Pesos: policy enhanced secure object store

Citation for published version:
Krahn, R, Trach, B, Vahldiek-Oberwagner, A, Knauth, T, Bhatotia, P & Fetzer, C 2018, Pesos: policy enhanced secure object store. in *Proceedings of the Thirteenth EuroSys Conference.*, 25, ACM, Porto, Portugal, Thirteenth EuroSys Conference, Porto, Portugal, 23/04/18. https://doi.org/10.1145/3190508.3190518

Digital Object Identifier (DOI):
10.1145/3190508.3190518

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the Thirteenth EuroSys Conference

Pesos: Policy Enhanced Secure Object Store

Robert Krahn[†], Bohdan Trach[†], Anjo Vahldiek-Oberwagner[‡], Thomas Knauth[★], Pramod Bhatotia[*], Christof Fetzer[†]

[†]TU Dresden [‡]MPI-SWS, Saarland Informatics Campus [★]Intel Corporation [*]University of Edinburgh

ABSTRACT

Third-party storage services pose the risk of integrity and confidentiality violations as the current storage policy enforcement mechanisms are spread across many layers in the system stack. To mitigate these security vulnerabilities, we present the design and implementation of Pesos, a Policy Enhanced Secure Object Store (Pesos) for untrusted third-party storage providers. Pesos allows clients to specify per-object security policies, concisely and separately from the storage stack, and enforces these policies by securely mediating the I/O in the persistence layer through a single unified enforcement layer. More broadly, Pesos exposes a rich set of storage policies ensuring the integrity, confidentiality, and access accounting for data storage through a declarative policy language.

Pesos enforces these policies on untrusted commodity platforms by leveraging a combination of two trusted computing technologies: Intel SGX for trusted execution environment (TEE) and Kinetic Open Storage for trusted storage. We have implemented Pesos as a fully-functional storage system supporting many useful end-to-end storage features, and a range of effective performance optimizations. We evaluated Pesos using a range of micro-benchmarks, and real-world use cases. Our evaluation shows that Pesos incurs reasonable performance overheads for the enforcement of policies while keeping the trusted computing base (TCB) small.

CCS CONCEPTS

• **Information systems → Cloud based storage**; • **Security and privacy → Trusted computing**;

KEYWORDS

Storage security, policy language, Intel SGX, Kinetic disks

1 INTRODUCTION

With the growth in popularity of Internet services, online data stored in datacenters is growing at an ever-increasing rate [16]. To tap into this vast market, many third-party online services such as Amazon S3 [1], Azure Blob Storage [7], Google Cloud Storage [27], or EMC Elastic Cloud Storage [14] offer consolidated storage at scale. These storage systems are ubiquitously used by Internet services to store data with a very high degree of availability and reliability at low cost.

At the same time, due to the complexity associated in designing large-scale storage systems, the risk of confidentiality and integrity violations has increased significantly. Many anecdotal pieces of evidence show that software bugs, configuration errors, and security vulnerabilities pose a serious threat to these storage systems [4, 11, 22, 28]. Furthermore, untrusted third-party cloud platforms expose an additional risk of unauthorized data access by a malicious administrator [59].

These threats are due to the fact that modern storage systems are quite complex [4, 52, 58, 73]. Currently, storage systems are characterized by multiple layers of software and hardware stacked together to enable a data path from the application to the storage persistence layer. This transit data path requires multiple layers of manipulation from the application-specific object database, through the POSIX interface, filesystem, volume manager, and drivers [30]. As a result, the access policies for ensuring confidentiality, integrity, and access accounting are scattered across different code paths and configurations. Thereby, the enforcement of access policies is carried out by many layers in the software stack; thus, exposing the data to security vulnerabilities. Furthermore, since the stored data is outside the control of the data owner, the third-party storage platforms provide an additional attack vector. The clients currently have limited support to verify whether the third-party operator, even with good intentions, can handle the data with confidentiality and integrity guarantees.

To address these challenges, we present Pesos, a Policy Enhanced Secure Object Store (Pesos) for untrusted third-party storage services. Pesos allows clients to specify per-object security policies concisely and separately from the remaining storage stack, and enforces these policies by securely mediating the I/O in the storage layer. Further, Pesos provides *cryptographic attestation* for the stored objects and their associated policies to verify the policy enforcement.

To achieve these goals, Pesos provides a unified interface for the client based on a *declarative policy language* to efficiently express a wide range of storage policies regarding data integrity, confidentiality, access accounting, and many other workflows. The declarative abstraction is imperative to minimize the complexity in specifying and auditing policies via a single unified interface. In particular, a policy specifies

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the conditions under which an object can be read, updated or have its policy changed; this may, in turn, depend on client authentication, object metadata, content, or a signed certificate from a trusted third-party. Pesos stores the specified policy as part of the object metadata and ensures that each access to the object complies with the associated policies.

The policy enforcement is carried out by Pesos through a combination of two trusted computing technologies: Intel Software Guard Extensions (SGX) [2, 48] for trusted execution environments (TEE) and Kinetic Object Storage [54, 63] for trusted storage. The basic idea is quite straightforward: Pesos carefully structures a policy-compiler, its binary-format interpreter, per-object policy metadata, and the enforcement logic into a single layer of the storage stack. Since the policy enforcement is unified in this single layer—i.e., the TCB for policy enforcement is reduced to the single layer as opposed to multiple code paths and configuration files—the security scope solely reduces to protecting this layer. Pesos protects this TCB layer by leveraging the recently released Intel SGX ISA extensions for TEE. From within the protected layer, Pesos connects directly to the Kinetic disk through an encrypted Ethernet connection. The secure connection terminates within the disk’s controller and allows Pesos to transfer the data from the TEE directly onto the drives without any intermediate layers in the storage stack.

Pesos’ end-to-end workflow is as follows: When Pesos is started, an attestation service verifies that Pesos is executed on the correct hardware and that the binary executable has not been altered. Afterwards the attestation service provides authentication and encryption keys used by Pesos during the runtime. Pesos uses the provided information to connect to a set of Kinetic disks and acquires exclusive access by removing any other user accounts on the disks. Clients can now connect to Pesos and transfer objects and policies. The object policies are compiled through a policy-compiler to produce a binary format, which allows for fast permission checking. Upon upload, objects are paired with policies based on the client’s request. Future requests to objects with attached policies require a positive evaluation by the policy checker with regard to the anticipated operation on the objects.

We implemented a fully-functional prototype of Pesos based on the Intel Skylake micro-architecture with SGX ISA extensions and Seagate Kinetic drives. Furthermore, we implemented four important use-cases based on Pesos (§5), including a content server, a verisoned object, a time-protected object storage system, and a mandatory access logging (MAL) storage system.

We evaluated Pesos using a wide range of micro-benchmarks, and aforementioned use-cases using YCSB workloads [9]. To realistically asses the performance of our system, in addition to using Seagate Kinetic HDD, we further examined the performance of Pesos with the Kinetic disk simulator. Our evaluation shows that Pesos achieves a peak performance similar to the native version: throughput is at least 85% of native and often better. Latency is less impacted and within 5% of native before Pesos becomes overloaded.

2 BACKGROUND

Pesos uses shielded execution in combination with Kinetic storage drives to build a secure policy-based object store. In this section, we first provide the relevant background about these building blocks. Thereafter, we explain our threat model.

2.1 Shielded Execution

Shielded execution [3, 5, 65, 80] provides strong confidentiality and integrity guarantees for legacy applications running on untrusted platforms. Our work builds on Scone [3]—a shielded execution framework based on Intel Software Guard Extensions (SGX) [2, 10, 32, 48].

Intel SGX, released as part of the Skylake architecture, extends the x86 instruction set with a processor mode and instructions to provide a trusted execution environment (TEE). SGX provides a hardware-protected memory region called *secure enclave*, which contains the trusted code and data, in combination with a call-gate mechanism to control entry and exit into the trusted execution environment. The enclave memory can only be accessed by the enclave it belongs to, i.e., the enclave memory is protected from concurrent enclaves and other (privileged) code on the platform. SGX protects the application against a powerful adversary controlling the entire system stack, including the OS or the hypervisor, and is able to launch physical attacks, such as performing memory probes.

However, a significant drawback of the SGX architecture is the limited enclave memory of 128 MB (only 96 MB are available to applications). To alleviate the problem, SGX supports paging of enclave memory to regular system memory, while maintaining the memory’s confidentiality, integrity and protecting against replay attacks. However, the enclave memory paging can be significantly expensive (2×—2000×) [3].

For buidling SGX-based applications, Intel software development kit (SDK) provides tooling and software infrastructure. Essentially, the application developer splits the application into a trusted and untrusted part and defines the interface between them. Code generators create stubs and boilerplate code to move data between the trusted and untrusted portion of the program. This programming model centered around pieces of application logic (PAL) forces the application developer to think carefully about the functionality to include in the trusted environment, increasing the development effort significantly. To ease the development and deployment of SGX-based applications, a number of shielded execution frameworks have been built based on SGX to provide strong confidentiality and integrity properties for running legacy applications [3, 5, 65, 80].

Pesos builds on the Scone shielded execution framework [3] to run the Pesos controller inside an SGX enclave. More specifically, we leverage Scone's support for remote attestation and user-level multithreading, and container support for the application deployment. We have adapted and optimized many of Scone's features in the context of Pesos (§4.6).

2.2 Kinetic Storage

In the cloud, applications are increasingly adopting the *object storage* model to store data [1, 20, 50]. For Pesos, we use a particular implementation of object storage that removes many of the traditional software and hardware pieces involved in a "classic" object storage implementation. Figure 1 (left) illustrates the traditional storage stack with the many hardware and software layers involved between an application issuing a write request and the data being written to physical media. The Kinetic storage drive, shown in Figure 1 (right), removes some of these layers entirely and consolidates others. The Kinetic drives bundle a traditional hard drive with a System-on-Chip (SoC) and an Ethernet interface. The disks are directly connected to the existing Ethernet network and applications use the Kinetic disks via an HTTP/S interface.

In the context of Pesos, Kinetic disks are particularly compelling. In combination with the trusted Pesos controller based on shielded execution running inside an Intel SGX hardware enclave, the disks represent the only other trusted component in our design. With Kinetic disks deployed in a cloud datacenter and exclusively accessed by the Pesos controller, the surrounding hardware and software cloud infrastructure remains outside the trusted computing base (§2.4). Because the disks are exclusively accessed via encrypted and authenticated communication channels, potential man-in-the-middle attacks are thwarted. Further, Pesos transparently encrypts objects using AES-GCM before storing them.

Section 3 and Section 4 will detail how Pesos utilizes the unique characteristics of Kinetic disks to provide a secure object storage in the cloud. In our prototype implementation, we build on the Kinetic disks from Seagate. However, Pesos' design is general enough to work with Ethernet drives from other vendors as well. For instance, Samsung and Toshiba also announced to work on similar disks [56, 76].

2.3 Policy-based Storage Systems

In today's systems, confidentiality and integrity depend on the intricate synergy of the entire software stack. Applications or databases enforce invariants (i.e., user authentication), file systems enforce access control, OS/VMM isolate data and special purpose storage systems guard persistent data. While protection in upper layers may depend on application-specific information (such as a user session), lower layers enforce storage invariants (such as append-only files). Compared to application-layer techniques, lower layers offer a more comprehensive protection against circumvention.

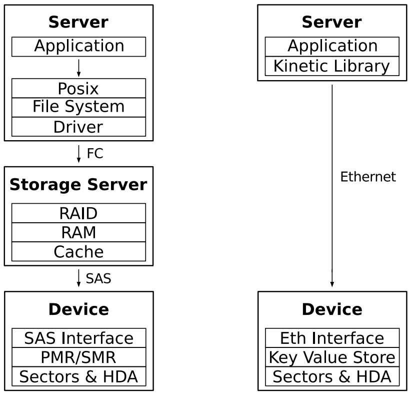

Figure 1: Traditional block storage (left) vs. Kinetic storage (right) [37].

Storing data in untrusted cloud storage services increases the software stack's complexity and requires trust in a third-party organization. While commercial and research systems [5, 35, 61, 80] exist to protect computation in an untrusted cloud environment using, e.g., SGX, data protection relies on encryption and Merkle trees to provide confidentiality and integrity. More complex guarantees have to be enforced by the application which spreads the enforcement logic and increases the burden on the developer. Also, cloud providers and in particular their employees may circumvent any application enforcement. Hence, invariants such as access logs or a time-based data release are out of scope for such storage systems. Pesos addresses these challenges by providing a general technique to associate policies to data objects and enforces these policies upon access.

In this respect, policy-based access control to secure storage systems is receiving increasingly more attention [23, 47, 66, 75]. The state-of-the-art system Guardat [82] protects the data confidentiality and integrity by enforcing client specified security policies. Guardat enforces policies at the lowest level within the storage stack to minimize the risk of circumvention. Pesos adopts this approach and extends Guardat. Instead of operating at the block-layer, Pesos operates on the common key-value interface [12, 13, 18, 86]. Whereas Guardat assumed a physically protected machine room, Pesos relies on trusted hardware and a trusted execution environment allowing its deployment in the cloud. Furthermore, Guardat couples the data storage and policy enforcement controller in a single machine. Using Intel SGX, Pesos separates the data storage and the controller offering better scalability and flexibility.

2.4 Threat Model

We assume a typical SGX threat model (e.g., [5]) in which we trust the Intel processor and its SGX implementation. We also trust the Kinetic disk. Everything else in the cloud provider’s hard- and software stack is untrusted.

Deviating from previous work, we do trust the cloud provider with physical security. In particular, we assume that an adversary cannot mount physical attacks on the otherwise trusted Kinetic disk. We believe this is a reasonable assumption since access to cloud data centers is strictly regulated.

We disregard physical attacks on the Kinetic disk since interposing between the system-on-chip and the drive controller may compromise security. Since clients communicate with the disk over a mutually authenticated and encrypted channel, we view physical attacks as the only practical attack vector. While Pesos can detect if entire disks are replaced (a type of rollback attack) based on each disk’s unique X.509 certificate, physical attacks may rollback state at a finer granularity, e.g., for single objects. Confidentiality of objects is guaranteed because Pesos encrypts objects before sending them to the disk. We consider side-channel attacks on SGX out-of-scope even though it is currently an area of active research (e.g., [29, 84]).

3 DESIGN

Pesos is a policy enhanced secure object store designed to operate in untrusted cloud environments. Clients connect to the Pesos controller to issue storage operations such as create, delete, and update objects. An object is referenced through a key. The key and object are essentially arbitrary sequence of bytes to Pesos. Each operation modifies the state of an object in its entirety, i.e., partial updates are not supported. (We do support ACID transactions for updating multiple objects together.) The connection between a client and the Pesos controller is established over a mutually authenticated channel after performing remote attestation. This ensures that the client is connected to the correct Pesos controller, while the controller uses the client’s certificate to perform access control. We explain the controller (§3.1) and the request flow (§3.2) in more detail later in this section.

Besides securely storing objects, Pesos offers the ability to associate policies with objects (§3.3) in a 1:M relationship, i.e., one policy to many objects. The Pesos controller unifies the policy enforcement. For each operation, the controller checks the associated policy of the accessed object. An operation only succeeds, if the policy complies. Pesos’ unique advantage is the ability to perform this policy compliance checking in an untrusted cloud environment by running inside a hardware-enforced trusted execution environment (TEE).

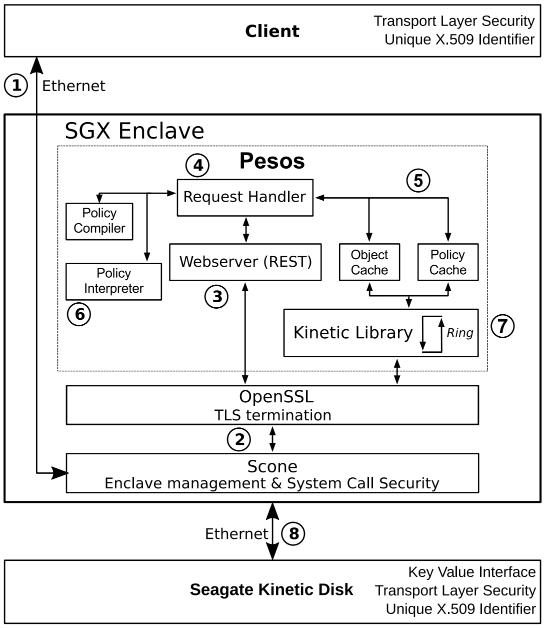

Figure 2: Pesos’ general architecture.

3.1 Pesos Controller

The central component of Figure 2 illustrates the main parts of the Pesos controller. The controller runs in the TEE based on Intel SGX [3, 5, 65, 80]. Clients communicate over a standard mutually authenticated encrypted channel, i.e., Transport Layer Security (TLS), with the controller. Importantly, the channel is terminated inside the TEE. At no time is the data exchanged between the client and the controller visible in clear text to any outsider.

The client sends HTTP requests over the trusted channel to the controller. The controller includes a small web server [53] to perform the connection handling and HTTP request parsing. The web server passes individual requests over to the request handler, which implements the actual request logic. The request handler, in turn, invokes the policy engine (compiler and interpreter) and various caches to speed up the request handling. If the object or policy is not present in the cache, the controller uses a library to communicate with the Ethernet-attached Kinetic disks over an authenticated channel.

Bootstrap process. To incorporate the hardware features provided by Intel SGX, we built Pesos based on Scone [3]. Besides efficiently handling system calls, the Scone framework includes a service for remote attestation and secure deployment of binaries. For remote attestation (RA) an enclave generates a signed measurement representing its identity [2, 40]. A third party can verify the enclave’s genuineness using the signature and inspect particular enclave properties by examining the measurement. The Scone attestation service, created to launch SGX-based applications in large-scale environments, incorporates RA and can securely launch an application inside an enclave. Only after successful attestation, the application is provided with the encrypted secrets required at runtime. For Pesos, the necessary information at application start-up time includes the TLS key-pair as well as credentials to access the disks.

In our design, the Pesos controller uses a configurable set of Kinetic disks to persistently store objects. As the very first step when bootstrapping the system, the controller takes exclusive control of all its assigned disks. It removes all existing user accounts and sets up a single administrative account which only this Pesos instance can access. This effectively locks out any other user, including the cloud provider. While the current prototype uses a static configuration, support for dynamically adding and removing disks to a controller instance can be added in the future (e.g., using consistent hashing [42]).

Session context. When a new client connects, as determined by its certificate, the controller creates a *session context*. The session context stores per-client soft-state required to execute requests, such as the state of asynchronous requests and policy-related metadata. The session context persists past a disconnect and only expires some time after. If a client reconnects while its session context still exists, it is reused.

Request modes. Pesos supports synchronous and asynchronous requests, while the later is only available where it is sensible (e.g., a put or a transaction). For synchronous requests, the client receives a response only once the handler is done with the request. For asynchronous requests, the client immediately receives a response. Within the response, there is an identifier that the client can subsequently use to inquire about the status of its asynchronous operation. Supporting both operational modes allows the client more flexibility to implement its application logic.

Policies. The client submits policies to the controller in a human-readable format. Afterwards, the policy is converted into a compact binary representation by the *policy compiler*.

The policy compiler uses the integrated lists of available predicates, literals and function implementations to parse and compile the submitted policy. The policy compiler relies on Flex [21] for lexical analysis and Bison [25] for input tokenization. Upon successful compilation of a policy, Pesos creates a unique policy identifier that is sent to the client. To accelerate policy checks, recently created policies are held in the *policy cache*. For persistence, the policies are also stored on the Ethernet-attached Kinetic disks.

After the successful submission of a policy, clients can start to associate installed policies with objects by including the policy identifier in the put request for objects. If the client accesses an object with an associated policy, the *policy interpreter* determines if the access is granted. In case the access is denied due to a policy violation, the client is notified through an appropriate response. The *policy interpreter* also determines if a client may associate a new policy to an existing object based on the currently associated policy.

Currently, Pesos relies on the client to manage, track and assign policies. In the future, we plan to extend the API towards policy management within Pesos. With regard to addressing clients or user-groups within policies, we designed the policy language to support integration of third-party services through certificates. A policy may state that the client has to present a certificate including group membership information that is signed by a specific authority.

Object cache. The *object cache* is a global in-memory data structure that allows Pesos to quickly fetch previously written objects. Additionally, for use cases that involve content-based policy checks (§5.4), the object cache supports the policy interpreter with fast data lookups. For future work, we are investigating support to securely store cached objects in untrusted memory (§2.1) to increase the effective cache size without enclave memory paging.

Kinetic library. Interaction with the Kinetic disks happens through an adapted C library originally provided by Seagate [36]. The library provides an abstract interface to the disks, hiding the details of the underlying communication protocol (based on Google Protocol Buffers). Requests and responses are decoupled from each other by using a ring buffer and a thread pool to receive messages from the disk. In the future, it may be beneficial to further customize the library to the specifics of the SGX framework to achieve better performance.

3.2 Pesos Workflow

We next explain the request/response flow using Figure 2. Initially, the *client* establishes a persistent connection to Pesos through HTTPS (step ①). The client's request is first handled by the Scone framework and passed to the *OpenSSL library*. The API of the SSL library has not been altered, and connection establishment through the integrated *webserver* is equivalent to non-SGX implementations (step ②). After the *webserver* receives a client request, it invokes the message processing in the *request handler* (step ③). The *request handler* parses the message (REST) and extracts the type and parameters of the client's request; for example, method: put, key: name, value: Bob (step ④). Based on the extracted method, corresponding sub-routines are called to execute the request. During the execution of a request (e.g., put), the request handler queries the *policy cache* for an existing policy to the object's key (step ⑤). Upon a cache hit, the *policy interpreter* (step ⑥) uses the policy and information from the client's request to derive whether the request complies with the policy. The request handler, if the permission is granted, executes the request (put) by, e.g., storing the key-value pair in the *object cache* (step ⑤). Objects and policies are additionally stored on disk (write-through semantic) using the *Kinetic library* (step ⑦). Messages to the disk are handled, similar to client requests, through the *OpenSSL library* and the Scone framework (steps ②, ⑧). The *request handler* parses responses from the disk and creates the corresponding responses for the client requests. Responses may be sent to the client prior to the interaction with the disk if (a) the policy check failed, or if (b) Pesos' asynchronous API is used. Requests that store a policy within Pesos take a similar path with the exception that Pesos compiles the policy using the *policy compiler* before storing the policy in the cache and on the disk.

3.3 Pesos Policy Language

Pesos' declarative policy language concisely specifies a request permission to read, update or delete an object. Updating the policy of an object is a special case of the object update request while maintaining the same content. Our policy language supports a wide range of use-cases (§5) while relying only on a basic set of predicates.

Declarative languages have been widely used as policy languages (e.g., Guardat [82], Thoth [17], Binder [15]). More specifically, Pesos' policy language is adapted based on Guardat [82]. In principle, the policy engine integrated in Pesos is equally powerful as in Guardat. However, the policy languages of Pesos and Guardat differ by the number of policy predicates that have been enabled in Pesos to support object-based applications. These applications access the data through a coarse-grained key-value API, where they always read and write objects as a whole. Guardat offers, in comparison, data access based on the POSIX file API (e.g., byte-offset and block addresses) for a fine-grained management. As such, policy predicates that make use of the POSIX file API were not integrated into Pesos as they are not applicable to objects. If fine-grained manipulation of objects is desirable in the future, we can add the related policy predicates offered by Guardat.

Each Pesos policy consists of three permissions (read, update and delete) to control confidentiality, integrity and when an object name can be reused. Permissions take the form of *perm : condition.* A condition is a disjunctive normal form of predicates. Table 1 describes the semantic of each predicate. A permission successfully evaluates, if the disjunctive normal form of predicates is successfully evaluated. Arguments of predicates are variables or explicit values. Variables (start with uppercase letters) stand for arbitrary values and are set when first used. The language supports five value types (integer, string, hash, public key, tuples). Tuples take the form of $key(v_1, ...)$, where key is a string and v_1 to v_n can be one of the five types.

An example access control policy is shown below in which user Alice can read the object, user Bob may update the object and the admin can delete the object.

Predicate	Meaning
Relational predicates	
eq(x,y)	x = y
le(x,y)	x <= y
lt(x,y)	x < y
ge(x,y)	x >= y
gt(x,y)	x > y
Certificate predicates	
certificateSays ($a, f, key(v_1, ...)$)	Authority a certifies tuple $k(v_1, ...)$ given freshness f
Session predicates	
sessionKeyIs(k)	Client is authenticated with key k
Object predicates	
objId(obj, id)	Compares or sets object id for object obj
currVersion(obj, v)	Compares or sets version v of object obj
nextVersion(v)	Compares or sets the version argument v of a put/update request
objSize(obj, v, s)	Compares or sets size s of obeject obj in version v
objPolicy(obj, v, ph)	Compares or sets the hash of the policy ph associated with object obj in version v
objHash(obj, v, h)	Compares or sets hash h of object obj in version v
objSays ($obj, v, key(v_1, ...)$)	Compares or sets contents of object obj (version v) with tuple $key(v_1, ...)$

Table 1: Pesos' policy language predicates.

$$\begin{aligned} &read :- sessionKeyIs(K_{\texttt{alice}}) \\ &update :- sessionKeyIs(K_{\texttt{bob}}) \\ &delete :- sessionKeyIs(K_{\texttt{admin}}) \end{aligned}$$

More complex policies are described in the case studies of section (§5); for example, access to objects is restricted within a time window, every version of an object is maintained, or accesses to objects require prior logging. To support such policies, predicates in the Pesos policy language provide access to information about objects, session and external certified facts from cryptographic certificates. Besides the version or size of an object, permissions also reason about the content (objSays) or hash (objHash) of the contents of an object. This allows policies to rely on state and alter their behavior based on prior object updates as described in the mandatory access logging use-case (§5.4). A client authenticated session is identified by its public key in sessionKeyIs. External facts such as time are queried using the predicate certificateSays which relies on a client to provide a cryptographically signed certificate to Pesos. The predicate certificateSays may also be used to establish a chain of trust to a certificate authority.

4 IMPLEMENTATION

Pesos (without third-party libraries) consists of 23,943 lines of code with the following distribution: the REST interface 8, 167, the policy handling and compiler 6, 355, and lastly, the controller 9,421. Third party libraries include OpenSSL, the SGX framework, and the Kinetic disk library. The policy interpreter, compiler and caches have been adapted based on previous work [82] to Pesos' unique environment and to comply with the semantics of the key-value interface.

In total, the statically linked executable is 16 MB large, of which 15 MB are loaded into the enclave. Code external to the enclave (e.g., enclave management and system call interface) occupies the remaining 1 MB.

4.1 Pesos API

Pesos does not require clients to use a special library to access its functionality. Instead, we solely rely on a regular REST interface on top of HTTPS. The client sends all request data in the HTTP POST request, and the TLS library provides a secure channel to Pesos. This interface allows a large variety of applications to use Pesos without a need for external components.

A Pesos POST request contains at most four parameters: a method, a key (part of the URL), a value, and a policy identifier. The *method* parameter defines the function that is called within the application to handle the key and the value. Depending on the method, the *key* identifies an object or a policy. The *value* parameter contains an object or policy data. When the clients upload a new object, they attach a previously stored policy to the object through the *policy* parameter.

Asynchronous interface. Pesos features a synchronous communication scheme for requests that return immediately, reporting operation results after the request has been completed. Additionally, an asynchronous interface is available for requests which would cause a long delay if the client would wait for their completion. Instead, upon receiving an asynchronous request, Pesos will immediately respond with an HTTP acknowledgment that includes a unique identifier. Clients may query Pesos for the result of the asynchronous request using this identifier. Due to the limited available enclave memory, Pesos stores the results of the last 2048 requests, while discarding older ones. The status and result of a request are stored in memory within the client's session context. The described asynchronous interface takes into account that write operations to the disks are slow, and allows the client to progress while also having an option to verify the result of an operation. Put, update and delete requests may be executed asynchronously. Other requests, like data retrieval (GET) or session management, are always synchronous.

The communication mode directly influences fault tolerance: If the Pesos controller crashes during an asynchronous operation, the API offers no direct notification to the client. The client has to query Pesos to learn that the last request has no result and re-transmit the initial request.

Session management. Pesos strongly relies on secure and mutually authenticated communication channels between the client and Kinetic disks. When a client connects to Pesos, it presents a TLS certificate to authenticate itself. Throughout the client's session and also afterwards, Pesos uses the certificate to map data to a client. For example, the buffered results of asynchronous write requests are organized based on the client's certificate.

4.2 Cache Management

Pesos uses caching and buffering to improve various aspects of the object store:

(1) Result buffer (final results for persistent operations)
(2) Buffer for pending operations
(3) In-memory storage for policies
(4) In-memory storage for session keys
(5) Buffer for transactions

The cache implementation approximates a least-frequently-used eviction policy. Pesos maintains separate memory regions for different request types (object, policies, and indices).

Due to the asynchronous communication scheme used in Pesos, it is necessary to incorporate a short-term result buffer. Considering a write request issued by a client, Pesos immediately replies with an acknowledgment (HTTP 200) that includes an operation identifier. Afterwards, Pesos will handle this request in parallel with others, and execute the operation on the disk. When the disk has finished the operation and returned a result, Pesos stores the result in the result buffer. The client that issued the request can ask Pesos for the result using the operation identifier.

When a client sends a read request, Pesos first has to fetch the corresponding policy to verify whether the client is allowed to access the object. In this case, Pesos first consults the cache. On a cache miss, the policy is read from the disk. Some policies require additional disk accesses, for example, to fetch other objects. To avoid a performance hit in this case, we cache objects accessed during policy evaluation. Then, in case of the permitted read request, Pesos looks up the object in the object cache before doing a disk read, which can also result in a cache hit. This way, we can eliminate multiple disk accesses that would have happened serially otherwise.

The enclave page cache (EPC) available to applications is limited to 96 MB in current SGX implementations [10]. The SGX kernel driver transparently transfers pages between the EPC and main memory (at additional cost) for SGX applications that exceed the EPC size. We restrict Pesos to not use more than the EPC size for its internal data structures and caches. Per connected client, Pesos allocates a session object with a default size of 30 KB. The global cache for policies is bounded to 5 MB by default. Additionally, Pesos keeps a cache of used object keys to improve performance (600 KB).

Buffers for currently requested objects and internal buffers of OpenSSL and Kinetic library consume the remaining memory. During start-up Pesos pre-allocates message buffers to hold objects for the transformation from the Kinetic library's internal data representation (Google Protocol Buffers) to the client's format (HTTP). The message buffers occupy the largest chunk of EPC memory, because they need to keep up to 1 MB large objects, and are independently allocated by multiple connection-handling threads.

4.3 Kinetic Library Changes

Seagate provides Kinetic disk client libraries for different programming languages. These libraries contain features for simple session management, data manipulation, replication management, device status monitoring, etc. We have used the C client library to develop Pesos.

We have applied numerous optimizations to the library to better exploit asynchronous system calls and userspace threading. We have replaced pipe-based thread synchronization with concurrent data structures to reduce the communication between cores in some common situations. Also, we reduced the number of service threads spawned, and the number of request/reply structures preallocated to reduce EPC usage.

4.4 Transaction Interface

Pesos provides a transaction interface to ensure atomic updates to multiple objects wrapped in a transaction with full ACID semantics. As distributed transactions and high network scalability are not required, we implement them using a simple and high-performance algorithm. In case distributed transactions are required, an additional transactional layer such as [74] can add them on top of Pesos. We rely on replication to recover from disk crashes, as we have no access to the internals of the Kinetic disks. Some advanced features of Pesos, such as append-only storage and mandatory access logging, are not supported inside transactions. We provide the following operations in our API: `createTx`, `abortTx`, `commitTx`, `addRead`, `addWrite`, and `checkResults`.

Pesos uses a modified variant of the VLL locking algorithm [57] tailored towards our anticipated use-cases. VLL tries to lock all requested objects prior to executing the transaction. While transactions that acquired all locks successfully execute immediately, blocked transactions remain in the transaction queue where they eventually move to the front of the queue. If the queue is not empty, the transaction at the front of the queue is unblocked and executed. VLL's design guarantees that once a blocked transaction reaches the queue's front all its keys are now unlocked. Deviating from the original VLL implementation, Pesos maintains a small data structure for storing keys and locks. This is required since the original implementation was designed for (in-memory) database systems.

We assume that only a fraction of all keys will be accessed via the transaction interface. Therefore, we allow mixing transactional and non-transactional accesses to keys. As an optimization, Pesos clients can use the non-transactional API to access the same values as the transactional interface does, including locked values (thus, we do not enforce concurrency control strictly). If such accesses overlap in time, the outcome is unspecified. Ensuring that such accesses do not overlap is the responsibility of the Pesos client. Non-overlapping accesses can, for example, be trivially ensured using the policy language. In the current version of Pesos, all policy language checks are supported within transaction, but we allow only simple object operations. Removing this restriction is part of our future work.

4.5 Replication Interface

Pesos provides a set of commands to clients for securely managing the replication and migration of objects across a set of Kinetic disks. Pesos regards a write operation and its replications as one request. Therefore, the result of a client's request—that Pesos stores in the client's session context—is successful only if all the replicas persist the value.

Pesos uses the following replication placement algorithm to avoid metadata synchronization between instances and to simplify the design: We map objects to disks through a deterministic hash function that takes the object's key and a list of available disks as input. A position in the list of disks is selected based on the object's hash. If replication is enabled, additional disks will be selected from that list starting at the position after the previously selected disk. If an object A is stored on disk D_i, its replicas are stored on disks D_{i+1}, D_{i+2}, ..., D_{i+N-1} where N is the replication factor. Upon disk failure, Pesos selects the next available disk to access the replica. While this approach may increase the load on some disks, depending on the distribution of keys, it does not require Pesos to keep any replication-related metadata.

4.6 Optimizations

We next describe some optimizations applied to the Pesos architecture to improve the performance.

Multithreading support. One of the restrictions of the current SGX implementation is the necessity to specify the maximum number of threads inside the enclave during the build process. This limits the flexibility of SGX enclaves in situations where this number is unknown, and allocate a number of data structures per each thread. We circumvent this restriction by using application-level scheduling provided by Scone.

In order to implement userspace threading, we multiplex userspace threads onto each enclave hardware thread. A userspace thread always runs to the next preemption point and then switches back to the internal scheduler. In our system, only system call submissions are preemption points as only they can be supported without compiler instrumentation.

Because we switch to another thread instead of waiting for a system call result, we further improve the efficiency of the system calls. In order to exploit this potential improvement, a sufficiently large number of threads have to be running. Pesos achieves this through multiple threads handling client connections and multiple threads interacting with the Kinetic disks via the Kinetic client library.

Memory management. The current version of SGX requires the application to specify the enclave memory range during the enclave initialization. It provides no mechanisms for modifying enclave memory range during the enclave execution. On the other hand, POSIX applications commonly use facilities that dynamically modify their address space, namely the `mmap` and `munmap` system calls.

To overcome this issue, Scone preallocates all code, data and heap memory when creating the enclave. We implement a simple bitmap-based memory allocator for enclave memory, which emulates POSIX-required `mmap`/`munmap` semantics. Our `malloc` implementation uses this allocator to request or return memory. We note that memory protection features of POSIX can only be provided in future versions of SGX, and all types of accesses, i.e., read, write and execute, to the preallocated pages are currently allowed.

The amount of EPC memory is limited. Current processors provide up to 96 MB of end-user-usable EPC memory. Aggressive caching and inefficient memory usage can lead to page accesses that belong to the enclave address space, but do not fit into the EPC. In this case, SGX-protected pages will be swapped out by the kernel paging mechanism. Encryption of the swapped pages provides confidentiality and storing a Merkle leaf hash inside the enclave ensures integrity.

I/O interface. To carry out I/O operations, Pesos must invoke system calls. A common way to do this is trap-based: an application sets up system call arguments in CPU registers and executes an instruction that transfers control to the OS system call handler. These control transferring instructions are forbidden inside SGX enclaves, forcing the application to exit and re-enter the enclave to execute a system call. In practice, enclave exits incur prohibitively large overheads. To overcome this issue, we use an asynchronous system call interface as proposed by FlexSC [69] and adopted by Scone [3].

The asynchronous interface consists of the system call wrappers inside the enclave and system call threads in the untrusted runtime outside. The enclave communicates with the untrusted runtime using shared-memory data structures—system call slots and system call queues. The system call wrapper populates a slot with system call arguments and sends slot index to the untrusted runtime via a submission queue. A system call thread dequeues the slot index from the thread and invokes the system call using arguments found in the system call slot. Once the system call has finished, its return value is written in the system call slot and its index is enqueued into the return queue. The wrapper examines this queue and gives the return value to the caller. Using asynchronous system call interface allows Pesos to maintain high I/O rate in spite of the SGX overhead.

In addition to the passing of system calls, Scone incorporates shields that transparently encrypt system call arguments such as data written to the local file system. Furthermore, these shields perform basic verification of arguments to prevent information leakage and Iago attacks [8].

5 USE CASES

To demonstrate the flexibility of the policy language and the performance of Pesos, we have implemented several real-world storage system usage scenarios, which we describe next.

5.1 Content Server

Cloud-based storage systems serve content to clients subject to an access control check. Pesos uses its policy engine to implement per-object access control lists granting read and write access to an object only to a specified, authenticated client. Clients are uniquely identified based on their X.509 certificate used to establish a TLS connection to Pesos. The example policy below allows Alice and Bob to read, but only Alice to update the object and an administrator to delete the object.

$$\begin{aligned}&\textbf{read} :- \mathit{sessionKeyIs}(K_{\texttt{alice}}) \lor \mathit{sessionKeyIs}(K_{\texttt{bob}})\\&\textbf{update} :- \mathit{sessionKeyIs}(K_{\texttt{alice}})\\&\textbf{destroy} :- \mathit{sessionKeyIs}(K_{\texttt{admin}})\end{aligned}$$

5.2 Time-based Storage

In addition to restricting access based on a client's identity, a content server may restrict access during a pre-defined time interval. For instance, objects containing secret information may not be released until a pre-defined date (time capsule), while other objects are preserved until their legally mandated storage lease expires (no updates before a certain date).

Time-based policies require a trusted time source to ensure the current time falls into the defined interval. A trusted time source is either available via the Intel SGX SDK [33] (provides the `sgx_get_trusted_time` function to receive a trusted time) or any other third-party time server specified as an authority in the policy. In the latter case, clients provide a time certificate from a time authority including a nonce generated by Pesos to ensure freshness of the certificate. Time-based policies mandate the authority as well as the freshness of the certificate.

The following example policy allows updates only after a specified date passed. In this example, we also demonstrate a chain of trust where a certificate authority (denoted by K_{CA}) authorizes the time server to sign the tuple *time*.

$$
\begin{aligned}
&\textbf{update}:-certificateSays(K_{\mathsf{CA}},'ts'(tskey))\\
&\quad\wedge certificateSays(tskey,'time'(t))\\
&\quad\wedge ge(t,DATETIMESTAMP)
\end{aligned}
$$

5.3 Versioned Store

Object versioning is useful to preserve the history of an object. If, for example, an object is corrupted, previous versions can be retrieved to find when and where the corruption occurred.

Versioned storage as implemented by Pesos' policies allows objects to define an index for the value. Using a version storage policy, Pesos restricts access to objects by the index, e.g., allowing writes only if the to-be-written index is an increment of the most recently stored index. Privileged clients may read the complete history of an object while regular clients could have limited access to only the latest index. The policy includes an exception allowing the initial creation of the object at version 0. The policy is expressed as follows:

$$
\begin{aligned}
&\textbf{update}:-\Big(objId(this,o)\wedge currVersion(o,cV)\\
&\quad\wedge nextVersion(cV+1)\Big)\\
&\quad\vee\Big(objId(this,NULL)\wedge nextVersion(0)\Big)
\end{aligned}
$$

5.4 Mandatory Access Logging

Mandatory Access Logging (MAL) combines access control, versioning, and information provenance to enforce logging accesses before performing the access. The client has to (1) append the intention of an operation into a log, and (2) execute the operation. A log is a separate object with a version storage policy. For each access to an MAL protected object, Pesos checks the log for an entry with the matching intent. Access is only granted, if the log contains the matching entry. The expected format of a log entry is defined by the MAL policy.

In the following example a log entry is a tuple of the operation, index, previous object-hash, new object-hash and client id. The log preserves a history of operations including the acting client. The following formula shows a simplified version of a MAL-policy which includes the version storage policy in the update permission (nextIndex = currentIndex +1):

$$
\begin{aligned}
&\textbf{read}:-objId(THIS,o)\wedge objId(LOG,l)\wedge currIndex(o,v)\\
&\quad\wedge sessionKeyIs(u)\wedge objSays(l,v,'read'(o,v,u))\\
&\textbf{update}:-objId(THIS,o)\wedge objId(LOG,l)\wedge sessionKeyIs(u)\\
&\quad\wedge currIndex(o,v)\wedge nextIndex(o,v+1)\wedge objHash(o,v,cH)\\
&\quad\wedge objHash(o,v+1,nH)\wedge objSays(l,lv,'write'(o,v,cH,nH,u))
\end{aligned}
$$

6 EVALUATION

We first describe the experiment setup including the hardware/software configuration, workloads and experiment design (§6.1). Next we evaluate Pesos raw performance without the policy enforcement (§6.2), and Pesos replication (§6.3). To demonstrate the functionality of Pesos and its end-to-end performance, we conclude by evaluating two use cases (§6.4).

6.1 Experiment Setup

Hardware. We use an Intel Xeon E3-1270 v5 CPU (v1 SGX) with 64 GB of main memory to run the Pesos controller. A dual-socket server (two Intel Xeon E5-2683 v3 CPUs, 128 GB of memory) hosts the workload generator. The Pesos controller connects to the workload generator via a dedicated switched 10 Gbit Ethernet network. In addition, we use three Seagate Kinetic drives [39] (4 TB per drive) assembled in an enclosure (Seagate's Ember Unit [62]). The enclosure includes an Ethernet switch that makes the disk's two Ethernet ports accessible to the outside. Seagate distributed the enclosures for testing and evaluation only. A production system would connect the disk's Ethernet ports directly into a SAS/SAN backplane. We connect one of the enclosure's 1 Gb Ethernet port to the Pesos controller machine over the switched network. In addition to the Kinetic disks, we repeat the same measurements against the Kinetic disk simulator [38]. We do this to show the Pesos controller's performance independent of the physical disk's (limited) performance and in anticipation of hardware improvements in the future. The simulator exposes the same API as the actual disks but runs entirely in-memory, collocated with the workload generator. For each physical disk, we use one simulator instance.

Software. Our server machines run Ubuntu 16.04 with a generic kernel (version 4.4). The Kinetic simulator and YCSB (version 0.4) execute in an Oracle Java 8 runtime environment. For the compilation of Pesos, we use the GNU Compiler Collection (GCC) 7.2.0 using OpenSSL 1.1.0 and the latest version of Scone (February 2018).

Workloads. Our workload generator creates YCSB-based traces [9] and stores them persistently before running the experiment. To optimize performance, we use an adapted client [18] to replay the traces to Pesos. We added a custom connector to the client to work in combination with Pesos' REST interface. YCSB comes with four stock workloads (A – D) each with its own key popularity distribution and read-/write ratio. During our experiments, we found that the results were similar in each case, hence, we only present graphs for YCSB workload A. YCSB workload A distributes read and write operations in a 50/50 split. We configure YCSB to generate workloads with 100,000 operations and 100,000 unique objects with a 1 KB payload.

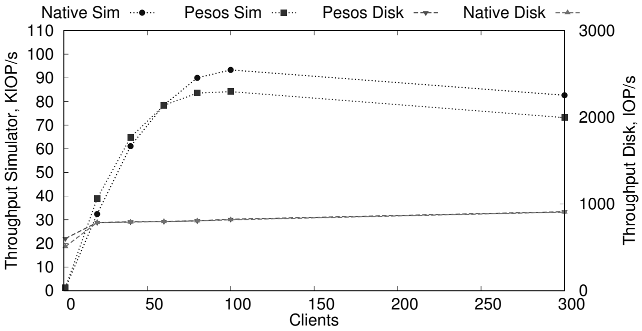

Figure 3: Pesos throughput with an increasing number of clients. (SD < 3.0%)

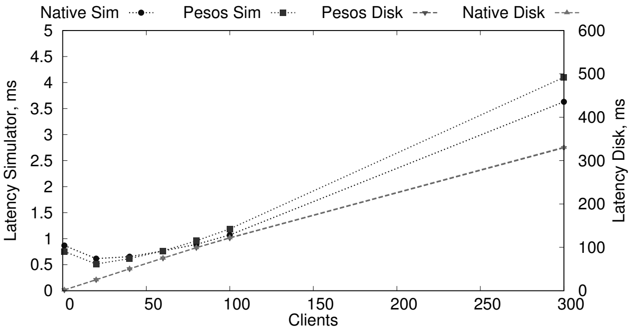

Figure 4: Pesos latency with an increasing number of clients. (SD < 3.0%)

Methodology. We report throughput (MB/s) and operations per second (IOP/s) as the basis of our performance evaluation. We evaluate a total of four different combinations of the Pesos controller and storage backend. First, we build two versions of the Pesos controller: a *native* version without SGX and a version with SGX (referred to as Pesos). By comparing Pesos (with Scone) to its native variant, we provide insight into the overhead that the secure execution environment imposes on an I/O-heavy application. Second, we evaluate two storage backends: one uses the *simulator* instead of actual Kinetic disks. The second storage backend uses the Kinetic disks. This setup allows us to eliminate the Kinetic disks as a potential performance bottleneck and achieve better visibility into the performance of the Pesos controller itself.

The graph's left axis present the performance numbers for simulated disks while the right axis shows the performance against the actual Kinetic disks. For each benchmark, we report the overall standard deviation (SD) beneath the plot. The client, Pesos controller and the storage backend all use TLS to secure their communication.

6.2 Performance

A first micro-benchmark gradually increases the number of clients to evaluate Pesos under load. We use an object size of 1 KB to focus on the sustained I/O operations per second. Large objects allow the system to amortize the per-object overhead more easily since Pesos copies more data per object. Hence, small objects are the more interesting workload.

Figure 3 depicts the throughput for four different configurations. We observe that the native variant scales up to 95 KIOP/s executing against simulated disks. Pesos, including the SGX overhead, achieves a throughput of 85 KIOP/s. To put this into perspective: Scone [3] reports around 40,000 requests per seconds for the Apache web server which also uses one thread per client connection. Arnautov et al. in Scone also benchmarked an event-driven, single-threaded web server (Nginx) which exhibited better resource efficiency. To increase performance in the future, we may introduce a similar non-blocking, single-threaded design in Pesos. A more immediate solution to increase the overall system throughput is to run multiple Pesos instances in parallel behind a load balancer while sharding the object space among them.

Running the same measurement against the real Kinetic disks, the peak throughput is at 1,080 IOP/s on average. This result is not surprising since the real disks are severely limited by the seek times of the disk head. Achieving performance similar to the simulated disk scenario requires using many more real Kinetic disks as the back end.

Figure 4 shows the average latency for each of the above cases. When running against the disk simulator, the latency is slightly elevated for only one client (0.75 ms for Pesos and 0.86 ms for native). This is an implementation artifact of the simulator that is only meant to test functionality but not designed for optimal performance. As the number of concurrent clients increases, the latency drops to 0.5 ms (Pesos) and 0.6 ms (native) at 20 clients against the simulated disks. Notice that the impact of SGX on the latency is negligible as the native variant only has a slightly smaller latency with 40 or more clients. The latency increases linearly as client requests are queued past the maximum throughput point. When running against physical disks the latency increases gradually starting from a single client since even a single client creates more requests than the disks can handle.

Figure 5 shows the throughput measurements with an increasing number of disks. For this experiment, we connected a single Pesos instance with one disk and increased the number of Pesos instances. Due to limitations of available hardware (three Seagate Kinetic disk and four SGX-capable servers), were not able to fully evaluate Pesos on a larger scale. The measurement shows that Pesos scales almost linearly with an increasing number of controller instances. The combined maximum throughput when using three simulated disks reaches 242 KIOP/s from 89 KIOP/s with one simulated disk for Pesos and 280 KIOP/s from 95 KIOP/s for native. Similarly, the throughput increases from 823 IOP/s to 2,439 IOP/s for the

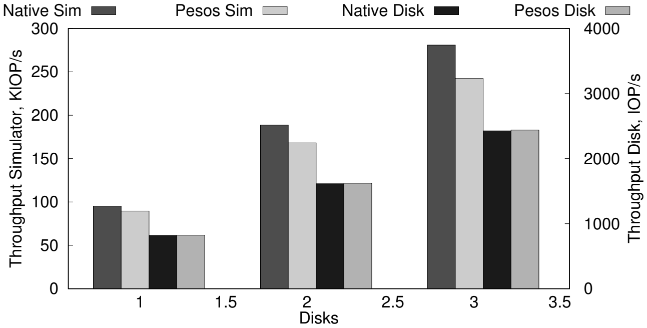

Figure 5: Scalability number of disks, 1 KB payload. (SD < 3.9%)

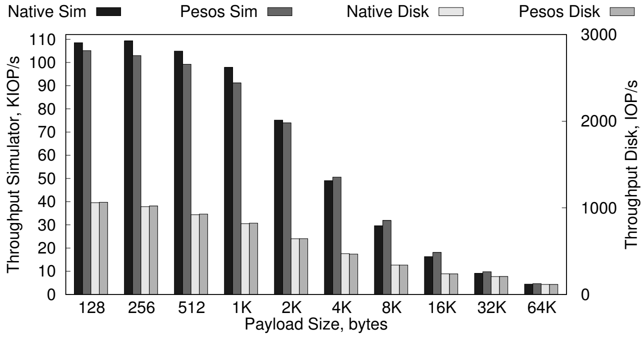

Figure 6: Evaluation of value sizes on overall system throughput using 100 clients. (SD < 8.4%)

measurements with the Kinetic disks for Pesos and from 818 IOP/s to 2,427 IOP/s for native.

Next, we measure the overhead caused by payload encryption. By default, Pesos encrypts the data before storing it on the disks, besides communicating with the clients through an encrypted channel. In this experiment, we used one Pesos instance connected to the simulator and measured the throughput with an increasing number of clients. At 1 KB payload size, payload encryption imposes a 1.5% overhead across 1-300 clients.

We conclude the section on Pesos micro-benchmarks with Figure 6. We increase the object size from 128 B up to 64 KB. At 256 B payload size, the throughput starts to gradually decreases towards larger object sizes. For small object sizes we observe CPU-bound workloads, whereas larger object sizes amortize the per-request processing and saturate the I/O paths decreasing the performance further. At 128 B objects, Pesos Sim achieves a maximum of 105 KIOP/s. In general Pesos overhead for object sizes smaller than 4 KB is within 4%.

6.3 Replication

As mentioned earlier (§4.5) the Kinetic disks offer an API to copy objects directly between disks without a third party relaying the communication. However, due to the limited performance of the P2P API, the Pesos controller instead coordinates replication and issues additional writes against the Kinetic disks.

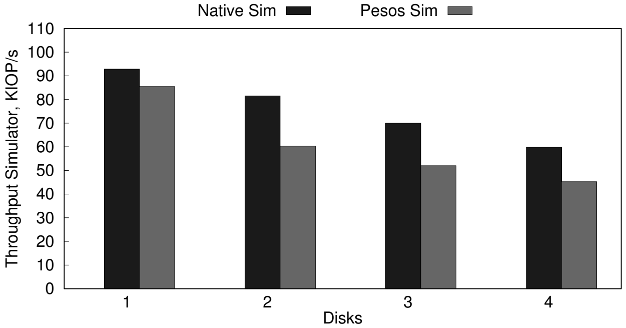

Figure 7: Effect of replication on the overall system throughput. Each object is replicated on all disks. (SD < 2.2%)

Figure 7 shows how enabling replication affects Pesos' overall throughput. Since Pesos has to issue multiple writes to the disks for each write from a client, we expect the overall system throughput to decrease. We only report numbers against the simulator to highlight the effect of replication on the Pesos controller. Running these experiments against the real disks would reveal no useful results, since the disks quickly become a bottleneck.

We increase the number of the simulated disks from 1 to 4 and replicate every object onto all available disks. We observe that the overall throughput in the native variant only decreases by 12% for each additional replication. In contrast, Pesos exhibits a larger decline in throughput of 30% between 1 and 2 disks and of 13% for more disks. The decrease in total system throughput is present, since the Pesos controller has to coordinate multiple backend writes for each client write.

6.4 Evaluation of Use Cases

Next we quantify the effect of policy enforcement. We begin the use case evaluation by measuring the effectiveness of the policy cache. The effectiveness of the cache depends on the number of policies in-use by Pesos. For instance, version storage policies can be reused by multiple objects lowering the number of policies in the Pesos controller. On the other hand complex policies like mandatory access logging policies are assigned per object.

Figure 8 shows the system throughput while varying the number of unique policies. We associate a variable number of policies (x-axis) to a set of 100,000 objects. The policy cache has a capacity of 50,000 entries. With one policy attached to all objects, the performance overhead stays below 5.5% compared to results without policy checking.

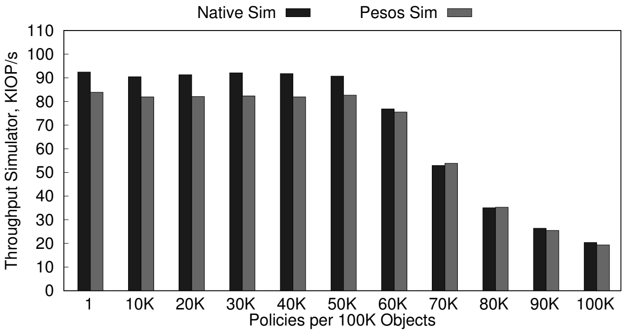

Figure 8: Effect of policy to object mapping with regard to caching and system throughput. (SD < 6.1%)

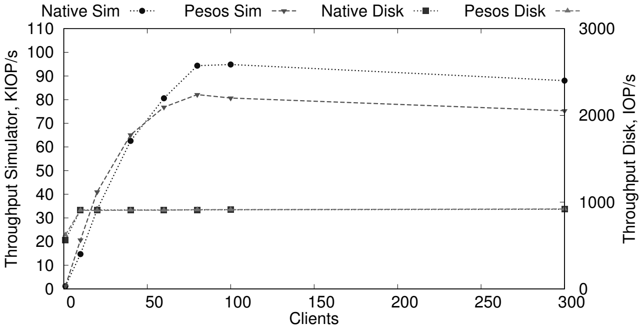

Figure 9: Effect of policy checking on overall throughput, verisoned storage use-case. (SD < 3.9%)

If the number of policies exceeds the policy cache size, Pesos has to fetch the policy from the simulator. The results indicate that while all policies fit into the cache, the throughput is nearly unaffected by the policy enforcement. At 60,000 policies the cache hit rate drops and the overall system throughput starts to decrease, due to additional disk accesses. We do not report measurements with the Kinetic disks as their overall slow performance does not benefit from policy caching.

Versioned storage. The versioned storage policy preserves the history of objects. To preserve the history, Pesos maintains a counter (version number) for the object. A client may only update a versioned object, if it provides the correct version number in its update request. The policy enforces that the client supplied version number matches what Pesos expects. Internally, Pesos creates a new key for each version of an object. The overhead imposed by the versioned storage is negligible as Figure 9 illustrates. Pesos achieves a maximum throughput of 82 KIOP/s against the simulator, which is 2.3% lower compared to earlier measurements without the policy checking (84 KIOP/s).

Mandatory access logging (MAL). An MAL policy maintains a log of operations performed on an object. The granularity of the log is a tuning parameter to trade off logging overhead and detail. In the fine-grained case, each client access requires an additional backend write to update the log. If the log granularity *G* is more coarse grained, the log must only be updated every *G* entries. We explore this relationship in Figure 10. The baseline shows the throughput for the MAL usecase without maintaining a log. At one log entry for every write operation, the effective throughput reduces to 50 KIOP/s. Logging only every 10th access, achieves 95% (92% native) of the baseline performance. When increasing the granularity even further, the throughput reaches 66 KIOP/s for Pesos and 77 KIOP/s for native. Compared to previous results, the additional log appends decrease performance by 23%.

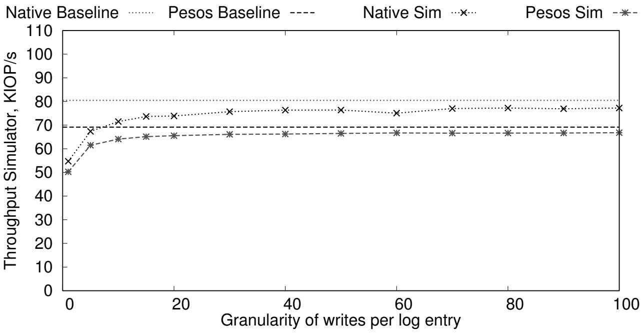

Figure 10: Effect of log granularity on overall throughput, compared to writes without logging. (SD < 5.4%)

7 RELATED WORK

Hardware object stores. The growing complexity of storage systems has urged developers to seek a solution that would simplify storage usage and management by extending device functionalities. While software-based approaches [55, 70, 72] encapsulate functionality within dedicated servers, hardware manufacturers have begun to incorporate advanced capabilities into hard drives. The Seagate Kinetic Disk features an SoC system running a customized Linux kernel, basic management software, and exports a LevelDB database on the network [81]. Samsung is developing a similar solution [85]. Although their solution does not have a network interface, it is designed to simplify and accelerate key-value storage systems. Pesos leverages such devices to build a policy-enhanced secure storage.

Secure storage systems. Traditionally commercial file systems use access control lists to protect against unauthenticated access [1]. In addition, many systems [23, 26, 35, 44, 45, 51, 83] provide secure storage in untrusted environments through software based cryptography. Many of these systems also incorporate aspects of decentralization, i.e. multi-client collaboration [6] to provide security. In contrast, Santos et al. [60] propose to use the trusted platform module (TPM) to bind software to a specific machine configuration and booted software stack. This ensures that data is only accessed in the presence of a trusted software stack. Pesos similarly relies on trusted hardware (SGX) but overcomes the security and functionality limitations of TPM. In the future Pesos could combine TPM and SGX to extend its security features.

Policy-based storage systems. Policies allow for data access control management at a fine granularity and high degree of flexibility. Policy languages have been used to express client capabilities [24], enforce confidentiality and integrity [23], enable data sharing with strong security guarantees [46], restricting access to rows and columns of relational DBMS [49], or to define decryption allowance beyond private keys [60]. Guardat [82] and Thoth [17] represent systems that allow clients to define individual per-data policies with a high degree of flexibility. Pesos has adopted their policy language to achieve access controlled data security while enabling a large variety of use cases.

Trusted Execution Environments (TEEs). Before the introduction of Intel SGX, trusted execution environments were built on top of special processors [43, 71] or modules [79]. Intel SGX brought a versatile and performant trusted execution environment to the x86 platform on commodity processors. Since its public release in October 2015, novel use cases [31, 61, 87], weaknesses [29, 84] and countermeasures [64] have been explored by academics. Various frameworks and tools [3, 5, 65, 80] ease the development effort but make different design decisions and trade-offs that have to be evaluated carefully. Pesos relies on Scone [3] which, for example, implements asynchronous system calls to reduce the number of expensive enclave transitions. Independent of the framework, the executed code may still expose vulnerabilities through bugs. Additional measures, such as, formal verification [67, 68], protection against side-channel attacks [84], and buffer overflows [41] are needed to improve safety of enclave code. Pesos can benefit from all these efforts to improve code safety.

Secure data processing. In the context of using SGX for secure computing, VC3 [61] is one of the first systems that incorporates SGX to the domain of big data processing by applying it to the Hadoop Map/Reduce framework. Likewise, Opaque [87] uses Intel SGX to provide encryption, oblivious computing, and integrity protection to a secure distributed data analytics applications. SGXBOUNDS [41] provides lightweight memory safety techniques to SGX-based enclaves for computation in production systems. Likewise, Slick [77] and ShieldBox [78] leverage SGX to build a secure middlebox framework for high-performance network processing. In contrast, Pesos focuses on secure data storage using a combination Intel SGX and Kinetic storage.

8 CONCLUSION AND FUTURE WORK

Today's complex storage systems hosted by untrusted third-parties are vulnerable to confidentiality and integrity violations. Pesos, a policy-enhanced secure object store, enforces data confidentiality and integrity policies while operating in an untrusted cloud environment. Pesos achieves these unique properties by leveraging a novel combination of the recent advancements in trusted computing technologies. More specifically, Pesos exposes a declarative policy language for concisely expressing a wide range of storage security policies. Further, Pesos enforces these security policies by leveraging the combination of shielded execution based on Intel SGX and Kinetic Open Storage for trusted storage. To its clients, Pesos provides an interface to verify storage operations via cryptographic attestation of the stored policies and content. We built Pesos as a fully functional storage system supporting many end-to-end important design features, and a range of effective performance optimizations. We evaluated Pesos based on micro-benchmarks and case-studies using the YCSB workloads. Our evaluation results demonstrate the viability of Pesos by achieving throughput within 85% of its native variant, while providing stronger security properties and smaller TCB.

For the future work, we plan to pursue mainly three research directions: Firstly, we will extend Pesos with a local SSD as the untrusted fast caching layer to overcome the limitations of main memory capacity (EPC paging) and slow disk performance, while protecting against integrity and freshness attacks. Secondly, Pesos could leverage integration with the storage performance development kit (SPDK) [34] for the faster access to storage directly from the userspace; thus bypassing the OS kernel for improved I/O performance. And lastly, Kinetic disks could be protected against physical attacks by building hardware protections into the SoC [19].

Acknowledgements. We thank Peter Druschel, Deepak Garg, Rodrigo Rodrigues, and our shepherd Sonia Ben Mokhtar for their helpful comments. We would also like to thank Franz Gregor and Sergei Arnautov for their continuous support for the Scone framework. Furthermore, we would like to thank Seagate for providing the Seagate Kinetic disks. This project was funded by the European Union's Horizon 2020 research and innovation program under grant agreements No. 645011 (SERECA), No. 777154 (ATOMSPHERE), No. 780681 (LEGaTO), and No. 690111 (SecureCloud).

REFERENCES

[1] Amazon. Amazon S3 Developer Guide. http://aws.amazon.com/documentation/s3. Last accessed: February, 2018.

[2] I. Anati, S. Gueron, P. S. Johnson, and R. V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[3] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[4] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[6] M. Brandenburger, C. Cachin, and N. Knežević. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. *ACM Transactions on Privacy and Security (TOPS)*, 2017.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[8] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010.

[10] V. Costan and S. Devadas. Intel sgx explained., 2016.

[11] CRN. The ten biggest cloud outages of 2013. https://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm, 2013. Last accessed: February, 2018.

[12] B. Debnath, S. Sengupta, and J. Li. FlashStore: High Throughput Persistent Key-value Store. In *Proceedings of the VLDB Endowment*, 2010.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[14] Dell. Elastic Cloud Storage. https://www.dellemc.com/en-us/storage/ecs/, 2017. Last accessed: February, 2018.

[15] J. DeTreville. Binder, a Logic-Based Security Language. In *Procceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)*, 2002.

[16] T. Economist. The data deluge Businesses, governments and society are only starting to tap its vast potential. http://www.economist.com/node/15579717, 2010. Last accessed: February, 2018.

[17] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[18] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[19] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[20] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004.

[21] The Fast Lexical Analyzer. https://github.com/westes/flex. Last accessed: February, 2018.

[22] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[23] D. Garg and F. Pfenning. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[24] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[25] GNU Bison. https://www.gnu.org/software/bison/. Last accessed: February, 2018.

[26] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2003.

[27] Google. Cloud Storage. http://www.cloud.google.com/storage, 2017. Last accessed: February, 2018.

[28] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[29] M. Hähnel, W. Cui, and M. Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[30] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[31] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[32] Intel Software Guard Extensions Programming Reference. https://software.intel.com/en-us/intel-sgx-programming-reference, 2014. Last accessed: February, 2018.

[33] Intel Software Guard Extensions SDK for Linux OS. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf. Last accessed: February, 2018.

[34] Intel Storage Performance Development Kit. http://www.spdk.io. Last accessed: February, 2018.

[35] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[36] Kinetic C library. https://github.com/Kinetic/kinetic-c. Last accessed: February, 2018.

[37] Kinetic Data Center Comparison. https://www.openkinetic.org/technology/data-center-comparison. Last accessed: February, 2018.

[38] Kinetic Disk Simulator. https://github.com/Kinetic/kinetic-java. Last accessed: February, 2018.

[39] Kinetic HDD Data Sheet. https://www.seagate.com/files/www-content/product-content/hdd-fam/kinetic-hdd/en-us/docs/kinetic-ds1835-1-1110us.pdf. Last accessed: February, 2018.

[40] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating Remote Attestation with Transport Layer Security. 2018, arXiv:1801.05863.

[41] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, 2017.

[42] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM Symposium on Principles*

of distributed computing (PODC). ACM, 2009.

[43] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[44] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 2007.

[45] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[46] K. Mast, L. Chen, and E. Gün Sirer. Enabling Strong Database Integrity using Trusted Execution Environments. 2018, arXiv:1801.01618.

[47] M. L. Mazurek, Y. Liang, W. Melicher, M. Sleeper, L. Bauer, G. R. Ganger, N. Gupta, and M. K. Reiter. Toward strong, usable access control for shared distributed data. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[48] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[49] A. Mehta, E. Elnikety, K. Harvey, D. Garg, and P. Druschel. Qapla: Policy compliance for database-backed systems. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[50] Microsoft. Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs. Last accessed: February, 2018.

[51] E. L. Miller, D. D. Long, W. E. Freeman, and B. Reed. Strong Security for Network-Attached Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[52] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[53] Mongoose embedded web server. https://cesanta.com. Last accessed: February, 2018.

[54] Open Kinetic. https://www.openkinetic.org/. Last accessed: February, 2018.

[55] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[56] T. Register. Press release. https://www.theregister.co.uk/2017/08/09/samsungs_128tb_ssd_bombshell/. Last accessed: February, 2018.

[57] K. Ren, A. Thomson, and D. J. Abadi. VLL: A Lock Manager Redesign for Main Memory Database Systems. In *The VLDB Journal*, 2015.

[58] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[59] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards Trusted Cloud Computing. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[60] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[61] F. Schuster, M. Costa, C. Gkantsidis, M. Peinado, G. Mainar-ruiz, and M. Russinovich. VC3 : Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[62] Seagate Kinetic Development Chassis (Ember). http://web.archive.org/web/20160403050114/developers.seagate.com/display/KV/Development+Chassis. Last accessed: February, 2018.

[63] Seagate Kinetic Enterprise Storage. https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/. Last accessed: February, 2018.

[64] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[65] S. Shinde, D. Le Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[66] J. Shu, Z. Shen, and W. Xue. Shield: A stackable secure storage system for file sharing in public storage. *Journal of Parallel and Distributed Computing*, 2014.

[67] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A Design and Verification Methodology for Secure Isolated Regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[68] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying Confidentiality of Enclave Programs. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[69] L. Soares and M. Stumm. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[70] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[71] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th ACM Annual International Conference on Supercomputing (ICS)*, 2003.

[72] S. Sundararaman, G. Sivathanu, and E. Zadok. Selective Versioning in a Secure Disk System. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[73] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[74] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD international Conference on Management of data (SIGMOD)*, 2012.

[75] V. Thummala and J. S. Chase. SAFE: A declarative trust management system with linked credentials. *CoRR*, 2015.

[76] Toshiba. Press release. http://news.toshiba.com/press-release/business-and-retail-solutions/toshiba-demonstrates-high-performance-object-storage-tec. Last accessed: February, 2018.

[77] B. Trach, A. Krohmer, S. Arnautov, F. Gregor, P. Bhatotia, and C. Fetzer. Slick: Secure Middleboxes using Shielded Execution. 2017, arXiv:1709.04226.

[78] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[79] Trusted Computing Group. TPM Main Specification. https://trustedcomputinggroup.org/tpm-main-specification, 2011. Last accessed: February, 2018.

[80] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[81] University of Minnesota—Center for Research in Intelligent Storage. Kinetic Action: Micro and Macro Benchmark-based Performance Analysis of Kinetic Drives Against LevelDB-based Servers. http://www-users.cselabs.umn.edu/classes/Spring-2017/csci5980/files/KVS/bare_conf.pdf, 2017.

[82] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*, 2015.

[83] C. Weinhold and H. Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[84] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[85] Yang Seok Ki. Key Value SSD Explained - Concept, Device, System, and Standard, 2017. Last accessed: February, 2018.

[86] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie. A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores. In *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*, 2017.

[87] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.