.54	82.43	78.13	70.33	74.14
			DEV	0.08	6.2	0.08	0.16	2.43	2.36	0.276	0.071	0.149	0.091	0.129	0.025	0.18	0.16	0.12	0.09	0.05

Table 8: Realistic tests results for the trace *mail*.

				Homes																
				Troughput(MiB/s)						**Latency(μs)**						**Deduplication(%)**				
				No deduplication	**Only deduplication**	**Enclave based**	**Epoch based**	**Estimated frequency based**	**Epoch and exact frequency based**	**No deduplication**	**Only deduplication**	**Enclave based**	**Epoch based**	**Estimated frequency based**	**Epoch and exact frequency based**	**Only deduplication**	**Enclave based**	**Epoch based**	**Estimated frequency based**	**Epoch and exact frequency based**
Memory	**Read**	**S1**	AVG	0.001	0.001	0.001	0.001	0.001	0.001	0.21	0.2	0.279	0.258	0.265	0.25	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.017	0.001	0.007	0.0	0.008	0.023	-	-	-	-	-
		S350	AVG	0.645	0.645	0.645	0.645	0.645	0.645	114.682	119.746	166.127	159.376	193.951	190.595	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	10.22	16.836	2.925	10.111	5.261	13.931	-	-	-	-	-
		S700	AVG	0.885	0.885	0.885	0.885	0.885	0.885	106.144	114.736	205.532	180.998	193.888	218.426	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	10.129	11.242	7.437	22.713	10.624	10.137	-	-	-	-	-
	Write	**S1**	AVG	0.08	0.08	0.08	0.08	0.08	0.08	0.002	0.001	0.001	0.001	0.001	0.001	41.75	43.51	43.54	41.89	42.02
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.75	0.05	0.45	1.67	0.74
		S350	AVG	22.77	22.77	22.77	22.77	22.77	22.77	0.811	0.62	1.167	0.763	0.739	0.715	20.49	20.72	20.35	18.57	19.32
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.218	0.18	0.694	0.18	0.083	0.357	0.14	0.59	0.08	0.17	0.31
		S700	AVG	27.7	27.7	27.7	27.7	27.7	27.7	1.136	0.823	1.214	0.901	1.253	0.823	20.76	20.82	20.22	18.52	19.31
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.122	0.311	0.628	0.034	0.301	0.156	0.07	0.15	0.37	0.05	0.39
Persistent	**Read**	**S1**	AVG	0.001	0.001	0.001	0.001	0.001	0.001	0.21	0.54	843.645	0.573	422.106	0.5	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.017	0.08	1460.149	0.027	730.119	0.033	-	-	-	-	-
		S350	AVG	0.645	0.645	0.645	0.645	0.645	0.645	114.682	172.874	280.818	233.593	293.474	260.587	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	10.22	10.214	6.699	8.881	43.888	15.382	-	-	-	-	-
		S700	AVG	0.885	0.885	0.885	0.885	0.885	0.885	106.144	177.321	295.106	255.854	333.783	306.773	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	10.129	23.442	11.826	27.106	55.535	14.864	-	-	-	-	-
	Write	**S1**	AVG	0.08	0.08	0.08	0.08	0.08	0.08	0.002	0.001	0.001	0.001	0.001	0.001	42.86	43.71	43.39	42.27	42.25
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.91	0.35	0.48	0.81	0.78
		S350	AVG	22.77	22.77	22.77	22.77	22.77	22.77	0.811	1.12	0.882	1.144	1.048	0.763	20.93	20.49	20.36	18.67	19.34
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.218	0.148	0.419	0.378	0.149	0.229	0.57	0.13	0.13	0.19	0.19
		S700	AVG	27.7	27.7	27.7	27.7	27.7	27.7	1.136	1.018	1.097	0.803	1.018	0.94	20.94	20.56	20.12	18.43	19.34
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.122	0.189	0.148	0.122	0.136	0.305	0.3	0.16	0.17	0.17	0.14

Table 9: Realistic tests results for the trace *homes*.

				Web-VM																
				Troughput(MiB/s)						Latency(μs)						Deduplication(%)				
				No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	No deduplication	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based	Only deduplication	Enclave based	Epoch based	Estimated frequency based	Epoch and exact frequency based
Memory	Read	S1	AVG	0.054	0.054	0.051	0.052	0.053	0.054	0.094	0.102	0.137	0.129	0.133	0.133	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.004	0.002	0.001	0.008	0.004	0.016	-	-	-	-	-
		S350	AVG	1.873	1.874	1.873	1.873	1.873	1.873	10.479	22.995	31.435	27.651	23.576	29.107	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.872	2.807	3.149	2.017	0.873	1.333	-	-	-	-	-
		S700	AVG	4.467	4.467	4.467	4.467	4.467	4.467	4.744	9.852	11.677	12.65	14.353	13.624	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.729	0.729	0.965	1.644	3.544	0.421	-	-	-	-	-
	Write	S1	AVG	0.02	0.02	0.02	0.02	0.02	0.02	0.002	0.002	0.002	0.002	0.002	0.002	61.0	60.97	60.86	59.16	59.27
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.06	0.13	0.21	0.32	0.44
		S350	AVG	6.94	6.94	6.94	6.94	6.94	6.94	1.408	0.861	0.392	0.721	1.329	0.939	22.1	22.01	21.49	19.68	20.93
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.62	0.358	0.136	0.029	0.271	0.235	0.09	0.45	0.79	0.31	1.11
		S700	AVG	16.78	16.78	16.78	16.78	16.78	16.78	1.164	0.873	1.131	0.873	1.39	0.97	20.49	21.32	21.33	19.42	19.6
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.422	0.097	0.34	0.257	0.148	0.256	0.17	0.25	0.3	0.19	0.28
Persistent	Read	S1	AVG	0.054	0.053	0.052	0.053	0.054	0.054	0.094	0.299	0.301	0.313	0.319	0.297	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.004	0.007	0.005	0.006	0.002	0.002	-	-	-	-	-
		S350	AVG	1.873	1.873	1.874	1.874	1.873	1.873	10.479	31.728	39.002	33.763	36.383	30.271	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.872	2.667	1.007	3.064	4.155	3.634	-	-	-	-	-
		S700	AVG	4.467	4.467	4.467	4.467	4.467	4.468	4.744	16.298	17.151	17.028	19.096	16.907	-	-	-	-	-
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.729	1.474	0.365	0.843	2.641	1.172	-	-	-	-	-
	Write	S1	AVG	0.02	0.02	0.02	0.02	0.02	0.02	0.002	0.002	0.002	0.002	0.002	0.002	60.66	60.42	60.72	59.16	59.35
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.001	0.001	0.0	0.0	0.56	0.16	0.04	0.16	0.33
		S350	AVG	6.94	6.94	6.94	6.94	6.94	6.94	1.408	0.86	1.095	0.939	0.743	1.329	21.86	21.82	21.12	19.99	20.64
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.62	0.271	0.271	0.235	0.231	0.59	0.78	0.78	0.29	0.24	0.81
		S700	AVG	16.78	16.78	16.78	16.78	16.78	16.78	1.164	0.841	0.873	1.325	0.808	1.422	21.19	21.55	21.58	20.21	20.5
			DEV	0.0	0.0	0.0	0.0	0.0	0.0	0.422	0.056	0.257	0.551	0.056	0.391	0.72	0.15	0.24	0.28	0.26

Table 10: Realistic tests results for the trace *web-vm*.

5.6 DISCUSSION

With our evaluation, we measured the impact that different secure deduplication schemes have on a system's performance and deduplication effectiveness.

Four different secure deduplication setups were put to test: "Enclave based", "Epoch based", "Estimated frequency based" and "Epoch and exact frequency based", each one offering more robust security guarantees, but, in some cases, with the cost of deduplication and throughput loss. With our evaluation, we assessed, for synthetic and realistic workloads, what is this cost.

Our results show that, as expected, introducing a basic security scheme aided by the Intel SGX ("Enclave based"), did not affect the deduplication gains, but it led to a considerable performance loss, since operating in the SGX enclave is known to be costly. By introducing epochs in this setup ("Epoch based"), we managed to achieve higher security guarantees, such as avoiding leakage of information between data from different epochs, while not affecting much the throughput and maintaining most of the deduplication effectiveness for a realistic workload.

The "Estimated frequency based" and "Epoch and exact frequency based" displayed the lowest performance and deduplication of all the setups. However, these also offer the highest security measures. Furthermore, it is noticeable an improvement in terms of deduplication effectiveness from using an exact counter instead of an estimated one. This proved that combining epochs with the concept of limiting the number of duplicates for a chunk does not affect the deduplication gains, whilst it offers an extra layer of security. Also, it confirms the theory that an exact counter is able to achieve overall better deduplication gains than an estimated counter.

Moreover, with the realistic workload, we found that the performance overhead of secure solutions is less noticeable and in some cases negligible, namely when the disk bandwidth is not being completely saturated. In fact, in a realistic setup, the storage system is not expected to be in a fully saturated mode at all times.

Resource usage across the different setups and secure deduplication schemes are very similar. The only exception, which can be seen at the Appendix Chapter A.1.2, is the increase in RAM usage when running the in-memory solutions, which is to be expected since deduplication metadata is kept in memory.

6 CONCLUSION

This dissertation proposes S2Dedup, a secure deduplication system based on trusted hardware, that takes into consideration not only the security guarantees it provides but also the attained I/O performance and deduplication gain.

Our solution manages to offer secure deduplication with the aid of Intel SGX, which assists in the calculation of block hash sums and data re-encryption. Additionally, and unlike in previous research, it proposes secure schemes that offer different trade-offs in terms of security, performance and deduplication gain. Namely, these schemes enable conducting deduplication in epochs, which prevents attackers from detecting duplicates from different epochs, and also, frequency based solutions, that by limiting the maximum number of duplicate copies per chunk, mask the real number of duplicates per block from an attacker.

The implemented prototype was developed using state of the art technologies, namely SPDK, in order to achieve good I/O performance and better understand the viability of the proposed schemes. This prototype was extensively evaluated with both synthetic and realistic workloads and by comparing multiple setups. The results show that it is possible to implement more robust security techniques, while maintaining overall interesting performance results and even achieve, in some cases, an improvement of deduplication effectiveness when compared to state of the art solutions.

To conclude, S2Dedup is the first solution that provides multiple secure deduplication alternatives that can be adapted to multiple applications according to their specific performance, space savings and security requirements. We strongly believe this is a crucial contribution to promote a wider usage of secure deduplication in third-party storage services while promoting the privacy and security for individuals using those services.

6.1 FUTURE WORK

This work opens the opportunity for new research directions and future improvements over S2Dedup design and implementation, namely:

Firstly, S2Dedup prototype I/O performance is decreased by using persistent metadata. Although this is an expected result that is common across other solutions, if the metadata structures were fully integrated in the SPDK framework, thus avoiding the need to partition the NVME disk and export the partition as an NBD driver for a LevelDB key-value store, one could expect significant performance improvements.

Another possible improvement for this work would be to mask when duplicates occur by applying ORAM [56]. However, this technique is known to have a significant overhead in performance, thus further research is needed to increase the security of the solution while keeping it practical and usable in production environments.

Finally, this solution could be expanded for other types of deduplication, namely, for offline deduplication solutions. In fact, most of the necessary mechanisms are already provided by S2Dedup. This would lead to a reduction of the I/O overheads, since the S2Dedup secure mechanisms would be performed outside of the critical I/O path.

BIBLIOGRAPHY

[1] LevelDB. https://github.com/google/leveldb.git.

[2] Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confisential-computing.

[3] Drew Perttula and Attacks on Convergent Encryption. https://tahoe-lafs.org/hacktahoelafs/drew_perttula.html.

[4] The Digitization of the World From Edge to Core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. Accessed: 2019-09-19.

[5] Dropbox hack leads to leaking of 68m user passwords on the internet . https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach. Accessed: 2019-11-19.

[6] FIU IODedup. http://iotta.snia.org/traces/391.

[7] Glib. https://github.com/GNOME/glib.git.

[8] A More Protected Cloud Environment: IBM Announces Cloud Data Guard Featuring Intel SGX. https://itpeernetwork.intel.com/ibm-cloud-data-guard-intel-sgx/#gs.oejhq1.

[9] Libiscsi. https://github.com/sahlberg/libiscsi.git.

[10] DEDISbench. https://github.com/jtpaulo/dedisbench.

[11] Intel Software Guard Extensions (Intel SGX). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. Accessed: 2019-09-19.

[12] Storage performance development kit. https://spdk.io/, .

[13] Spdk github. https://github.com/spdk/spdk, .

[14] Report: Cloud Storage Data Breach Exposes Users' Private Information . https://www.vpnmentor.com/blog/report-datadepositbox-leak/. Accessed: 2020-10-20.

[15] Ieee standard for cryptographic protection of data on block-oriented storage devices. *IEEE Std 1619-2018 (Revision of IEEE Std 1619-2007)*, pages 1–41, Jan 2019. ISSN null. doi: 10.1109/IEEESTD.2019.8637988.

[16] K Akhila, Amal Ganesh, and C Sunitha. A study on deduplication techniques over encrypted data. *Procedia Computer Science*, 87:38–43, 2016.

[17] Frederik Armknecht, Colin Boyd, Gareth T Davies, Kristian Gjøsteen, and Mohsen Toorani. Side channels in deduplication: trade-offs between leakage and efficiency. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 266–274. ACM, 2017.

[18] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from sgx. In *International Conference on Financial Cryptography and Data Security*, pages 477–497. Springer, 2017.

[19] Mihir Bellare and Sriram Keelveedhi. Interactive message-locked encryption and secure deduplication. In *IACR International Workshop on Public Key Cryptography*, pages 516–538. Springer, 2015.

[20] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 296–312. Springer, 2013.

[21] Steven M Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE, 1992.

[22] George Robert Blakley and Catherine Meadows. Security of ramp schemes. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 242–268. Springer, 1984.

[23] Rongmao Chen, Yi Mu, Guomin Yang, and Fuchun Guo. Bl-mle: Block-level message-locked encryption for secure large file deduplication. *Information Forensics and Security, IEEE Transactions on*, 10:2643–2652, 12 2015. doi: 10.1109/TIFS.2015.2470221.

[24] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[25] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[26] Helei Cui, Huayi Duan, Zhan Qin, Cong Wang, and Yajin Zhou. Speed: Accelerating enclave applications via secure deduplication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1072–1082. IEEE, 2019.

[27] Hung Dang and Ee-Chien Chang. Privacy-preserving data deduplication on trusted processors. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 66–73. IEEE, 2017.

[28] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. *Proceedings on Privacy Enhancing Technologies*, 2017(3):21–38, 2017.

[29] Alfredo De Santis and Barbara Masucci. Multiple ramp schemes. *IEEE Transactions on Information Theory*, 45(5):1720–1728, 1999.

[30] John (JD) Douceur, Atul Adya, Bill Bolosky, Daniel R. Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. Technical Report MSR-TR-2002-30, July 2002. URL https://www.microsoft.com/en-us/research/publication/reclaiming-space-from-duplicate-files-in-a-serverless-distri\buted-file-system/.

[31] Morris J Dworkin. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. Technical report, 2010.

[32] Tânia da Conceição Araújo Esteves, João Tiago Medeiros Paulo, and José Pereira. Sistemas de armazenamento configuráveis e seguros. *Master's dissertation*, 2018.

[33] Danny Harnik, Eliad Tsfadia, Doron Chen, and Ronen Kat. Securing the storage data path with sgx enclaves. *arXiv preprint arXiv:1806.10883*, 2018.

[34] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):13, 2018.

[35] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[36] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. Dupless: server-aided encryption for deduplicated storage. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 179–194, 2013.

[37] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, page 7. ACM, 2015.

[38] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010.

[39] Jin Li, Xiaofeng Chen, Mingqiang Li, Jingwei Li, Patrick PC Lee, and Wenjing Lou. Secure deduplication with efficient and reliable convergent key management. *IEEE transactions on parallel and distributed systems*, 25(6):1615–1625, 2013.

[40] Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick PC Lee, and Xiaosong Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[41] Jian Liu, Nadarajah Asokan, and Benny Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 874–885. ACM, 2015.

[42] Takanori Machida, Dai Yamamoto, Ikuya Morikawa, Hirotaka Kokubo, and Hisashi Kojima. Poster: A novel framework for user-key provisioning to secure enclaves on intel sgx.

[43] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2017.

[44] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[45] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 581–586. IEEE, 2011.

[46] João Paulo and José Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[47] João Paulo and José Pereira. Efficient deduplication in a distributed primary storage infrastructure. *ACM Transactions on Storage (TOS)*, 12(4):1–35, 2016.

[48] Joao Paulo, Pedro Reis, Jose Pereira, and Antonio Sousa. Dedisbench: A benchmark for deduplicated storage systems. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 584–601. Springer, 2012.

[49] AJ Paverd, Andrew Martin, and Ian Brown. Modelling and automatically analysing privacy properties for honest-but-curious adversaries. *Tech. Rep.*, 2014.

[50] Zahra Pooranian, Kang-Cheng Chen, Chia-Mu Yu, and Mauro Conti. Rare: Defeating side channels based on data-deduplication in cloud storage. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 444–449. IEEE, 2018.

[51] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. Cloudedup: Secure deduplication with encrypted data for cloud storage. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 363–370. IEEE, 2013.

[52] Joydeep Rakshit and Kartik Mohanram. Leo: Low overhead encryption oram for non-volatile memories. *IEEE Computer Architecture Letters*, 17(2):100–104, 2018.

[53] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[54] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[55] Jan Stanek, Alessandro Sorniotti, Elli Androulaki, and Lukas Kencl. A secure data deduplication scheme for cloud storage. In *International conference on financial cryptography and data security*, pages 99–118. Springer, 2014.

[56] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[57] Bruno Vavala, Nuno Neves, and Peter Steenkiste. Secure tera-scale data crunching with a small tcb. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 169–180. IEEE, 2017.

[58] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. *ACM SIGARCH Computer Architecture News*, 45(2): 81–93, 2017.

[59] Dag Wieers. Dstat: Versatile resource statistics tool. *http://dag.wiee.rs/home-made/dstat*.

[60] Qirui Yang, Runyu Jin, and Ming Zhao. Smartdedup: Optimizing deduplication for resource-constrained devices. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 633–646, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/yang-qirui.

[61] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[62] H. Yu, X. Zhang, W. Huang, and W. Zheng. Pdfs: Partially dedupped file system for primary workloads. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):863–876, 2017.

[63] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. Fork path: improving efficiency of oram by removing redundant memory accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 102–114. IEEE, 2015.

[64] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 283–298, 2017.

[65] Y. Zhou, Y. Deng, L. T. Yang, R. Yang, and L. Si. Ldfs: A low latency in-line data deduplication file system. *IEEE Access*, 6:15743–15753, 2018.

[66] Yukun Zhou, Dan Feng, Wen Xia, Min Fu, Fangting Huang, Yucheng Zhang, and Chunguang Li. Secdep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

APPENDIX

A.1 RESOURCE USAGE

A.1.1 *Environment Results*

		read + dist_highperf + 1 process											
		Server				Client							
		CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
		AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
Local disk	**Sequential**	6.08	0.08	0.64	0.0	-	-	-	-	-	-	-	-
	Uniform	2.6	0.02	0.64	0.0	-	-	-	-	-	-	-	-
Local SPDK	**Sequential**	17.11	0.21	2.78	0.0	-	-	-	-	-	-	-	-
	Uniform	17.8	0.01	2.77	0.0	-	-	-	-	-	-	-	-
Server-based SPDK	**Sequential**	15.22	0.05	2.28	0.0	9.89	0.31	0.65	0.01	392.537	14.493	1.112	0.048
	Uniform	16.54	0.01	2.28	0.0	1.08	0.01	0.65	0.0	28.57	0.266	1.115	0.01
Client and server-based SPDK	**Sequential**	15.27	0.05	2.27	0.0	18.99	0.2	2.8	0.0	526.782	0.922	1.258	0.007
	Uniform	16.57	0.01	2.27	0.0	17.45	0.04	2.78	0.0	27.508	0.011	1.087	0.0

Table 11: Resource usage by the *read* operations of the environment tests.

		write + dist_highperf + 1 process											
		Server				Client							
		CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
		AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
Local disk	**Sequential**	6.89	0.41	0.68	0.01	-	-	-	-	-	-	-	-
	Uniform	8.08	0.29	0.64	0.0	-	-	-	-	-	-	-	-
Local SPDK	**Sequential**	31.01	0.09	2.79	0.0	-	-	-	-	-	-	-	-
	Uniform	31.83	0.13	2.78	0.01	-	-	-	-	-	-	-	-
Server-based SPDK	**Sequential**	15.73	0.02	2.28	0.0	15.74	0.45	0.71	0.01	0.773	0.006	379.976	2.084
	Uniform	16.13	0.06	2.28	0.0	16.89	0.82	0.65	0.01	4.848	0.251	262.757	14.366
Client and server-based SPDK	**Sequential**	15.79	0.02	2.27	0.0	33.15	0.53	2.8	0.0	1.375	0.023	376.024	6.121
	Uniform	16.31	0.03	2.27	0.0	26.75	0.14	2.8	0.0	4.288	0.011	218.965	0.629

Table 12: Resource usage by the *write* operations of the environment tests.

A.1.2 *Prototype Results*

A.1.2.1 *Synthetic experiments*

				read + dist_highperf											
				Server				Client							
				CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
				AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No deduplication	**-**	**Sequential**	**p1**	15.88	0.02	2.28	0.0	20.97	0.06	2.8	0.0	528.705	0.61	1.26	0.004
			p4	15.64	0.02	2.27	0.0	22.75	0.06	3.14	0.06	720.726	0.269	1.512	0.002
		Uniform	**p1**	16.64	0.02	2.28	0.0	17.61	0.01	2.78	0.0	27.56	0.034	1.088	0.001
			p4	16.71	0.01	2.27	0.0	20.48	0.03	3.16	0.01	104.089	0.056	3.882	0.003
		Zipfian	**p1**	16.53	0.0	2.28	0.0	18.11	0.03	2.8	0.0	28.89	0.018	1.055	0.001
			p4	16.18	0.02	2.27	0.0	20.93	0.02	3.12	0.09	91.529	0.053	3.185	0.001
Only deduplication	**Memory**	**Sequential**	**p1**	42.04	14.81	7.75	0.0	17.12	0.05	3.04	0.02	237.167	0.141	0.892	0.001
			p4	50.19	0.1	7.75	0.0	17.37	0.03	4.13	0.08	226.962	1.493	0.754	0.015
		Uniform	**p1**	50.5	0.07	7.75	0.0	17.23	0.02	3.03	0.0	26.031	0.031	1.074	0.001
			p4	51.04	0.12	7.75	0.0	19.66	0.02	4.2	0.0	97.338	0.097	3.835	0.005
		Zipfian	**p1**	50.65	0.14	7.75	0.0	17.43	0.05	3.05	0.01	27.908	0.06	1.065	0.002
			p4	50.97	0.09	7.75	0.0	19.42	0.02	4.15	0.08	95.33	0.059	3.482	0.003
	Persistent	**Sequential**	**p1**	32.98	18.11	3.14	0.72	17.0	0.09	3.06	0.0	148.29	32.078	0.547	0.118
			p4	42.53	2.68	2.97	0.76	17.01	0.02	4.2	0.01	134.177	0.864	0.423	0.002
		Uniform	**p1**	44.89	0.2	2.97	0.76	16.99	0.01	3.05	0.01	18.758	0.028	0.775	0.001
			p4	44.57	0.46	2.97	0.76	18.71	0.01	4.19	0.03	64.245	0.659	2.545	0.026
		Zipfian	**p1**	45.19	0.49	2.97	0.76	17.21	0.01	3.04	0.0	20.589	0.083	0.787	0.003
			p4	42.6	2.66	2.97	0.76	18.01	0.05	4.2	0.0	43.805	0.759	1.605	0.028
Enclave based	**Memory**	**Sequential**	**p1**	50.07	0.27	7.85	0.0	17.83	0.11	3.03	0.02	163.938	0.011	0.48	0.004
			p4	50.39	0.18	7.85	0.0	18.01	0.02	4.11	0.06	157.639	0.218	0.395	0.001
		Uniform	**p1**	50.41	0.2	7.85	0.0	17.43	0.06	3.02	0.01	22.6	0.095	0.933	0.004
			p4	50.8	0.05	7.85	0.0	19.5	0.04	4.15	0.03	77.038	0.059	2.903	0.003
		Zipfian	**p1**	50.4	0.08	7.85	0.0	17.63	0.03	3.02	0.02	23.908	0.071	0.908	0.003
			p4	50.79	0.22	7.85	0.0	19.93	0.07	4.1	0.07	73.967	0.093	2.554	0.003
	Persistent	**Sequential**	**p1**	49.03	0.08	3.75	0.0	17.36	0.01	3.03	0.01	148.374	1.491	0.52	0.006
			p4	48.98	0.14	3.75	0.0	17.52	0.01	4.15	0.06	147.7	0.181	0.343	0.0
		Uniform	**p1**	49.99	0.05	3.75	0.0	17.26	0.02	3.01	0.01	21.932	0.061	0.905	0.002
			p4	50.1	0.04	3.75	0.0	19.13	0.02	4.15	0.01	72.481	0.048	2.693	0.003
		Zipfian	**p1**	49.87	0.04	3.75	0.0	17.48	0.01	3.03	0.01	23.235	0.008	0.883	0.0
			p4	49.2	0.05	3.75	0.0	19.3	0.03	4.12	0.06	69.073	0.088	2.362	0.003

Epoch based	Memory	Sequential	p1	49.22	0.19	7.96	0.0	17.35	0.03	3.04	0.0	163.506	0.169	0.5	0.001
			p4	48.85	0.38	7.96	0.0	17.57	0.02	4.19	0.0	175.066	1.368	0.415	0.002
		Uniform	p1	50.05	0.22	7.96	0.0	17.28	0.04	3.03	0.01	22.422	0.021	0.925	0.001
			p4	50.17	0.05	7.96	0.0	19.22	0.03	4.17	0.03	77.076	0.112	2.896	0.005
		Zipfian	p1	50.01	0.03	7.96	0.0	17.42	0.03	3.03	0.01	23.883	0.065	0.907	0.002
			p4	49.39	0.13	7.96	0.0	19.26	0.02	4.16	0.04	73.77	0.045	2.537	0.003
	Persistent	Sequential	p1	49.21	0.15	3.78	0.0	17.47	0.02	3.04	0.0	148.815	1.31	0.487	0.004
			p4	49.28	0.09	3.77	0.0	17.59	0.01	4.15	0.06	148.428	0.159	0.348	0.001
		Uniform	p1	50.09	0.12	3.78	0.0	17.27	0.06	3.03	0.01	21.771	0.03	0.899	0.001
			p4	50.48	0.08	3.77	0.0	19.13	0.02	4.17	0.03	72.816	0.015	2.708	0.001
		Zipfian	p1	50.15	0.09	3.78	0.0	17.51	0.03	3.04	0.0	23.241	0.026	0.883	0.001
			p4	49.37	0.12	3.77	0.0	19.32	0.02	4.07	0.06	69.373	0.052	2.371	0.002
Estimated frequency based	Memory	Sequential	p1	50.05	0.37	8.02	0.0	18.04	0.01	3.01	0.0	171.633	0.132	0.58	0.004
			p4	50.56	0.18	8.02	0.0	18.29	0.02	4.14	0.07	175.819	0.353	0.414	0.002
		Uniform	p1	50.66	0.17	8.02	0.0	17.45	0.02	3.01	0.01	22.82	0.003	0.942	0.0
			p4	51.11	0.23	8.02	0.0	19.55	0.03	4.17	0.03	77.577	0.011	2.908	0.001
		Zipfian	p1	50.6	0.18	8.02	0.0	17.7	0.02	3.04	0.0	24.366	0.015	0.925	0.001
			p4	50.71	0.15	8.02	0.0	19.98	0.02	4.18	0.0	74.487	0.09	2.559	0.003
	Persistent	Sequential	p1	49.15	0.16	3.79	0.0	17.4	0.05	3.04	0.0	149.632	1.347	0.513	0.004
			p4	49.05	0.24	3.79	0.0	17.57	0.02	4.15	0.06	147.907	0.048	0.345	0.0
		Uniform	p1	50.06	0.03	3.79	0.0	17.27	0.06	3.03	0.01	21.757	0.053	0.898	0.002
			p4	50.22	0.1	3.79	0.0	19.16	0.02	4.15	0.03	72.642	0.055	2.702	0.002
		Zipfian	p1	50.05	0.12	3.79	0.0	17.51	0.03	3.03	0.0	23.237	0.029	0.883	0.001
			p4	49.44	0.13	3.79	0.0	19.33	0.02	4.14	0.06	69.275	0.117	2.369	0.004
Epoch and exact frequency based	Memory	Sequential	p1	48.68	0.05	8.04	0.0	17.22	0.03	3.01	0.01	168.746	0.081	0.574	0.002
			p4	48.43	0.11	8.04	0.0	17.47	0.07	3.98	0.07	175.617	0.43	0.382	0.003
		Uniform	p1	49.96	0.06	8.04	0.0	17.19	0.01	3.02	0.01	22.395	0.02	0.924	0.001
			p4	49.95	0.05	8.04	0.0	18.98	0.03	4.19	0.0	76.678	0.082	2.875	0.004
		Zipfian	p1	49.92	0.09	8.04	0.0	17.42	0.01	3.03	0.01	23.909	0.058	0.908	0.002
			p4	48.89	0.05	8.04	0.0	19.06	0.01	4.11	0.06	73.679	0.057	2.536	0.002
	Persistent	Sequential	p1	49.01	0.1	3.8	0.0	17.39	0.01	3.04	0.0	148.119	1.214	0.515	0.005
			p4	49.16	0.17	3.8	0.0	17.55	0.04	4.15	0.06	148.239	0.021	0.345	0.001
		Uniform	p1	50.0	0.04	3.8	0.0	17.22	0.01	3.01	0.01	21.757	0.012	0.898	0.001
			p4	50.06	0.07	3.8	0.0	19.11	0.02	4.14	0.01	72.305	0.006	2.687	0.0
		Zipfian	p1	49.91	0.03	3.8	0.0	17.48	0.04	3.03	0.0	23.092	0.042	0.878	0.002
			p4	49.24	0.16	3.8	0.0	19.35	0.08	4.12	0.06	68.882	0.066	2.355	0.002

Table 13: Resource usage by the *read* operations of the synthetic tests for the distribution *dist_highperf*.

				write + dist_highperf											
				Server				Client							
				CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
				AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No deduplication	-	Sequential	p1	16.23	0.01	2.27	0.0	34.26	0.73	2.8	0.0	1.389	0.023	378.806	9.121
			p4	16.24	0.02	2.27	0.0	31.46	0.19	3.18	0.0	1.32	0.015	359.14	3.511
		Uniform	p1	16.56	0.04	2.27	0.0	27.22	0.07	2.8	0.0	4.341	0.021	222.018	0.874
			p4	16.55	0.01	2.27	0.0	27.16	0.06	3.17	0.0	4.336	0.008	221.486	0.298
		Zipfian	p1	16.38	0.0	2.27	0.0	24.6	0.54	2.8	0.0	2.396	0.038	144.183	2.349
			p4	16.39	0.01	2.27	0.0	23.75	0.1	3.17	0.0	2.43	0.01	147.172	0.676
Only deduplication	Memory	Sequential	p1	47.9	0.31	3.42	0.0	22.41	0.1	3.0	0.04	0.844	0.001	355.89	0.812
			p4	47.94	0.05	3.43	0.0	19.95	0.04	4.09	0.13	0.847	0.001	357.07	0.433
		Uniform	p1	49.38	0.16	3.4	0.0	23.4	0.03	3.05	0.0	4.041	0.006	245.275	0.533
			p4	49.37	0.13	3.39	0.0	21.57	0.05	4.19	0.02	4.015	0.002	242.843	0.139
		Zipfian	p1	48.18	0.13	2.96	0.0	24.13	0.05	3.04	0.01	3.469	0.001	246.487	0.283
			p4	48.59	0.17	3.02	0.0	21.3	0.03	4.12	0.03	3.704	0.179	253.117	0.98
	Persistent	Sequential	p1	49.72	0.05	2.61	0.0	17.04	0.02	3.04	0.0	0.237	0.001	86.087	0.224
			p4	49.58	0.08	2.62	0.0	17.06	0.01	4.18	0.04	0.237	0.001	86.224	0.446
		Uniform	p1	49.94	0.13	2.6	0.0	17.92	0.02	3.04	0.0	1.148	0.005	82.706	0.351
			p4	49.99	0.02	2.6	0.0	17.94	0.03	4.17	0.03	1.152	0.001	82.778	0.043
		Zipfian	p1	48.64	0.49	2.46	0.0	17.81	0.39	3.04	0.0	1.06	0.002	86.785	0.159
			p4	49.46	0.14	2.48	0.0	17.69	0.06	4.19	0.0	1.08	0.005	87.511	0.374
Enclave based	Memory	Sequential	p1	49.84	0.16	3.58	0.01	17.6	0.01	3.05	0.0	0.281	0.001	104.232	0.297
			p4	49.88	0.12	3.59	0.01	17.63	0.04	4.16	0.04	0.277	0.007	102.627	2.509
		Uniform	p1	49.83	0.03	3.52	0.0	18.83	0.01	3.04	0.0	1.425	0.004	101.42	0.297
			p4	49.83	0.03	3.53	0.01	18.87	0.09	4.15	0.04	1.425	0.017	101.203	1.201
		Zipfian	p1	49.65	0.05	3.16	0.0	18.59	0.01	3.04	0.01	1.256	0.002	98.505	0.135
			p4	49.74	0.06	3.17	0.0	18.74	0.04	4.16	0.04	1.277	0.001	98.61	0.091
	Persistent	Sequential	p1	50.15	0.08	2.7	0.0	17.14	0.01	3.03	0.01	0.143	0.001	51.41	0.214
			p4	50.07	0.05	2.7	0.0	17.19	0.03	4.17	0.05	0.143	0.001	51.559	0.213
		Uniform	p1	50.11	0.04	2.68	0.0	17.71	0.02	3.03	0.01	0.703	0.003	50.649	0.236
			p4	50.11	0.07	2.68	0.0	17.78	0.03	4.16	0.02	0.705	0.002	50.663	0.124
		Zipfian	p1	49.99	0.17	2.58	0.0	17.7	0.02	3.04	0.01	0.656	0.008	51.634	0.665
			p4	50.07	0.04	2.59	0.0	17.75	0.01	4.18	0.02	0.668	0.001	51.747	0.056

Epoch based	**Memory**	**Sequential**	**p1**	50.04	0.07	3.57	0.0	17.61	0.03	3.03	0.01	0.283	0.002	104.348	0.153
			p4	50.11	0.11	3.57	0.0	17.66	0.02	4.18	0.03	0.283	0.001	104.687	0.403
		Uniform	**p1**	50.09	0.12	3.51	0.0	18.75	0.01	3.04	0.01	1.428	0.004	101.645	0.196
			p4	50.13	0.18	3.51	0.0	18.81	0.02	4.17	0.04	1.433	0.004	101.806	0.205
		Zipfian	**p1**	49.9	0.08	3.14	0.0	18.51	0.02	3.04	0.02	1.264	0.002	98.94	0.128
			p4	50.13	0.18	3.16	0.0	18.65	0.02	4.15	0.05	1.281	0.003	98.746	0.192
	Persistent	**Sequential**	**p1**	50.29	0.02	2.69	0.0	17.15	0.03	3.04	0.01	0.142	0.0	51.128	0.173
			p4	50.26	0.09	2.69	0.0	17.18	0.03	4.17	0.03	0.142	0.0	51.115	0.098
		Uniform	**p1**	50.27	0.1	2.66	0.0	17.68	0.03	3.04	0.01	0.703	0.003	50.697	0.191
			p4	50.23	0.03	2.66	0.0	17.73	0.02	4.14	0.02	0.701	0.002	50.391	0.102
		Zipfian	**p1**	50.21	0.11	2.57	0.0	17.65	0.0	3.05	0.01	0.655	0.001	51.581	0.077
			p4	50.3	0.08	2.57	0.0	17.73	0.04	4.18	0.02	0.669	0.001	51.866	0.054
Estimated frequency based	**Memory**	**Sequential**	**p1**	49.72	0.35	3.59	0.0	17.35	0.04	3.05	0.0	0.234	0.0	85.204	0.109
			p4	49.54	0.08	3.59	0.0	17.42	0.01	4.15	0.02	0.234	0.0	85.154	0.089
		Uniform	**p1**	49.7	0.07	3.54	0.0	18.28	0.01	3.04	0.01	1.158	0.005	83.141	0.297
			p4	49.71	0.16	3.54	0.0	18.33	0.02	4.19	0.0	1.165	0.003	83.441	0.39
		Zipfian	**p1**	49.63	0.08	3.16	0.0	18.1	0.05	3.04	0.01	1.032	0.004	81.131	0.322
			p4	49.36	0.05	3.18	0.0	18.2	0.04	4.16	0.03	1.05	0.005	81.327	0.329
	Persistent	**Sequential**	**p1**	49.77	0.03	2.68	0.0	17.03	0.02	3.03	0.01	0.125	0.001	45.485	0.319
			p4	49.75	0.03	2.68	0.0	17.08	0.03	4.14	0.02	0.125	0.0	45.317	0.072
		Uniform	**p1**	49.85	0.07	2.66	0.0	17.52	0.02	3.03	0.01	0.622	0.003	44.852	0.208
			p4	49.8	0.08	2.66	0.0	17.55	0.03	4.13	0.01	0.623	0.003	44.733	0.264
		Zipfian	**p1**	49.81	0.03	2.58	0.0	17.5	0.03	3.04	0.01	0.581	0.003	45.764	0.259
			p4	49.81	0.04	2.58	0.0	17.54	0.06	4.18	0.03	0.588	0.005	45.516	0.419
Epoch and exact frequency based	**Memory**	**Sequential**	**p1**	50.2	0.28	3.61	0.0	17.46	0.06	3.04	0.01	0.222	0.001	80.388	0.048
			p4	50.22	0.07	3.61	0.0	17.48	0.03	4.18	0.03	0.223	0.001	80.493	0.176
		Uniform	**p1**	50.26	0.24	3.54	0.0	18.36	0.01	3.03	0.01	1.109	0.001	79.573	0.063
			p4	50.12	0.18	3.55	0.01	18.46	0.06	4.15	0.04	1.107	0.003	79.254	0.227
		Zipfian	**p1**	50.14	0.17	3.17	0.0	18.17	0.01	3.04	0.01	0.985	0.002	77.488	0.142
			p4	50.25	0.25	3.19	0.01	18.27	0.03	4.15	0.03	0.995	0.01	77.136	0.765
	Persistent	**Sequential**	**p1**	50.21	0.14	2.68	0.0	17.07	0.0	3.03	0.01	0.126	0.001	44.723	0.216
			p4	50.13	0.07	2.68	0.0	17.12	0.03	4.14	0.02	0.126	0.001	44.91	0.295
		Uniform	**p1**	50.21	0.1	2.66	0.0	17.59	0.01	3.04	0.0	0.613	0.003	44.107	0.202
			p4	50.24	0.05	2.66	0.0	17.65	0.0	4.17	0.02	0.613	0.001	44.03	0.101
		Zipfian	**p1**	50.19	0.07	2.58	0.0	17.55	0.01	3.04	0.01	0.569	0.002	44.77	0.158
			p4	50.27	0.05	2.58	0.0	17.64	0.03	4.14	0.05	0.579	0.001	44.846	0.051

Table 14: Resource usage by the *write* operations of the synthetic tests for the distribution *dist_highperf*.

				read + dist_kernels											
				Server				Client							
				CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
				AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No deduplication	-	Sequential	p1	16.02	0.01	2.28	0.0	21.07	0.05	2.42	0.0	528.474	0.832	1.262	0.007
			p4	15.83	0.01	2.27	0.0	22.99	0.04	2.8	0.0	722.568	0.402	1.509	0.005
		Uniform	p1	16.67	0.03	2.27	0.0	17.64	0.0	2.42	0.0	27.581	0.05	1.087	0.002
			p4	16.73	0.0	2.27	0.0	20.51	0.04	2.77	0.0	104.059	0.017	3.869	0.001
		Zipfian	p1	16.56	0.0	2.27	0.0	18.12	0.03	2.42	0.0	28.872	0.079	1.054	0.003
			p4	16.28	0.01	2.27	0.0	21.09	0.06	2.8	0.0	91.697	0.168	3.188	0.005
Only deduplication	Memory	Sequential	p1	50.23	0.2	4.14	0.0	17.22	0.04	2.68	0.0	304.4	3.01	1.094	0.009
			p4	50.36	0.18	4.14	0.0	17.64	0.06	3.8	0.05	405.308	0.06	1.088	0.001
		Uniform	p1	50.62	0.16	4.14	0.0	17.22	0.02	2.67	0.01	26.115	0.014	1.077	0.001
			p4	50.98	0.15	4.14	0.0	19.77	0.04	3.83	0.0	99.581	0.006	3.92	0.001
		Zipfian	p1	50.62	0.13	4.14	0.0	17.43	0.01	2.68	0.0	28.214	0.013	1.076	0.0
			p4	50.87	0.17	4.14	0.0	19.13	0.36	3.82	0.0	97.434	0.067	3.549	0.002
	Persistent	Sequential	p1	43.2	1.64	2.79	0.01	17.18	0.11	2.68	0.02	208.255	60.208	0.742	0.218
			p4	40.85	3.45	2.79	0.01	16.99	0.03	3.83	0.0	202.941	2.163	0.545	0.006
		Uniform	p1	45.38	1.62	2.79	0.01	17.11	0.04	2.68	0.01	21.823	0.02	0.901	0.001
			p4	44.88	0.77	2.79	0.01	19.03	0.02	3.83	0.0	74.003	0.199	2.926	0.008
		Zipfian	p1	45.44	1.61	2.79	0.01	17.34	0.05	2.68	0.0	23.449	0.133	0.894	0.005
			p4	42.71	1.55	2.79	0.01	18.28	0.01	3.83	0.0	54.83	0.032	2.004	0.001
Enclave based	Memory	Sequential	p1	49.91	0.15	4.24	0.0	17.69	0.06	2.67	0.0	152.529	2.027	0.507	0.014
			p4	49.87	0.3	4.24	0.0	17.93	0.09	3.79	0.03	176.432	0.04	0.408	0.001
		Uniform	p1	50.41	0.02	4.24	0.0	17.36	0.01	2.66	0.01	22.725	0.09	0.938	0.004
			p4	50.64	0.14	4.24	0.0	19.36	0.06	3.81	0.0	77.197	0.024	2.888	0.001
		Zipfian	p1	50.49	0.05	4.24	0.0	17.62	0.02	2.66	0.0	24.243	0.016	0.92	0.0
			p4	50.25	0.1	4.24	0.0	19.79	0.05	3.72	0.03	74.177	0.034	2.535	0.001
	Persistent	Sequential	p1	48.87	0.08	2.89	0.0	17.29	0.04	2.66	0.01	138.954	1.415	0.466	0.009
			p4	48.89	0.19	2.88	0.0	17.51	0.02	3.73	0.04	152.35	0.077	0.354	0.0
		Uniform	p1	49.96	0.08	2.89	0.0	17.21	0.04	2.65	0.01	22.008	0.043	0.908	0.002
			p4	50.04	0.05	2.88	0.0	19.08	0.01	3.81	0.01	73.046	0.079	2.713	0.003
		Zipfian	p1	49.85	0.03	2.88	0.0	17.44	0.02	2.66	0.0	23.469	0.04	0.89	0.001
			p4	48.97	0.06	2.88	0.0	19.21	0.05	3.81	0.0	69.543	0.077	2.368	0.003

Epoch based	Memory	Sequential	p1	49.14	0.26	5.48	0.0	17.37	0.03	2.67	0.0	149.968	0.871	0.473	0.008
			p4	48.82	0.05	5.48	0.0	17.54	0.01	3.81	0.0	176.01	0.011	0.411	0.0
		Uniform	p1	50.18	0.24	5.48	0.0	17.29	0.03	2.66	0.01	22.684	0.076	0.936	0.003
			p4	50.4	0.18	5.48	0.0	19.2	0.05	3.81	0.0	77.238	0.044	2.893	0.002
		Zipfian	p1	49.96	0.08	5.48	0.0	17.45	0.02	2.67	0.0	24.241	0.045	0.92	0.002
			p4	49.09	0.21	5.48	0.0	19.32	0.02	3.77	0.05	74.139	0.057	2.537	0.003
	Persistent	Sequential	p1	49.38	0.32	3.19	0.0	17.43	0.01	2.67	0.0	137.576	1.347	0.434	0.008
			p4	49.22	0.16	3.19	0.0	17.63	0.02	3.82	0.0	148.234	0.158	0.347	0.0
		Uniform	p1	50.12	0.1	3.19	0.0	17.28	0.06	2.66	0.01	21.756	0.066	0.898	0.003
			p4	50.29	0.04	3.19	0.0	19.14	0.01	3.82	0.0	72.515	0.043	2.693	0.002
		Zipfian	p1	50.14	0.11	3.19	0.0	17.52	0.09	2.67	0.0	23.284	0.082	0.883	0.003
			p4	49.72	0.18	3.19	0.0	19.34	0.02	3.72	0.04	68.873	0.18	2.344	0.005
Estimated frequency based	Memory	Sequential	p1	49.68	0.26	6.28	0.0	17.81	0.03	2.67	0.0	167.072	0.173	0.566	0.003
			p4	50.28	0.01	6.27	0.0	18.05	0.05	3.81	0.0	176.713	0.136	0.413	0.001
		Uniform	p1	50.52	0.1	6.28	0.0	17.38	0.02	2.66	0.01	22.743	0.018	0.939	0.001
			p4	50.88	0.23	6.27	0.0	19.46	0.03	3.81	0.0	77.569	0.117	2.904	0.006
		Zipfian	p1	50.37	0.13	6.28	0.0	17.62	0.01	2.67	0.0	24.323	0.01	0.923	0.001
			p4	50.42	0.17	6.27	0.0	19.78	0.04	3.72	0.07	74.13	0.078	2.538	0.001
	Persistent	Sequential	p1	48.93	0.09	3.44	0.0	17.36	0.05	2.67	0.0	146.309	1.481	0.504	0.005
			p4	49.1	0.06	3.44	0.0	17.51	0.03	3.81	0.0	148.256	0.261	0.346	0.001
		Uniform	p1	49.92	0.04	3.44	0.0	17.22	0.04	2.66	0.01	21.726	0.021	0.897	0.001
			p4	50.24	0.13	3.44	0.0	19.09	0.03	3.81	0.0	72.721	0.087	2.702	0.004
		Zipfian	p1	50.04	0.09	3.44	0.0	17.46	0.04	2.66	0.0	23.255	0.024	0.883	0.001
			p4	49.1	0.1	3.44	0.0	19.22	0.03	3.74	0.06	68.847	0.142	2.347	0.005
Epoch and exact frequency based	Memory	Sequential	p1	48.83	0.26	7.36	0.0	17.23	0.03	2.67	0.0	165.673	0.138	0.619	0.002
			p4	48.8	0.08	7.36	0.0	17.47	0.01	3.69	0.0	175.767	0.016	0.374	0.0
		Uniform	p1	50.03	0.07	7.36	0.0	17.22	0.02	2.66	0.0	22.42	0.021	0.925	0.001
			p4	50.11	0.04	7.36	0.0	18.97	0.01	3.79	0.03	76.517	0.071	2.866	0.003
		Zipfian	p1	49.92	0.01	7.36	0.0	17.43	0.03	2.66	0.0	23.947	0.025	0.908	0.001
			p4	49.02	0.11	7.36	0.0	19.09	0.04	3.81	0.0	73.54	0.026	2.524	0.001
	Persistent	Sequential	p1	49.09	0.12	3.59	0.0	17.45	0.04	2.67	0.0	145.292	2.09	0.494	0.039
			p4	49.03	0.13	3.59	0.0	17.53	0.02	3.81	0.0	147.111	1.953	0.377	0.057
		Uniform	p1	49.94	0.12	3.59	0.0	17.1	0.38	2.65	0.01	15.533	10.712	0.641	0.441
			p4	50.2	0.05	3.59	0.0	19.1	0.02	3.81	0.0	72.168	0.529	2.683	0.018
		Zipfian	p1	49.97	0.03	3.59	0.0	17.45	0.07	2.66	0.01	21.799	2.348	0.828	0.088
			p4	49.3	0.1	3.59	0.0	19.38	0.03	3.76	0.04	68.884	0.025	2.35	0.002

Table 15: Resource usage by the *read* operations of the synthetic tests for the distribution *dist_kernels*.

				write + dist_kernels											
				Server				Client							
				CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
				AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No deduplication	-	Sequential	p1	16.33	0.01	2.27	0.0	34.19	0.57	2.43	0.0	1.382	0.031	379.505	7.931
			p4	16.33	0.01	2.27	0.0	31.43	0.11	2.8	0.0	1.317	0.012	358.485	3.346
		Uniform	p1	16.61	0.04	2.27	0.0	27.34	0.02	2.42	0.0	4.361	0.006	223.428	0.147
			p4	16.61	0.02	2.27	0.0	27.22	0.03	2.8	0.0	4.347	0.002	222.534	0.058
		Zipfian	p1	16.47	0.03	2.27	0.0	25.37	0.3	2.42	0.0	2.384	0.037	144.101	2.157
			p4	16.46	0.01	2.27	0.0	23.8	0.17	2.8	0.0	2.419	0.031	147.207	1.935
Only deduplication	Memory	Sequential	p1	48.19	0.24	2.97	0.0	22.31	0.08	2.68	0.0	0.859	0.0	362.97	0.302
			p4	48.26	0.09	2.97	0.0	19.93	0.05	3.82	0.0	0.863	0.001	364.608	0.286
		Uniform	p1	49.32	0.08	2.96	0.0	23.43	0.08	2.66	0.03	4.087	0.019	245.493	1.751
			p4	49.28	0.05	2.96	0.0	21.57	0.03	3.82	0.0	4.082	0.005	245.176	0.142
		Zipfian	p1	48.02	0.23	2.82	0.0	24.14	0.06	2.68	0.0	3.513	0.012	249.368	1.053
			p4	48.0	0.23	2.83	0.0	20.92	0.06	3.82	0.0	3.525	0.005	247.822	0.211
	Persistent	Sequential	p1	50.14	0.03	2.47	0.0	17.62	0.01	2.68	0.0	0.299	0.001	108.729	0.224
			p4	49.75	0.3	2.47	0.0	17.64	0.03	3.82	0.0	0.295	0.002	107.233	0.848
		Uniform	p1	50.11	0.03	2.46	0.0	18.58	0.02	2.68	0.0	1.418	0.006	102.182	0.424
			p4	50.14	0.12	2.46	0.0	18.52	0.02	3.82	0.0	1.419	0.007	101.972	0.469
		Zipfian	p1	49.81	0.02	2.43	0.0	18.28	0.02	2.68	0.0	1.188	0.001	97.369	0.099
			p4	49.75	0.09	2.43	0.0	18.35	0.02	3.82	0.0	1.215	0.005	98.602	0.398
Enclave based	Memory	Sequential	p1	49.76	0.23	3.09	0.0	18.0	0.04	2.67	0.01	0.392	0.001	145.905	0.208
			p4	49.6	0.19	3.08	0.01	18.0	0.02	3.82	0.0	0.384	0.012	143.214	4.46
		Uniform	p1	49.55	0.04	3.07	0.0	19.7	0.03	2.67	0.0	1.989	0.008	141.161	0.609
			p4	49.64	0.1	3.07	0.0	19.78	0.05	3.82	0.0	1.998	0.004	141.224	0.324
		Zipfian	p1	49.3	0.03	2.96	0.0	19.28	0.02	2.67	0.0	1.693	0.004	132.163	0.291
			p4	49.32	0.22	2.97	0.0	19.3	0.09	3.82	0.0	1.721	0.028	132.259	2.127
	Persistent	Sequential	p1	49.97	0.09	2.58	0.0	17.31	0.01	2.68	0.0	0.191	0.001	70.076	0.458
			p4	50.02	0.04	2.58	0.0	17.37	0.03	3.82	0.0	0.192	0.0	70.406	0.043
		Uniform	p1	50.05	0.03	2.57	0.0	18.09	0.02	2.67	0.0	0.954	0.002	68.333	0.123
			p4	50.05	0.07	2.57	0.0	18.14	0.02	3.82	0.0	0.947	0.007	67.709	0.48
		Zipfian	p1	49.93	0.06	2.55	0.0	17.9	0.02	2.67	0.0	0.818	0.004	64.516	0.328
			p4	50.03	0.03	2.55	0.0	18.02	0.02	3.82	0.0	0.843	0.004	65.473	0.275

Epoch based	**Memory**	**Sequential**	**p1**	49.89	0.08	3.2	0.0	17.81	0.01	2.67	0.0	0.36	0.001	131.671	0.28
			p4	50.01	0.18	3.2	0.0	17.89	0.02	3.82	0.0	0.36	0.001	131.493	0.407
		Uniform	**p1**	50.16	0.17	3.19	0.0	19.23	0.04	2.67	0.0	1.778	0.004	126.256	0.281
			p4	50.09	0.08	3.19	0.0	19.32	0.02	3.82	0.0	1.786	0.007	126.426	0.427
		Zipfian	**p1**	49.99	0.16	3.02	0.0	18.94	0.04	2.68	0.0	1.537	0.006	120.206	0.427
			p4	50.06	0.07	3.02	0.0	18.98	0.05	3.83	0.0	1.558	0.005	119.901	0.393
	Persistent	**Sequential**	**p1**	50.3	0.04	2.6	0.0	17.26	0.04	2.68	0.0	0.176	0.0	63.746	0.157
			p4	50.21	0.03	2.6	0.0	17.29	0.01	3.82	0.0	0.176	0.001	63.814	0.202
		Uniform	**p1**	50.22	0.03	2.59	0.0	17.89	0.01	2.67	0.0	0.855	0.0	61.429	0.017
			p4	50.26	0.11	2.59	0.0	17.99	0.04	3.82	0.0	0.859	0.001	61.521	0.053
		Zipfian	**p1**	50.3	0.03	2.54	0.0	17.77	0.01	2.68	0.01	0.76	0.001	59.94	0.109
			p4	50.15	0.14	2.54	0.0	17.83	0.02	3.82	0.0	0.771	0.002	59.862	0.139
Estimated frequency based	**Memory**	**Sequential**	**p1**	49.39	0.06	3.37	0.0	17.4	0.01	2.67	0.0	0.257	0.0	94.32	0.112
			p4	49.45	0.02	3.38	0.01	17.47	0.01	3.82	0.0	0.257	0.0	94.407	0.193
		Uniform	**p1**	49.61	0.26	3.34	0.01	18.44	0.04	2.67	0.0	1.287	0.004	91.921	0.305
			p4	49.56	0.08	3.35	0.01	18.49	0.05	3.82	0.01	1.289	0.004	91.949	0.304
		Zipfian	**p1**	49.39	0.08	3.1	0.0	18.21	0.01	2.68	0.0	1.14	0.004	89.784	0.282
			p4	49.43	0.23	3.12	0.0	18.34	0.01	3.82	0.0	1.159	0.001	89.898	0.037
	Persistent	**Sequential**	**p1**	49.83	0.03	2.64	0.0	17.07	0.01	2.67	0.01	0.139	0.001	50.286	0.274
			p4	49.74	0.02	2.64	0.0	17.09	0.01	3.81	0.02	0.138	0.001	49.886	0.303
		Uniform	**p1**	49.82	0.09	2.63	0.0	17.55	0.01	2.67	0.0	0.683	0.003	49.189	0.242
			p4	49.85	0.09	2.63	0.0	17.63	0.03	3.79	0.02	0.692	0.004	49.707	0.323
		Zipfian	**p1**	49.76	0.08	2.56	0.0	17.58	0.03	2.67	0.0	0.625	0.002	49.352	0.132
			p4	49.86	0.17	2.57	0.0	17.57	0.04	3.82	0.0	0.639	0.001	49.73	0.093
Epoch and exact frequency based	**Memory**	**Sequential**	**p1**	50.15	0.2	3.37	0.0	17.52	0.02	2.68	0.0	0.245	0.001	89.372	0.374
			p4	50.13	0.06	3.37	0.0	17.56	0.04	3.83	0.0	0.245	0.001	89.404	0.349
		Uniform	**p1**	50.03	0.06	3.33	0.01	18.51	0.01	2.77	0.01	1.225	0.003	87.739	0.19
			p4	50.17	0.13	3.34	0.0	18.61	0.0	3.87	0.03	1.234	0.002	88.128	0.098
		Zipfian	**p1**	50.09	0.14	3.1	0.0	18.35	0.01	2.77	0.01	1.093	0.001	86.064	0.072
			p4	50.02	0.13	3.11	0.0	18.48	0.03	3.93	0.0	1.106	0.003	85.809	0.271
	Persistent	**Sequential**	**p1**	50.19	0.09	2.64	0.0	17.13	0.01	2.67	0.0	0.141	0.0	50.095	0.01
			p4	50.2	0.06	2.64	0.0	17.18	0.03	3.78	0.0	0.14	0.0	49.977	0.095
		Uniform	**p1**	50.3	0.13	2.62	0.0	17.68	0.01	2.67	0.01	0.679	0.002	48.889	0.181
			p4	50.16	0.09	2.63	0.0	17.68	0.02	3.79	0.02	0.681	0.002	48.946	0.179
		Zipfian	**p1**	50.14	0.04	2.57	0.0	17.62	0.01	2.67	0.0	0.614	0.003	48.484	0.259
			p4	50.18	0.05	2.57	0.0	17.67	0.01	3.82	0.0	0.627	0.001	48.696	0.07

Table 16: Resource usage by the *write* operations of the synthetic tests for the distribution *dist_kernels*.

A.1.2.2 *Realistic experiments*

			Mail											
			Server				Client							
			CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
			AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No dedup.	-	**S1**	16.18	0.13	2.28	0.0	16.66	0.01	2.58	0.0	0.095	0.0	0.157	0.0
		S200	16.22	0.36	2.28	0.0	21.71	0.03	2.58	0.0	21.354	0.011	119.278	0.692
		S400	16.64	0.23	2.28	0.0	25.12	0.04	2.58	0.0	39.473	0.062	188.664	0.146
Only deduplication	Memory	**S1**	49.81	0.06	2.68	0.0	16.61	0.0	2.59	0.0	0.095	0.0	0.157	0.0
		S200	50.31	0.24	2.75	0.01	18.82	0.03	2.59	0.0	20.987	0.047	114.112	1.657
		S400	50.3	0.09	2.77	0.01	20.46	0.05	2.58	0.0	39.281	0.418	174.631	5.368
	Persistent	**S1**	50.05	0.06	2.33	0.0	16.65	0.01	2.58	0.0	0.095	0.0	0.157	0.0
		S200	50.59	0.01	2.41	0.0	18.71	0.02	2.58	0.0	20.903	0.051	92.326	3.661
		S400	50.46	0.12	2.41	0.0	19.03	0.06	2.58	0.0	28.011	0.408	97.278	3.508
Enclave based	Memory	**S1**	50.23	0.05	2.77	0.0	16.66	0.04	2.59	0.0	0.095	0.0	0.157	0.0
		S200	50.4	0.15	2.85	0.01	18.51	0.65	2.59	0.0	16.019	8.635	77.234	43.543
		S400	50.54	0.16	2.85	0.0	19.6	0.17	2.58	0.0	33.36	0.985	112.874	5.823
	Persistent	**S1**	50.15	0.07	2.44	0.01	16.66	0.04	2.59	0.01	0.095	0.0	0.157	0.0
		S200	50.31	0.01	2.52	0.0	18.31	0.01	2.58	0.0	17.686	0.098	63.16	0.266
		S400	50.34	0.08	2.52	0.0	18.33	0.01	2.59	0.0	18.853	0.024	62.224	0.108
Epoch based	Memory	**S1**	50.13	0.17	2.75	0.0	16.66	0.0	2.58	0.0	0.095	0.0	0.157	0.0
		S200	50.34	0.18	2.87	0.0	18.88	0.05	2.59	0.01	20.969	0.036	104.964	0.156
		S400	50.75	0.12	2.88	0.0	19.35	0.01	2.58	0.0	30.227	0.965	104.786	1.442
	Persistent	**S1**	50.09	0.24	2.44	0.01	16.63	0.03	2.59	0.0	0.095	0.0	0.156	0.002
		S200	50.38	0.07	2.5	0.0	18.33	0.06	2.58	0.0	19.131	0.999	64.673	1.225
		S400	50.38	0.08	2.5	0.0	18.32	0.04	2.58	0.0	19.825	0.09	63.486	0.109
Estimated frequency based	Memory	**S1**	50.24	0.04	2.76	0.0	16.65	0.0	2.58	0.0	0.095	0.0	0.157	0.0
		S200	50.48	0.05	2.89	0.0	18.76	0.02	2.58	0.0	20.831	0.024	84.542	0.389
		S400	50.35	0.01	2.9	0.0	18.87	0.06	2.57	0.0	24.899	0.449	82.205	2.93
	Persistent	**S1**	50.21	0.07	2.44	0.0	16.65	0.0	2.58	0.0	0.095	0.0	0.157	0.0
		S200	50.39	0.07	2.51	0.01	18.11	0.05	2.58	0.01	15.97	0.156	53.293	1.315
		S400	50.34	0.06	2.51	0.0	18.11	0.04	2.58	0.01	16.14	0.129	52.886	0.398
Epoch and exact frequency based	Memory	**S1**	49.94	0.11	2.76	0.01	16.64	0.03	2.58	0.01	0.095	0.0	0.157	0.0
		S200	50.58	0.32	2.87	0.0	18.64	0.05	2.58	0.0	20.825	0.153	82.409	3.038
		S400	50.71	0.06	2.88	0.0	18.79	0.05	2.58	0.0	24.759	0.276	80.411	2.358
	Persistent	**S1**	50.04	0.17	2.44	0.01	16.64	0.03	2.59	0.01	0.095	0.0	0.157	0.0
		S200	50.34	0.09	2.51	0.0	18.06	0.02	2.59	0.0	16.018	0.057	52.886	0.074
		S400	50.38	0.01	2.51	0.0	18.14	0.03	2.58	0.0	16.137	0.201	53.234	1.544

Table 17: Resource usage by the realistic tests for the trace *mail*.

			Homes											
			Server				Client							
			CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
			AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No dedup.	-	**S1**	16.18	0.27	2.28	0.0	16.65	0.0	3.17	0.0	0.004	0.0	0.082	0.003
		S350	16.19	0.16	2.28	0.0	17.1	0.02	3.32	0.0	0.803	0.002	8.116	0.114
		S700	16.19	0.23	2.28	0.0	17.22	0.01	3.33	0.0	1.147	0.002	10.841	0.14
Only deduplication	**Memory**	**S1**	50.12	0.05	2.66	0.0	16.65	0.0	3.17	0.0	0.004	0.0	0.081	0.003
		S350	49.98	0.19	2.77	0.01	16.91	0.03	3.32	0.0	0.784	0.001	7.866	0.045
		S700	50.11	0.18	2.8	0.0	16.95	0.03	3.33	0.0	1.12	0.002	10.672	0.052
	Persistent	**S1**	50.08	0.02	2.34	0.0	16.65	0.0	3.18	0.0	0.004	0.0	0.077	0.003
		S350	50.21	0.1	2.41	0.0	16.93	0.05	3.32	0.0	0.782	0.004	8.027	0.084
		S700	50.2	0.09	2.41	0.0	16.97	0.03	3.33	0.0	1.122	0.001	10.806	0.102
Enclave based	**Memory**	**S1**	50.31	0.1	2.76	0.0	16.63	0.01	3.17	0.0	0.004	0.0	0.074	0.0
		S350	50.33	0.12	2.87	0.0	16.93	0.02	3.32	0.0	0.783	0.001	7.939	0.093
		S700	50.17	0.18	2.89	0.0	16.92	0.01	3.33	0.0	1.12	0.002	10.761	0.141
	Persistent	**S1**	50.16	0.16	2.44	0.01	16.65	0.02	3.17	0.01	0.004	0.0	0.074	0.001
		S350	50.25	0.12	2.52	0.0	16.91	0.03	3.33	0.0	0.784	0.0	8.1	0.037
		S700	50.25	0.07	2.53	0.0	16.98	0.02	3.34	0.0	1.118	0.0	10.729	0.07
Epoch based	**Memory**	**S1**	50.35	0.17	2.75	0.0	16.64	0.0	3.17	0.0	0.004	0.0	0.074	0.002
		S350	50.41	0.2	2.85	0.0	16.89	0.0	3.32	0.0	0.784	0.0	7.939	0.031
		S700	50.37	0.08	2.89	0.0	16.95	0.02	3.33	0.0	1.12	0.002	10.732	0.015
	Persistent	**S1**	50.22	0.25	2.44	0.01	16.64	0.03	3.18	0.0	0.004	0.0	0.075	0.001
		S350	50.36	0.04	2.5	0.0	16.93	0.02	3.32	0.0	0.784	0.001	8.092	0.049
		S700	50.23	0.16	2.5	0.0	16.95	0.04	3.33	0.0	1.119	0.001	10.638	0.044
Estimated frequency based	**Memory**	**S1**	50.24	0.03	2.76	0.0	16.65	0.0	3.17	0.0	0.004	0.0	0.078	0.006
		S350	50.27	0.1	2.87	0.0	16.92	0.04	3.32	0.0	0.783	0.0	8.016	0.01
		S700	50.28	0.24	2.9	0.0	16.97	0.01	3.33	0.01	1.119	0.001	10.756	0.067
	Persistent	**S1**	50.29	0.1	2.44	0.0	16.65	0.0	3.17	0.0	0.004	0.0	0.077	0.003
		S350	50.24	0.01	2.51	0.0	16.94	0.01	3.32	0.0	0.783	0.0	8.034	0.061
		S700	50.14	0.05	2.52	0.0	16.97	0.01	3.33	0.0	1.118	0.001	10.716	0.143
Epoch and exact frequency based	**Memory**	**S1**	50.16	0.33	2.76	0.01	16.66	0.01	3.17	0.01	0.004	0.0	0.077	0.002
		S350	50.12	0.06	2.86	0.0	16.92	0.02	3.33	0.0	0.784	0.001	7.994	0.077
		S700	50.27	0.15	2.89	0.0	17.0	0.04	3.33	0.0	1.119	0.001	10.773	0.047
	Persistent	**S1**	50.12	0.28	2.44	0.01	16.64	0.03	3.17	0.01	0.004	0.0	0.077	0.003
		S350	50.15	0.09	2.5	0.0	16.92	0.03	3.32	0.0	0.79	0.011	8.093	0.029
		S700	50.28	0.16	2.51	0.0	17.0	0.03	3.33	0.0	1.121	0.002	10.807	0.16

Table 18: Resource usage by the realistic tests for the trace *homes*.

			Web-VM											
			Server				Client							
			CPU (%)		RAM (GB)		CPU (%)		RAM (GB)		RECV (MiB/s)		SEND (MiB/s)	
			AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV	AVG	DEV
No dedup.	-	S1	50.38	0.18	2.29	0.0	16.68	0.08	2.34	0.0	0.001	0.0	0.017	0.0
		S350	50.42	0.12	2.28	0.0	16.81	0.0	2.34	0.0	0.639	0.005	3.341	0.045
		S700	50.22	0.03	2.28	0.0	16.99	0.01	2.38	0.0	0.674	0.005	6.754	0.214
Only deduplication	Memory	S1	50.12	0.14	2.66	0.0	16.64	0.01	2.34	0.0	0.001	0.0	0.017	0.0
		S350	50.06	0.26	2.7	0.01	16.72	0.03	2.35	0.0	0.636	0.004	3.317	0.029
		S700	49.97	0.06	2.71	0.01	16.82	0.03	2.39	0.01	0.67	0.001	6.864	0.217
	Persistent	S1	49.99	0.12	2.34	0.02	16.63	0.02	2.34	0.0	0.001	0.0	0.017	0.0
		S350	50.04	0.16	2.4	0.01	16.73	0.03	2.35	0.0	0.633	0.005	3.428	0.161
		S700	50.13	0.06	2.4	0.0	16.84	0.0	2.39	0.0	0.664	0.004	7.009	0.287
Enclave based	Memory	S1	50.37	0.08	2.76	0.0	16.63	0.0	2.33	0.0	0.001	0.0	0.017	0.0
		S350	50.32	0.13	2.79	0.0	16.72	0.01	2.34	0.01	0.635	0.004	3.452	0.032
		S700	50.07	0.4	2.8	0.0	16.81	0.09	2.39	0.01	0.667	0.005	6.97	0.275
	Persistent	S1	50.37	0.31	2.44	0.0	16.65	0.01	2.33	0.0	0.001	0.0	0.017	0.0
		S350	50.13	0.14	2.51	0.0	16.74	0.03	2.35	0.0	0.637	0.0	3.469	0.131
		S700	50.4	0.03	2.5	0.0	16.86	0.0	2.38	0.0	0.67	0.0	6.944	0.118
Epoch based	Memory	S1	50.41	0.04	2.74	0.0	16.65	0.03	2.34	0.0	0.001	0.0	0.017	0.0
		S350	49.96	0.2	2.79	0.01	16.7	0.02	2.35	0.0	0.633	0.005	3.458	0.188
		S700	50.09	0.39	2.8	0.01	16.82	0.03	2.39	0.0	0.664	0.004	6.901	0.226
	Persistent	S1	50.27	0.16	2.43	0.0	16.67	0.03	2.34	0.0	0.001	0.0	0.017	0.0
		S350	50.23	0.23	2.48	0.0	16.74	0.01	2.34	0.0	0.638	0.0	3.641	0.059
		S700	49.99	0.23	2.5	0.01	16.79	0.1	2.39	0.01	0.667	0.005	6.881	0.122
Estimated frequency based	Memory	S1	50.26	0.09	2.76	0.0	16.64	0.0	2.33	0.0	0.001	0.0	0.017	0.0
		S350	50.11	0.08	2.8	0.0	16.72	0.0	2.35	0.0	0.635	0.005	3.607	0.073
		S700	50.37	0.22	2.8	0.0	16.86	0.01	2.39	0.0	0.67	0.0	7.083	0.12
	Persistent	S1	50.22	0.16	2.44	0.0	16.63	0.01	2.33	0.0	0.001	0.0	0.017	0.0
		S350	50.16	0.12	2.5	0.0	16.73	0.01	2.34	0.0	0.636	0.003	3.574	0.036
		S700	50.24	0.11	2.5	0.0	16.83	0.0	2.39	0.0	0.669	0.0	6.877	0.109
Epoch and exact frequency based	Memory	S1	50.27	0.2	2.75	0.0	16.67	0.04	2.34	0.0	0.001	0.0	0.017	0.0
		S350	50.26	0.23	2.78	0.0	16.74	0.0	2.34	0.0	0.632	0.004	3.475	0.177
		S700	50.01	0.07	2.79	0.0	16.86	0.01	2.38	0.0	0.664	0.005	7.104	0.126
	Persistent	S1	50.27	0.09	2.43	0.0	16.65	0.01	2.34	0.0	0.001	0.0	0.017	0.0
		S350	50.12	0.16	2.5	0.01	16.76	0.06	2.35	0.0	0.632	0.005	3.522	0.183
		S700	50.24	0.08	2.49	0.0	16.87	0.02	2.39	0.0	0.667	0.004	6.855	0.06

Table 19: Resource usage by the realistic tests for the trace *web-vm*.