Guardat: A foundation for policy-protected persistent data

[1], Anjo Vahldiek[†], Eslam Elnikety[†], Aastha Mehta[†], Deepak Garg[†], Peter Druschel[†], Johannes Gehrke[‡], Rodrigo Rodrigues[†], and Ansley Post[†]

[†]Max Planck Institute for Software Systems (MPI-SWS)
[‡]Cornell University

Abstract

We present Guardat, an architecture that enforces rich data access policies at the storage layer. Users, application developers and system administrators can provide per-object policies to Guardat. Guardat enforces these policies and provides attestations about the state of stored objects. With Guardat, the data integrity, confidentiality and access accounting rules for a collection of objects can be stated as a single declarative policy. Policy enforcement relies only on the integrity of the Guardat controller and any external policy dependencies; it does not depend on correct software, configuration and operator actions in other parts of a system. Guardat allows developers, system administrators and third-party hosting platform providers to enforce concise, system-wide data protection policies based on a small trusted computing base, and to demonstrate their compliance to any party that trusts the Guardat layer. We present a design and prototype implementation of Guardat, show experimentally that the overhead of making policy checks and storing additional metadata are low, and discuss applications and policies.

1 Introduction

As the volume and value of digitally stored assets keep increasing, so do the risks to the integrity and confidentiality of said data. Computer and storage systems are increasing in complexity, exposing data to risks from software bugs, security vulnerabilities and human error. In addition, data is increasingly stored on third-party platforms, introducing additional risks like unauthorized data use by a third party.

In today's systems, the confidentiality and integrity of persistent data depend on the absence of design errors, bugs, malware and operator mistakes in most components of a system. Moreover, the applicable policies for a collection of data objects may be implicit in the code, and their specification and enforcement spread over different parts of a system, increasing the risk of misconfigurations. For data stored on third-party platforms, data confidentiality and integrity, as well as proper accounting of data use, additionally depend on the reliability of the third-party provider.

To address these challenges, we present *Guardat*, an architecture that includes a policy interpreter, crypto and enforcement logic at the storage layer. With Guardat, users, developers and administrators can state the integrity, confidentiality, and accounting rules for a collection of data objects using a concise, declarative policy language. Applications communicate with Guardat through secure channels, tunneling through untrusted system layers like storage servers or hosting platforms. Applications send policies, commands and evidence of policy compliance (e.g., proof of authentication) to Guardat and request attestations of stored data and their policies from Guardat. Guardat enforces the policies while relying only on its own interpreter and enforcement logic and any explicit policy dependencies, thus min-

imizing the computing base relied upon for enforcement, and its attack surface.

A Guardat policy specifies the conditions under which an object may be read, updated, or have its policy changed. These conditions, written in a simple but expressive declarative language, may depend on client authentication, the initial and final states of the object (size and content) in an update transaction, or signed statements by external trusted components (certifying, for instance, the current wall-clock time). Guardat stores the policy as part of its own metadata and ensures that each access to the object complies with the policy.

Following are some example Guardat policies that mitigate important threats: System binaries can be protected from viruses through a policy that allows modification only when the updated binary is signed by a trusted party; system log corruption and tampering can be avoided through a Guardat-enforced append-only policy; accidental deletion or corruption of backup data can be prevented by a policy that prevents modification for a specific period of time; confidentiality of a user's private data can be enforced by allowing reads only in a session authenticated by the user's public key; and, accesses to a data object can be permitted only if a corresponding access record is added to an append-only log file, enforcing mandatory access logging.

While these policies can be implemented in higher software layers, the merit of using Guardat is that the policy applicable to a collection of data objects can be specified using a concise, declarative language, and enforced by a small trusted computing base (TCB) with a small attack surface. Guardat complements existing techniques for ensuring the reliability of data processing systems, including software testing, verification, security auditing, sealed data and trusted computing. While no technique can provide comprehensive protection, Guardat provides a safety net that protects a system's persistent data from a wide range of threats. Moreover, Guardat can demonstrate compliance with client and provider policies, as well as legal mandates to any party that trusts Guardat.

Guardat can provide additional benefits in multi-party environments where all parties trust Guardat, e.g., a client storing her data at a hosting provider, or a service provider allowing caching of parts of its database on a client device. Here, Guardat can enforce the data owner's policies on third-party data accesses, and the data holder can use Guardat to demonstrate its compliance with client and provider policies.

The Guardat design is based on three principles. First, enforcing policy at the storage layer minimizes the TCB and its attack surface. Second, a simple, declarative policy language allows the concise specification of all policy related to a collection of data objects. Third, the policy language supports a small set of declarative primitives expressive enough to specify the access policy, leaving it to untrusted code to provide the mechanism required to satisfy the policy.

Guardat can be implemented in different ways, depending on the deployment scenario and threat model. For instance, an integration into a SAN server enforces the policy on all accesses from network clients as long as the server is trusted; while an integration into the microcontroller of a storage device enforces the policy on all accesses as long as the controller chip is not compromised. This paper contributes the Guardat architecture and design, its policy language, a prototype implementation and a performance evaluation based on sample policies and application scenarios.

We cover background and related work in Section 2. Section 3 describes the design of Guardat and its policy language. Section 4 presents example policies and the guarantees they provide. In Section 5, we present results from experiments with a prototype implementation in the iSCSI IET SAN server. Using microbenchmarks and experiments with a Web server, we show that Guardat policies can be enforced with low overhead, thanks to the efficient caching of metadata and its persistent storage in Flash memory, as well as optimizations that overlap metadata operations with disk reads and writes. We conclude in Section 6 and provide additional details about the Guardat API, policy language and implementation in an Appendix.

2 Background and related work

Storage work group specification. Although developed independently, the Guardat architecture bears some resemblance to a set of specifications for storage devices standardized by the storage work group of the trusted computing group (TCG) [43]. Similar to Guardat, the TCG standard prescribes session-based communication with storage devices and access control on all calls to them. This industry interest supports the case for Guardat's architecture. Unlike our work, however, the TCG standard does not describe a concrete design, implementation, or policy language, leaving these to device vendors; nor does it include certification of stored data by the storage device. Implementations exist for a subset of the TCG specification [42], providing full-disk encryption to preserve confidentiality of data upon device theft, loss or end of life. They do not include secure sessions, universal access checks or integrity policies, all of which Guardat does.

Guardat vs. trusted computing. At a high level, trusted computing (TC) relies on a trusted platform module (TPM) attached to a computer's motherboard to provide a hardware root-of-trust [31], while Guardat relies on a controller (GDC) attached to a storage device, enclosure or server (called a storage device in the sequel). While TC provides remote attestation of the hardware/software executing on a computer, Guardat attests the state of stored objects, and enforces an application-defined policy for read and write accesses to objects. TC provides sealed storage, where disk data is encrypted with a key stored within the TPM and released only when the computer runs a specific, trusted software configuration. Guardat instead enforces a declarative policy on all data accesses from untrusted client code. Compared to TC, Guardat can reduce the size of the TCB and its attack surface. Depending on the policy and implementation, the TCB can be as small as the Guardat controller. Lastly, TC can complement Guardat: A Guardat policy for access to an object can require that trusted software, verified via TC remote attestation, execute on the client computer.

Related trusted computing proposals. Building on TC, semantic attestation [17] enforces properties of a computation by a runtime verification substrate within a virtual machine monitor. Guardat provides a limited form of semantic attestation that enforces a data access policy, and does not require machine virtualization.

Excalibur [36] extends sealed storage with a primitive that binds cryptographically sealed data to a policy, such that a node can decrypt the data only if a trusted authority states that it obeys the policy (e.g., "this node is in Europe" or "this node is running Xen"). Guardat can be used to implement a similar capability, possibly with the help of a trusted authority as required by the Excalibur policy. However, Guardat can enforce many rich policies directly, without requiring an external trusted authority to map high-level policy descriptions to the nodes that meet those requirements.

Pasture [22] is a messaging and logging library that enables data to be stored on an untrusted client machine while ensuring that a user cannot access the data without logging that access. Furthermore, users can delete unaccessed data in a way that provably prevents future access. The protocol relies on a TPM on the client machine. In Section 4, we describe a Guardat policy that can enforce the more general property of mandatory access logging for a collection of objects.

Protecting data integrity and confidentiality. Butler et al. [8, 9, 10] describe storage devices that control access to storage segments contingent on the presence of a hardware token, or on successful remote attestation of the host computer. In Guardat, these forms of access control can be expressed as policies. Moreover, Guardat supports much richer and per-object policies that can additionally depend on client authentication, wall-time clock or object contents, and Guardat supports certification of object states.

Commercially available self-encrypting disks [37] encrypt data to ensure its confidentiality when the device is lost or stolen. Guardat includes this capability as well, and additionally enforces rich data access policies. Web storage services like Amazon S3 [3] provide access control to a client's data based on user identities, groups and roles, encryption for secure data storage and transit, and access logging.

Guardat can enforce these (and many other) policies and provides object attestations. Because it operates at the storage layer, it does not require trust in the Cloud provider's remaining platform.

In capability-based network-attached storage (NAS) [16, 2, 13], individual access requests include a capability, i.e., a tamper-proof description of client access rights. This capability is created out of band by a policy manager, a trusted component that serves all storage devices in a data center. A Guardat device, on the other hand, can interpret and enforce rich policies without relying on an external policy manager; thus, Guardat can operate in an otherwise untrusted or offline environment (unless a policy specifically requires validation by an external trusted component). Guardat can enforce state-based policies and certify the state of objects and their policies, which capability-based NAS cannot.

Type-safe disks (TSD) [40] track the filesystem's relationship among disk blocks using an extended block interface. Thus, a TSD can enforce basic filesystem integrity invariants, such as preventing access to unlinked blocks. A security extension called ACCESS adds read and write capabilities to selected disk blocks, thus enabling access control for entire files and directories. Guardat goes beyond TSD in two ways. First, Guardat devices have signing keys and support secure channels, which lends them accountability and stronger security with respect to compromised hosts, buggy filesystems and operator mistakes. Second, Guardat's policy language can support policies far beyond filesystem metadata integrity.

Storage systems such as Self Securing Storage (S4) [44] and NetApp's SnapVault [18] RAID storage server retain shadow copies of overwritten data or disable writes for a given period of time to address the specific problem of accidental or malicious corruption of data. Guardat can enforce these and much richer integrity constraints (Section 4), as well as confidentiality and accountability.

jVPFS [47, 48] is a stacked, microkernel-based filesystem that combines a small, isolated trusted component with a conventional untrusted filesystem. jVPFS uses encryption, hash trees and logging to ensure data confidentiality and integrity. Guardat can provide similar functionality at the storage layer, and supports a much richer set of policies.

The filesystems PCFS [14] and PFS [46] enforce integrity and confidentiality policies expressed in rich policy languages similar to that of Guardat. However, unlike Guardat, PCFS and PFS trust the entire storage stack below the filesystem, cannot enforce integrity policies that depend on the content or size of objects, do not certify the state of stored objects, and can be bypassed by booting into a different configuration. PFS policies are expressed in a formal logic similar in expressiveness to the Guardat policy language. PCFS uses a formal logic that is more expressive than the Guardat policy language, but much more expensive (in terms of time and space) to implement. The Guardat policy language deliberately avoids policy features like recursive predicates that increase complexity but are rarely used in practice.

Protecting data availability. Storage systems like RAID [32], snapshotting filesystems [19, 30, 26] and some backup utilities [6, 27] use redundancy to ensure data availability. Guardat addresses the orthogonal problem of ensuring integrity, confidentiality and access accounting in the face of human error, adversarial threats and software bugs (e.g., a bug in a backup application that overwrites backed up data [15]). In practice, Guardat must be combined with redundant storage to ensure the availability of data in case of a media failure, loss, destruction or failure of a Guardat device.

Extended storage functionality. Commercial hybrid disks [39] package a magnetic disk drive with a modest amount of NAND Flash memory, used as a non-volatile write-back cache to increase performance. Guardat uses a comparable amount of Flash memory to store its policy metadata but, in addition, protects data. Object-based storage devices replace the traditional block-based with an object-based interface [24]. These systems offer capability-based security for whole objects, which we already compared to. Part of the Guardat API is also object-based, and could therefore be integrated with an emerging object-based storage standard. Several storage subsystems like active disks [35], semantically smart disks [41] and differentiated storage services [25] in-

clude program logic to improve performance. Guardat addresses the orthogonal concerns of data confidentiality, integrity and access accounting.

Pennington et al. [33] describe an intrusion detection system (IDS) at the storage layer, which raises an alarm when an access matches a per-file or global rule. Guardat instead is able to *enforce* per-file security policies, and these policies can be richer than the rules of an IDS system. However, intrusion detection rules could be specified as Guardat policies that allow offending accesses but log an alarm record.

Novelty. To the best of our knowledge, Guardat is the first system that enforces per-object and per-block general confidentiality, integrity and access accounting policies at the storage layer. Policies are expressed in a concise declarative language and can be predicated on a wide range of conditions, including client authentication, remote attestation, physical authorization tokens, trusted wall-clock time, and the state (content) of objects. The use of an expressive declarative language enables the concise specification of unified access control, integrity and accounting policies for persistent data, even at sub-object granularity. Enforcing policies at the storage (block-device) layer reduces the attack surface and, in many cases, the size of the TCB relied upon for enforcement. Existing techniques, on the other hand, either rely on a larger TCB, spread the specification and enforcement of policies affecting a given data object over many different components and layers of a system, or support a smaller set of policies.

3 Guardat design

The design of Guardat is guided by three principles. First, Guardat enforces policies entirely in the *storage layer* to minimize the TCB and its attack surface. Second, we keep policy specifications concise and separate from code by expressing policies in a domain-specific *declarative policy language*. Third, in the interest of a small TCB, the Guardat policy language provides only a minimal set of primitives sufficient to *check* a rich set of policies, but we rely on untrusted code to specify *how to satisfy* a policy. We point out instances of such design economy throughout this section, as we describe the Guardat API and policy language.

Design overview Data stored in Guardat is organized into *objects*, e.g., files. With each object, Guardat associates metadata, which includes the object's access policy. Guardat's program logic, called the Guardat controller or GDC, executes just above or inside the storage layer and enforces the object's policy on every read and write to it. The GDC exports an *extended block-device API* that allows users, applications and system administrators to (a) create, delete, read and update objects, (b) cryptographically authenticate and establish secure sessions to tunnel commands and data through other untrusted software and hardware, (c) associate policies with objects, (d) provide credentials and other information to satisfy these policies during subsequent access, and (e) obtain Guardat attestation on stored objects and their policies.

The policy of an object consists of four rules, one for each of the permissions **read**, **update**, **destroy** and **setpolicy**. Each rule, expressed declaratively in the Guardat policy language, specifies conditions on the context and environment under which the respective permission holds. Abstractly, the **read** rule represents the object's confidentiality policy; the **update** rule encodes the object's integrity policy; the **destroy** rule governs when the object's name can be recycled; and the **setpolicy** rule describes when the policy can be changed. API calls that read or update an object or its metadata check conditions of the corresponding policy rules.

Besides a device for storing data, the GDC requires a small amount of fast, persistent memory like Flash for storing policies and other metadata. Flash memory is widely available now; hybrid disks even combine a HDD and Flash in a single enclosure [39]. To authenticate itself as a legitimate Guardat device, sign attestations and encrypt data, the GDC includes a manufacturer-provided unique key pair and certificate.

Implementation, threat model and scope The Guardat design can be implemented in different ways

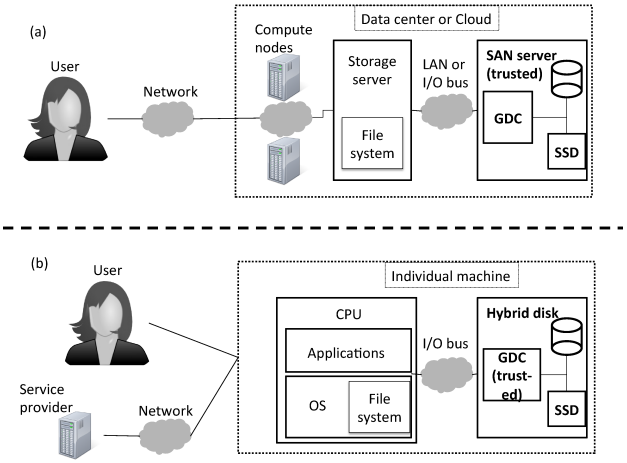

Figure 1: Possible implementations of Guardat: (a) In a SAN server and (b) In a hybrid disk's microcontroller

depending on deployment. Figure 1 shows two possible implementations. In (a), the GDC is implemented in a SAN server for use in a data center. In (b), the GDC is integrated with the microcontroller of a hybrid disk for use in an individual machine.

In each implementation, the GDC, metadata and data must be protected from unauthorized access and undetected tampering. In implementation (a), which is the basis of our Guardat prototype described in Section 5, the SAN server includes the GDC, data and metadata storage devices, and must be physically protected, e.g., in a machine room where access is restricted to trusted staff. In this scenario, the Guardat policies are enforced despite any bugs, misconfigurations, or security incidents outside the SAN server, and regardless of actions by employees without access to the machine room.

In implementation (b), the GDC is implemented as part of a microcontroller embedded in a hybrid disk. Here, the metadata and data are encrypted and authenticated to protect them from unauthorized access and undetected tampering. The microcontroller implements the GDC and stores its private key in an embedded TPM. In this scenario, the Guardat policies are enforced as long as the microcontroller has not been physically tampered with. While we have not attempted this implementation, we believe it is feasible with a high-end microcontroller that has on-chip hardware support for secure hashing and cryptography, as well as a TPM.

Table 1 lists examples of deployment scenarios, their threat models and trust assumptions. For instance, if a reputable Cloud provider deploys Guardat-enabled SAN servers to protect user data from bugs and misconfigurations in its infrastructure and from opportunistic access by employees, then the user must trust the Cloud provider to prevent physical access to the SAN by all but trusted employees (line A in Table 1). Similarly, if a digital content provider locally caches copyrighted content in a Guardat-enabled hybrid disk on a user's machine, then it must trust that the user is unable to tamper with the controller chip, much in same way trusted computing applications trust that users are unable to tamper with a TPM.

We also make obvious and standard assumptions about policies: correct policies must be installed when data is first stored, and external dependencies of policies (e.g., time servers, client's authentication keys) must be trustworthy. Under these assumptions, Guardat defends against threats to confidentiality and integrity of stored data or to data in transit to storage. This includes threats due to bugs and vulnerabilities in intermediate software layers including operating systems, filesystems, storage services built on top of Guardat, and networks, and threats due to human negligence and opportunistic malice. Guardat is not concerned with data availability. To mask the effects of a hardware or media failure, loss, or destruction of a Guardat device, data must be replicated on multiple devices with independent failure modes.

3.1 Guardat API

We describe the Guardat API that allows higher layers to manipulate objects, establish sessions, provide policies, provide credentials to satisfy policies, and obtain attestations. For reasons of space, we defer many details to the Appendix.

	Deployment objective	Guardat implementation	Trust assumption	Who trusts?	How trust is discharged
A.	User wishes to protect her data from bugs above the storage layer, and opportunistic employees at a reputable Cloud provider	Cloud storage servers	Only trusted staff has physical access to servers	User	Provider restricts physical access to servers
B.	Data center wishes protection from bugs, misconfigurations, disgruntled employees	Storage servers	ditto	Data center	Center restricts physical access to servers
C.	Service provider wishes to protect proprietary content cached on user's machine	Microcontroller in user's disk	User cannot compromise the controller	Provider	User lacks ability to tamper with controller chip
D.	User wants to protect data on her machine from bugs, viruses and mistakes	Microcontroller in user's disk	None needed	–	–

Table 1: Guardat deployment scenarios and trust assumptions

Guardat organizes data into objects. Physically, an object is an ordered sequence of disk extents with a unique name and an access policy, which together constitute the object's metadata. Relevant rules of an object's policy are evaluated on every read or write to the object or its metadata.

A user or an application (generically called *client*) interacts with Guardat in a *session*. A session is established with a handshake protocol in which the client and Guardat authenticate each other using their private keys. As part of the protocol, new, session-specific keys are created. These keys are used to encrypt and/or authenticate (through message authentication codes) all subsequent communication in the session. This protects in-transit data and commands from snooping and modification in intermediate layers. Moreover, the public key of the client (which acts as a unique identifier for the client) becomes available during every policy evaluation in the session; hence, Guardat can enforce policies that restrict access to specific users. At the end of the handshake, Guardat returns a unique session identifier that links later API calls to the session. Our simple handshake protocol and the two API calls it uses are shown in Appendix A. (To access objects whose policies do not require authentication, it is possible to communicate with Guardat outside of a session. Such communication is conceptually treated as part of a default, untrusted session.)

Within a session, the client may read and update stored objects in *batches*. A batch is a sequence of reads and updates on a *single* object; batches on distinct objects may proceed concurrently. The updates in a batch have transactional semantics: At the end of a batch, either all the updates are atomically persisted or they are all discarded. This design allows enforcing integrity policies that refer to the old and the new contents of more than one block within the object (the integrity policy is checked once for the entire object at the end of the batch). We find this design useful in encoding policy state machines and access-accounting policies, as illustrated in Section 4. However, this design decision comes with a trade-off: To avoid buffering a potentially unbounded number of updates during a batch, we do not allow destructive content changes to objects during a batch. Instead, new content must be written to fresh (unused) extents on disk. (This decision is consistent with trends in modern file systems designs.) Metadata changes are bufferred in memory.

The call openBatch(sessionId, objName) starts a new batch on the object named objName. If objName does not exist, a new empty object is created and given this name (this is the only way to create an object in Guardat). The call returns a batch id that links later calls to the batch and the session. Subsequently, the call readBatched(batchId, off, len, buf) reads len bytes of the object starting at logical offset off in the object and returns the result to the buffer buf. The **read** rule of the object's policy is evaluated before writing to buf; if it denies access, the call fails. This enforces data confidentiality. Note that we allow byte-level addressing on objects.

An object is updated by reusing content from its *current* version and adding fresh content to create a *new* version. The call reuse(batchId, off, len, off') takes content in the logical range [off,off+len-1] from the current version and inserts it at offset off' in the new version (insertion is purely a metadata operation). The call fresh(batchId, off, len, buf, off') writes len bytes from buffer buf to the extent starting at byte number off on disk and adds the resulting extent to the new version at logical offset off'. Before writing the extent, Guardat checks that it is not occupied by any object (including the object being modified). The new version of the object may be given a new policy with the call setPolicy(batchId, new_policy).

The updates in a batch are committed with the call endBatch(batchId). Guardat evaluates the **update** rule of the object's policy before committing the new version. This enforces data integrity. The **update** rule has access to the current and new content of the object, as well as relevant metadata, e.g., the offsets and lengths of reads and writes in the batch. Additionally, if the policy has been updated, Guardat evaluates the **setpolicy** rule of the object's policy; this protects the policy from unauthorized changes.

The call destroy(objName) deletes objName and all its metadata. The object's **destroy** policy authorizes the call. The call also requires that objName be *empty*. This design, following our goal of concise policy representation, ensures that the integrity policy of an object is represented entirely in the **update** permission; **destroy** only controls removal of the object's name from Guardat metadata.

Content caches Two Guardat caches buffer object content for use in policy evaluation. There is a per-session cache of two types of records: (obj,off,len,content), where content is the sequence of len bytes stored starting at offset off within the object obj; and (obj,((off_1,len_1),...,(off_n,len_n)),hash), where hash is the 32-byte SHA-256 hash over the bytes stored at the specified ordered list of (off,len) extents within the object. The session cache reflects the current committed content of the referenced objects. In addition, there is a separate per-batch cache of records (off,len,content) and ((off_1,len_1),...,(off_n,len_n)),hash) which represent the new uncommitted content of the object manipulated within the batch. Records are inserted into the session cache as a side-effect of the readBatched() call, while records are inserted into the batch cache as a side-effect of the fresh() call. Flags to these calls indicate whether content, hashes or neither should be inserted (these flags are described in Appendix A). When a batch commits, any records in the batch cache are moved into the session cache, and any existing session cache records they supersede are evicted. When a batch aborts, the records in the batch cache are discarded.

All relevant object content must exist in the caches before policy evaluation, else access is denied. We rely on the untrusted client for this: The client must set appropriate cache flags in readBatched() and fresh() calls. This is in line with our principle of economy in design: denying access for lack of content in the cache is *safe*, whereas searching for that information on disk is inefficient and can easily lead to DoS attacks. Policy-aware wrapper libraries could alleviate the need for adding cache flags in every application's code.

Certificate API The call setCertificate(certificate) forwards a third-party certificate to Guardat for use in subsequent policy evaluations, whereas getNonce() returns a fresh nonce, which can be embedded in a subsequent certificate. Third-party certificates are described further in Section 3.2.

The call attest(objName, nonce) returns a Guardat-signed certificate that attests the existence

of objName, its extents and its policy. Optionally, the certificate also includes a hash of any of the object's data in the session cache. The attestation certificate embeds a client-provided nonce, which is useful for preventing replay attacks in protocols built over Guardat. The **read** policy rule authorizes this call.

Backwards compatibility For compatibility with existing systems, Guardat supports the standard block-device API calls read(blk,cnt,buf) and write(blk,cnt,buf). The read() call reads cnt blocks sequentially from disk starting at block blk and returns the data in buffer buf. The write() call is dual. In executing these calls, Guardat uses its metadata to find all objects that intersect the extent being read or written. It evaluates the respective **read** or **update** policy rule of all these objects, and fails with an error if any evaluation denies access. Disk blocks not associated with any object can be accessed without restriction through the read() and write() calls. Hence, Guardat may be configured to selectively protect only a part of a storage disk. Also, Guardat can interoperate with existing, unmodified file systems using an application library. More details can be found in Appendix C.

3.2 Guardat policy language

Clients specify object protection policies in an expressive and simple declarative language. Each object's policy contains four *rules*, one for each of the permissions **read**, **update**, **destroy** and **setpolicy**. Each rule specifies the *conditions* under which the respective permission holds. In Section 3.1, we explained which API calls evaluate each of these rules.

A rule has the form (perm :- conds) and means that permission "perm" is granted if the conditions "conds" are satisfied. The conditions "conds" consist of *atomic facts* connected with conjunction ("and", written $\wedge$) and disjunction ("or", written $\vee$). Operationally, policy rules are clauses of constrained Datalog, with all atomic facts in conditions treated as external [23]. Datalog is a standard foundation for writing access policies [7, 12, 34], known for its clarity, high-level of abstraction and ease of implementation.

Each atomic fact contains a predicate that relates object names, content, public keys, extent lists, etc. to each other. The expressiveness of the Guardat policy language stems from the wide range of available predicates. *Universal predicates* are available in all policy rules. The predicate $\mathsf{session\_is}(K)$ checks that the ongoing session is authenticated with the public key K and $\mathsf{object\_name\_is}(O)$ means that the object being accessed has name O. The predicate $(\mathrm{obj}, \mathrm{off}, \mathrm{len})$ says R provides access to the session cache. It means that a record in the session cache states that object obj has content R at offset off. Similarly, $(\mathrm{obj}, ((\mathrm{off}_1, \mathrm{len}_1), \ldots (\mathrm{off}_n, \mathrm{len}_n)))$ hasHash H provides access to hashes in the session cache. Through these predicates, an object's policy may test the content of another object. We find this useful in representing many policies, including mandatory access logging (Section 4).

Additionally, *contextual predicates* provide information specific to a policy rule. In **read**, this information includes the length of the read and its logical and physical offsets. As a result of such fine-grained information, confidentiality policies may be specified at the granularity of bytes. In **update**, contextual predicates provide information about the current and new extents of the object, the current and new object sizes, and access to the batch cache through the predicates $(\mathrm{off}, \mathrm{len})$ willsay R (the new content at offset off will be R) and $((\mathrm{off}_1, \mathrm{len}_1), \ldots, (\mathrm{off}_n, \mathrm{len}_n))$ willHaveHash H (the new bytes stored in the list of (off,len) pairs will have hash H). This facilitates rich integrity policies that correlate old and new object content as well as content across two different parts of an object. Again, we find this handy for many policies, including mandatory access logging. All available contextual predicates are listed in Appendix B.

Finally, Guardat policies may contain arbitrary *uninterpreted predicates* that are established through signed third-party certificates. These include time-server certificates to establish clock time. When a third-party certificate is provided to Guardat through the setCertificate() API call, Guardat checks its signature using standard certificate chain verification [11] and stores its content and its signer's public key in a certificate cache. This cache is available

during policy evaluation, through two types of uninterpreted predicates:

• Public key binding, $\texttt{key\_is}(k, a)$, which states that public key k has attribute a. For example, a may be "TimeServer", suggesting that k is a time server's public key. The corresponding certificate must be signed by a certifying authority (CA) or its delegatee.

• Signed relation, k signs $r(t_1, \ldots, t_n)$ at t: There is a certificate verified by public key k and received at timing counter value t, which contains the relation $r(t_1, \ldots, t_n)$. (The suffix "at t" is optional; the timing counter is explained below.)

The policy designer and certificate issuers must agree on the meaning of the attribute a in the first point and of the relation r in second point. Guardat treats both a and r as bitstrings. Section 4 illustrates this further.

To enforce time-sensitive policies, Guardat relies on time-server certificates and an internal *timing counter*. When a time-server certificate is received, its cache entry is stamped with the value of the timing counter. Later, clock time can be estimated by adding the difference of the then-value of the timing counter and the value of this stamp to the time mentioned in the time-server certificate. This timing counter need not be very precise because it can be reset periodically to prevent a large drift. Whenever the timing counter is reset, all time-server certificates must be evicted from the cache.

To prevent certificate replay attacks, each certificate must include a Guardat-generated nonce, obtained through the API call getNonce(). Guardat waits for a certificate containing a nonce it generates for a small period only. This wait time is an upper bound on the delay between the issuance of a certificate and its acceptance by Guardat and, hence, also an upper bound on the difference between the clock time estimated by Guardat and the clock time known to a time server. Nonces may be created using a pseudorandom number generator.

Following our principle of economy in design, Guardat does not include logic to contact third-parties to obtain relevant policy certificates. Instead, populating the cache with relevant certificates before access is the responsibility of the Guardat client. If required certificates are missing, access is denied. (When a certificate issuer is offline, access to objects that rely on certificates from that issuer may be denied, but access to other objects remains unaffected.)

4 Policy examples

We illustrate the capabilities of Guardat by presenting several example policies. For brevity, we introduce the following convention to omit default policy rules: If the rule for the **read** or **update** permissions is omitted, then the permission is always allowed and if the rule for the **setpolicy** or **destroy** permission is omitted, then that permission is never allowed.

Protected executables For a binary file, it is desirable to defend against accidental or malicious overwriting or rollback to a prior version. A representative Guardat policy to accomplish this is shown below. The policy states that the new content of the binary after any update must be signed by the software vendor (called "Vendor") as being version 10 or later. Moreover, any changes to the policy must be certified with the administrator's key, k_{ad}.

update :- object_name_is(O) $\wedge$ new_length_is(L) $\wedge$
($0, L$) willHaveHash Nh $\wedge$ $\texttt{key\_is}(K,$"Vendor") $\wedge$
K signs $\texttt{ok\_hash}(O, N, Nh)$ $\wedge$ $(N \geq 10)$
setpolicy :- object_name_is(O) $\wedge$ new_pol_hash_is(Nph) $\wedge$
k_{ad} signs $\texttt{good\_policy}(O, Nph)$

The first rule allows an update to the object only if there is a public key K belonging to "Vendor" (condition $\texttt{key\_is}(K,$"Vendor")), which signs that the object's new content hash, Nh, is the Nth version of the binary (condition K signs $\texttt{ok\_hash}(O, N, Nh)$) and $N \geq 10$. The uninterpreted predicates $\texttt{key\_is}(K,$"Vendor") and K signs $\texttt{ok\_hash}(O, N, Nhash)$ are verified from client-provided certificates signed by a certifying authority and the vendor, respectively. Because the vendor's certificate contains a single hash over the entire object content, atomic update transactions are needed to satisfy this policy.

The second rule allows a change to the binary's policy only if the hash of the new policy, called Nph, has been certified by the administrator (condition k_{ad} signs $\texttt{good\_policy}(O, Nph)$).

Properties: As long as the integrity of the vendor's and admin's keys is maintained, files protected by the policy cannot be overwritten except with content signed by the vendor and version ≥ 10, even if the entire system is compromised (write integrity). Moreover, a client operating system can make sure it executes only trusted executables despite a compromised storage service (read integrity) as follows: before executing a binary, it obtains an attestation certificate for the file from the Guardat, verifies the policy and object name (full path name) in the certificate, and compares the hash of the data delivered by the storage system with that manifest in the certificate.

Append-only logs The following policy specifies an append-only file that may be extended by anyone but modified otherwise (e.g., rotated) only by an administrator identified by the public key k_{ad}. The policy would prevent accidental or malicious record deletion from system log files.

$$\begin{aligned}&\textbf{update} \text{ :- } \mathsf{session\_is}(k_{\mathsf{ad}}) \vee \\ &(\mathsf{old\_length\_is}(Lo) \wedge \mathsf{new\_length\_is}(Ln) \wedge (Ln \geq Lo) \wedge \\ &\ \mathsf{updated\_locations\_are}(M) \wedge \mathsf{disjoint}(M, [0, Lo]))\end{aligned}$$

The policy allows an update if either the session is authenticated by the administrator (condition $\mathsf{session\_is}(k_{\mathsf{ad}})$) or the object's new length Ln exceeds its current length Lo and the first Lo bytes of the object are not modified.

Properties: As long as the integrity of Guardat and the admin's key is maintained, append-only writes are ensured for any file with the policy, even if the entire remaining system is compromised.

Storage lease Backup files can be protected from accidental or malicious modification for a fixed period of time by attaching the following policy to them.

$$\begin{aligned}&\textbf{update} \text{ :- } \mathtt{key\_is}(K, \text{“TimeServer”}) \wedge \\ &\ K \text{ signs } \mathtt{time}(T) \text{ at } T_i \wedge \\ &\ \mathsf{time\_is}(T_j) \wedge (T + T_j - T_i > \mathsf{endT})\end{aligned}$$

The policy allows modification to the object only if the current time exceeds a pre-determined time endT. In detail, there should be a key K belonging to a time server (condition $\mathtt{key\_is}(K, \text{“TimeServer”})$), which issued a certificate that the clock time was T when the internal timing counter had value T_i (condition K signs $\mathtt{time}(T)$ at T_i), the current counter value is T_j (condition $\mathsf{time\_is}(T_j)$) and the current clock time (calculated as $T + T_j - T_i$) exceeds the lease end time endT.

Properties: As long as the integrity of the time server and its signing key is maintained, a file with this policy cannot be modified before the designated time, even if the system, the admin's and the file owner's private keys are compromised.

Mandatory access logging Legislation and organizational policies often mandate that all access — read and write — to sensitive information like medical records be logged to a separate file. Although application-level solutions to enforce such mandatory access logging (MAL) exist, enforcing the policy in Guardat is desirable because it would result in a smaller trusted computing base. We show here how a MAL policy can be encoded in the Guardat policy language. The representation is non-obvious and contains several straightforward but mundane details, so we focus on the high-level idea, but elide the details. (An alternate design for MAL could add a "logging rule" to the policy language. In line with our design principles of a minimal language that specifies policy but not mechanism, we rejected this design.)

For this exposition, let P be the sensitive object which must be protected by MAL and let L be its log object. We assume that the log object is append-only, through the policy described earlier. The MAL requirement is three-fold: 1) (Completeness) For every read on P, an entry in L should describe *who* read and from *where* in P. For every write, a similar entry must exist in L and it must additionally contain a hash of the content written. 2) (Causality) Given two write entries in L, the order in which they were applied to P should be evident and, similarly for a read and a write entry. 3) (Precision) Call a write entry in L *dangling* if it does not correspond to an actual write on P. Then, either dangling entries should not

be allowed in L or they should be detectable.[1]

We start with an obvious strawman policy for P, which is complete, but does not provide causality and precision. We refine the design later. We define two kinds of entries for L: `may_read`(K, S), which indicates that the client with public key K has potentially read the set S of (off,len) ranges from P; and `change`(K, S, H), which states that content with hash H has been written to the ranges in S. To force logging of reads, we require in the **read** rule of P's policy that if the range R is read by client K, then an entry `may_read`(K, S) with $R \subseteq S$ exist in L. Similarly, write logging could be forced through P's **update** rule.

This strawman policy for P can be expressed in the Guardat policy language because the set R of locations read or updated is available through contextual predicates in the policy language, the client K is available through the predicate is_session(K) and L's content is available through the session cache (predicate says). The policy can also be easily satisfied by the client: Prior to reading or writing, the client could append an appropriate entry to L and have it cached for P's subsequent policy evaluation. Even though this policy satisfies the MAL requirement of completeness, it does not satisfy causality and precision. Nothing in L's policy prevents the client from creating entries that are never used and such entries cannot be distinguished from others (this violates precision). Moreover, nothing in P's policy prevents use of L's entries out-of-order, which violates causality.

To obtain causality and precision, we refine this strawman design. We embed a counter in each entry in L and enforce through L's policy that the counter increase by 1 at each successive `change` entry and remain the same at each `may_read` entry. We enforce through P's policy that the value of the counter in the last `change` entry that has already been applied to P be written at a designated locus in P. Further, the entry used to justify a read must have a counter number that matches the current counter in P. We describe below how we enforce these requirements. Assuming that they have been enforced, both causality and precision are satisfied. Causality holds because the policies just described force that `change` entries apply to P in increasing order of their counter numbers, and that a read corresponding to a `may_entry` is used after all `change` entries with smaller or equal counter numbers have been applied. Precision holds because a `change` entry is dangling if and only if its counter number is higher than the counter in P.

The log's entries are revised to include counter numbers. They take the forms `may_read`(N, K, S) and `change`(N, K, S, H), where N denotes a counter. We reserve a fixed locus in P for a counter, called C. The log is initialized with a dummy entry with $N = 0$ and P is initialized with $C = 0$. We describe relevant policies of L and P in words, omitting symbolic representations for clarity. (We have formally represented these policies in our prototype implementation; experimental results are presented in Section 5.)

L's **update** policy: Only appends are allowed and only entries of the two designated forms may be added. If the added entry has the form `may_read`$(N, \ldots)$, then N must be copied from the previous entry and if the added entry has the form `change`$(N, \ldots)$, then N must be one more than the previous entry's counter. (These requirements can be represented in the Guardat policy language because the previous entry and the new entry are accessible through the session and batch caches, respectively, during evaluation of the **update** rule.)

P's **read** policy: L must contain a `may_read` entry with the same counter number as C and range set larger than the actual range read. (L's relevant entry and C are accessible through the session cache during P's policy evaluation. In particular, C can be referenced because Guardat supports byte-level addressing on objects and the locus of C is fixed in advance. The client is responsible for specifying which entry of L in the session cache satisfies the policy.)

P's **update** policy: L must contain an entry describing the update precisely. The counter in the entry must be one more than C. The update must also increment C by 1. (When evaluating P's policy, L's relevant entry and the old value of C are accessible through the session cache. The new value of C is accessible through the batch cache.)

[1] Dangling read entries are usually not a problem, because it is in the client's interest to establish that it did *not* read certain data and, hence, not create dangling read entries. We also describe later how read entries can be made precise.

Properties: Our policies enforce all MAL requirements.

MAL client: The MAL client must perform some bookkeeping steps to satisfy the MAL policy. Prior to each access on P, appropriate log entries must be created. When creating log entries, flags must be set to buffer them in the content cache for use in P's policy evaluation. A log entry's cache record is also necessary to create the next log entry. Similarly, when C is updated, flags must be set to cache it for use in future policy evaluations. This approach follows from our design principle of placing the burden and complexity of how to satisfy a policy on the untrusted code.

The overhead of creating log entries for updates can be reduced by committing batches less frequently (and, hence, requiring fewer `change` entries). Similarly, the overhead of creating log entries for reads can be reduced by clubbing several anticipated reads into a single `may_read` entry. The performance benefit of these optimizations is substantial and we report on it in Section 5. Applications that cannot accurately estimate their read-sets ahead of time can simply create blanket `may_read` entries that cover the entire object and periodically *commit read-only batches* accompanied by special log entries that specify precisely what has been read in the batch (the precise read set is available to Guardat during end batch, so the log entry's accuracy can be verified). This mode of use requires a second counter in log entries and the sensitive object to count read-only batches. We elide the details here.

5 Experimental evaluation

In this section, we present results of an experimental evaluation of a Guardat prototype implementation in a SAN server.

5.1 Prototype Implementation

Our prototype is a modified iSCSI Enterprise Target (IET) SAN server. IET implements the server-side iSCSI protocol, which provides SCSI block storage access via Ethernet. IET is in production use and available for many Linux distributions, e.g., SUSE, RHEL and Debian.

IET consists of a kernel module, which implements block accesses, and a user-level daemon process, which implements iSCSI management functions. To implement Guardat, we extended the kernel module and added a second user-level daemon process, which implements the metadata structures, Guardat API and policy evaluation. The kernel module performs upcalls to determine if iSCSI block accesses should be allowed. The server has access to an SSD for storing Guardat metadata and one or more magnetic disks for payload data.

The Guardat daemon maintains two B-tree index structures: a block-to-object index to find the object and policy associated with a given disk location (block id), and a name-to-object index to retrieve the object information (set of extents, policy, etc.) given an object name. For performance, the Guardat daemon maintains a write-through DRAM cache of B-tree nodes and policies, backed by the SSD.

When the kernel module receives a disk access request, it passes the access type (read/write) and location (disk offset and length) to the multi-threaded Guardat daemon. The daemon consults the block-to-object index. If the disk location is not associated with any object, the access is granted. Otherwise, the daemon retrieves and evaluates relevant policies. The result is returned to the kernel module. For read requests, the disk read is performed while checking the permission. This may result in some wasted work if the read is denied, but results in lower latency if it is not. During a write request, the disk write must be deferred until the daemon grants the permission.

The Guardat daemon maintains certificate, session and batch caches in DRAM. It denies access based on a policy rule that refers to an evicted cache record. The client may refresh the record using API calls to regain access.

Our prototype's attack surface consists of the IET management interface, the block-device interface, the Guardat interface extensions as well as the policy language. Despite the relatively large IET codebase, which includes a Linux kernel, the resulting attack surface is likely to be significantly smaller than that of the system built on top of Guardat in most cases.

Our Guardat implementation adds less than 13,000 LOC to the existing IET codebase, plus the OpenSSL and glib libraries it relies on.

5.2 Experimental setup

In our setup, the Guardat enhanced IET SAN server (based on version 1.4.20.3-9.6.1) [45] runs on a separate physical server connected to the client via 10Gbit Ethernet links. The client software runs on OpenSuse Linux 12.1 (kernel version 3.1.10-1.16, x86-64). The Linux iSCSI client connects to the IET server, and appears to the Linux ext4 filesystem as a locally connected SCSI block device.

The IET server and the Linux client each run on a Dell Precision T1600 workstation with an Intel Xeon 3.1Ghz quad core CPU (AES and AVX instruction set) and 8GB main memory. The server has a 500GB disk drive with the server OS installation, and two disks that are used for Guardat. Data is stored on a Seagate Barracuda 2TB 7200 rpm hard drive with a 64MB cache [38], and the Guardat metadata is stored on a OCZ Deneva 2 C SLC 60GB (raw 64GB) SSD [28]. Only 4GB of the SSD is actually used for Guardat metadata; the remaining capacity is available for general use by clients.

The openSSL crypto library [29], Intel AES encryption library [20], and Intel's fast SHA256 implementation [21] are used for Guardat cryptographic operations.

5.3 Microbenchmarks

We performed a series of microbenchmarks to quantify the overheads incurred by the Guardat prototype in terms of storage space, read/write latency and throughput, and Flash memory wear.

5.3.1 Space requirements for metadata

First, we quantify the metadata storage requirements. Because the metadata size depends on the structure of the payload data, we analyzed the metadata space requirements for 70,825 filesystem snapshots collected by Agrawal et al. [1]. The snapshots were taken from Windows systems within Microsoft corporation between 2000 and 2004, and contain between 30k and 90k files each with an average file size between 108KB and 189KB. For evaluation purposes, we give each file in each snapshot an integrity policy that disallows modification prior to a given date.

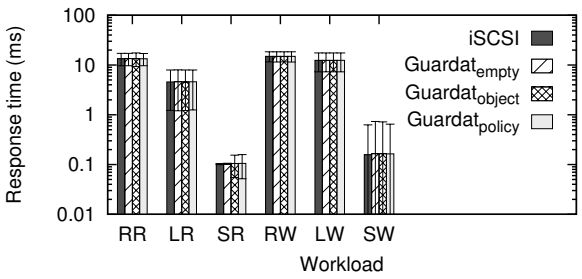

Figure 2: End-to-end I/O latency for synthetic workloads

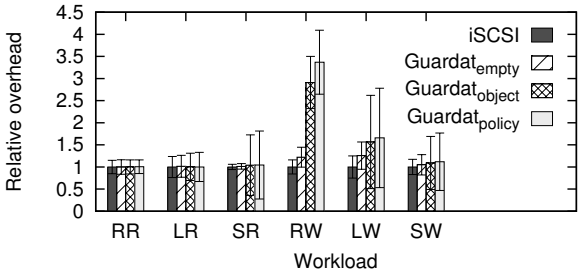

Figure 3: Normalized I/O latency for synthetic workloads excluding disk access latency

As a point of reference, the ratio of solid state to magnetic disk capacity in commercially available hybrid disks is at least 0.8% [39] at the time of this writing. At this ratio, the required metadata can be accommodated in the solid state memory for 99.99% of the snapshots. Newer combinations of Flash/disk devices like Apple's Fusion Drive (128GB Flash/1TB HDD) achieve much higher Flash to disk capacity ratios, and would easily accommodate the metadata for all snapshots.

5.3.2 Read/write latency

Next, we examine the read/write latency of the Guardat prototype under synthetic workloads. For this ex-

periment, we fill the 2TB disk with 3.8 million files, each spanning a single 512KB extent, and compare the read/write latency of the Guardat prototype with the original IET under three different configurations:
Guardat_empty: No objects are protected by a policy. The overheads incurred by this configuration are limited to the cost of communication between the kernel module and the Guardat daemon, and the (negative) check for any policy.
Guardat_object: An "allow all" policy is associated with each object. Each access to a disk block requires the userspace Guardat daemon to lookup the metadata associated with the corresponding object and interpret the null policy.
Guardat_policy: Each object is protected by a policy selected at random from a set of 40,000 different policies, each of which allows access after a past date. The additional overhead includes fetching and interpreting the different policies.

Each configuration is exercised with three different access patterns (**S**equential: blocks accessed in order of increasing block id, **L**ocal: each accessed block chosen randomly within 40,000 block ids of the previous block, **R**andom: each accessed block chosen randomly within the entire disk), and two access types (**R**ead and **W**rite). Each access reads or writes a 512B disk block.

For the different configurations, access patterns and types, Figure 2 show the absolute end-to-end access latency. Five runs were performed with each configuration, for a total of 100,000 accesses. Each run was started at a randomly chosen block id as the starting location. The bars show the average of the total number of measured latencies; error bars indicate the standard deviation.

The Guardat latency overheads for local and random read/write accesses are all negligible (below 1%), because they are dominated by the access latency of the magnetic disk. The Guardat overheads are more noticeable during the much faster sequential accesses (2.9% for **SR** and 4% for **SW** in the Guardat_policy configuration). During sequential read accesses, Guardat can partially hide the policy check latency by issuing the disk read in parallel with the policy check, and squashing the read if the check is negative. Writes, on the other hand, cannot be scheduled safely until the policy check completes, thus the higher overhead. There is room for further optimization, for instance, by caching the results of previous policy evaluations in the kernel module while they remain valid, thus avoiding upcalls in the common case.

To zoom in on the delays introduced by Guardat, Figure 3 shows the measured end-to-end latencies *excluding the disk access latency*, and normalized to the latency of the unmodified IET server without the disk access latency. We see that the overheads relative to the original IET are small, except during random writes and, to a lesser extent, local writes. As noted above, the policy checks cannot be overlapped with the disk access during writes. Moreover, during random and local writes, the Guardat metadata lookups tend to miss the cache more often, leading to average latencies of up to 930us compared to the IET latency of 278us in the Guardat_policy configuration on random writes (**RW**). However, this overhead has little impact on the end-to-end latency, which is dominated by disk access latency, as shown in Figure 2.

5.3.3 Read/write throughput

Next, we examine the read/write throughput of the Guardat prototype, using the same configurations used in the latency experiment. However, instead of issuing 512B accesses sequentially, the test client issues four 64KB requests concurrently, which suffices to achieve maximal throughput in the experiment.

Figure 4 shows the absolute throughput of the various configurations and workloads. The results shown are the averages of 5 runs, each starting at a block id picked randomly within the disk and containing 20,000 accesses. Error bars indicate the standard deviation of all 100,000 accesses. The Guardat overheads for all access patterns are negligible (below 0.6%), because the Guardat policy checks are largely overlapped by disk accesses.

5.3.4 Flash memory wear

Flash memory can endure only a limited number of erase/program cycles. It is important that Flash not wear out before the expected lifetime of the magnetic

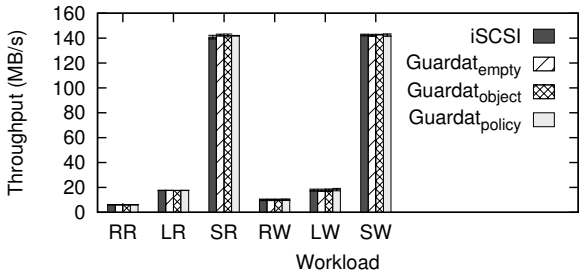

Figure 4: I/O throughput for synthetic workloads

disk; to be conservative, we assume that the Flash must last at least 10 years. The lifetime is influenced by the size of the metadata, the rate of metadata updates (more updates require more Flash writes), and the Flash capacity. (A smaller capacity causes the Flash log to wrap around faster and leads to higher utilization, which in turn reduces cleaning efficiency and requires even more Flash writes.)

Under the configuration of Guardat_policy used above in 5.3.2, we keep track of how much wear the Flash experiences while presented with a series of metadata updates, i.e., adding and removing extents to a content file picked at random. Using only 4GB of Flash memory with a nominal lifetime of 100,000 erase/program cycles, we can accommodate up to 19.5M updates per day (225 per second). This is an extraordinarily high update rate and can accommodate even the most write-intensive applications.

5.4 Use case: Web server

Next, we study the capabilities and performance of the Guardat prototype as part of a Web server. The server holds a 220GB static snapshot of English language Wikipedia articles from 2008 [50] and images from 2005 [49], containing 15 million files with an average file size of 15KB and maximum file size of ∼500KB.

We use three different machines connected by 10 Gbit Ethernet links to run the IET storage server, the Apache Web server (version 2.2.23)[4] and Apache HTTPAsyncClient for Java (version 4.0-beta-1)[5]. The Web server either fetches the contents from the mounted iSCSI device or from Linux's buffer cache. Apache does not use a dedicated disk cache.

The HTTP client asynchronously requests HTML pages from the Web server, using a workload based on the actual access counts of Wikipedia pages during one hour on April 1, 2012 [51]. Because our snapshot is much older and had fewer articles at the time, we ignore accesses to non-existing pages. In total, about 350,000 different pages were accessed in the trace, of which 250,000 are part of the 2008 snapshot. Since we do not have access to time stamps, we distributed the individual accesses evenly within an hour, and replayed the first 100,000 page requests.

We use the following Guardat policies: **Static content pages:** Allow updates signed by a fixed owner. We use 40,000 different owner identities and randomly assign them to the content files. **System binaries:** Allow updates signed by a special vendor signature only.

The workload and policies used are particularly challenging for Guardat, due to the large number of small protected files and a disk intensive workload.

Figure 5 shows the average throughput of three runs as a function of the number of concurrent HTTP accesses. Each run loads 100,000 Wikipedia pages. Results are shown for four different configurations:

iSCSI: The plain IET iSCSI implementation.

Guardat_content: Guardat protecting content changes.

Guardat_content+binary: Guardat protecting changes to content and Apache binaries (e.g. apachectrl, httpd, libapr).

There is little difference (less than 1%) between the throughput achieved by the unmodified IET server and the Guardat_content and Guardat_content+binary configurations, confirming that the Guardat overheads are mostly hidden by disk access latencies in these configurations.

To summarize, the use of Guardat in the Web server use case has negligible performance overhead. In terms of functionality, Guardat protects content files from tampering and corruption by unauthorized parties, and prevents intruders from modifying executables to install Trojan horses.

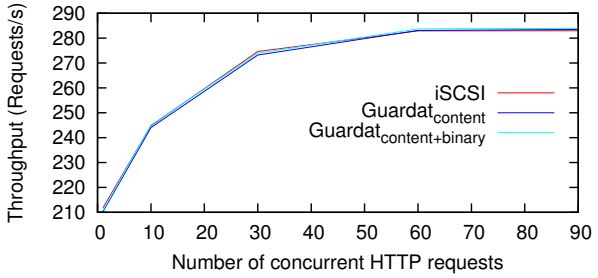

Figure 5: Throughput of 100,000 page loads, as a function of the number of concurrent accesses. Average of five runs, variations were below 0.6%

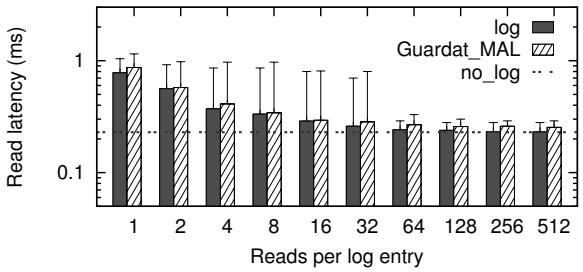

Figure 6: Read latency with MAL, voluntary and no logging

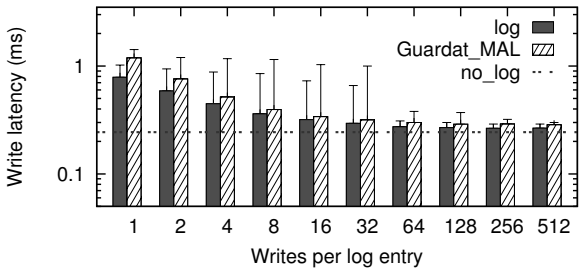

Figure 7: Write latency with MAL, voluntary and no logging

5.5 Use case: Mandatory access logging (MAL)

In the final experiment, we perform accesses to a file with a mandatory access logging policy. The MAL policy requires adding the proper log entry to a separate log file in order for an access to be allowed by Guardat. We use a 64MB file with or without the MAL policy in place. The primary file and log file reside on different disks attached to the same Guardat IET server. The version counter embedded in the primary file is stored in a block of available Flash memory. The client connects to the Guardat device and accesses the file as follows:

no_log: the unprotected file is accessed without any logging.

log: the unprotected file is accessed and the accesses are logged without policy enforcement.

Guardat_MAL: the policy-protected file is accessed and the accesses logged, as enforced by the policy.

Figures 6 and 7 show the access latency for sequential 4KB reads and writes, respectively, of the same location within the file. We vary the number of accesses per log entry. The bars show the average latency of 100,000 accesses and the error bars show the 98th percentile. In the case of a single read/write, voluntary logging increases the latency from 0.20/0.16ms to 0.78/0.79ms, and policy enforcement increases the latency to 0.87/1.19ms. The higher cost of enforced logged writes reflects the need to update the sequence number. In both cases, the synchronous log write contributes a significant part of the overhead; policy enforcement increases the read/write latency by 11.5% and 50.6%, respectively, for individually logged operations. As shown, the cost of MAL can be amortized by logging several operations with a single entry.

6 Conclusion

Guardat enforces confidentiality, integrity and access accounting policies for persistent data at the storage layer, and attests the state of stored objects. The Guardat design is rich enough to enable powerful policies, primitives and applications, yet is amenable to an efficient implementation, as demonstrated by an experimental evaluation. In future work, we intend to extend Guardat to a distributed environment, enabling policy enforcement, tracking and access ac-

counting for objects that migrate among Guardat devices.

References

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3), 2007.

[2] Marcos K. Aguilera, Minwen Ji, Mark Lillibridge, John MacCormick, Erwin Oertli, David G. Andersen, Mike Burrows, Timothy Mann, and Chandramohan Thekkath. Block-level security for network-attached disks. In *Proceedings of the 2nd USENIX FAST*, 2003.

[3] Amazon simple storage service (S3). `http://aws.amazon.com/s3/`.

[4] Apache Software Foundation. Apache http server. `http://httpd.apache.org/`, 2012.

[5] Apache Software Foundation. Apache httpasyncclient. `http://hc.apache.org/httpcomponents-asyncclient-dev/index.html`, 2012.

[6] Apple Inc. Time Machine. `http://www.apple.com/osx/what-is/`.

[7] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *20th IEEE Computer Security Foundations Symposium*, 2007.

[8] Kevin Butler, Steve McLaughlin, Thomas Moyer, and Patrick McDaniel. New security architectures based on emerging disk functionality. *IEEE Security and Privacy*, 8(5):34–41, 2010.

[9] Kevin R. B. Butler, Stephen E. McLaughlin, and Patrick Drew McDaniel. Rootkit-resistant disks. In *ACM Conference on Computer and Communications Security*, pages 403–416, 2008.

[10] Kevin R. B. Butler, Stephen E. McLaughlin, and Patrick Drew McDaniel. Kells: a protection framework for portable data. In *ACSAC*, pages 231–240, 2010.

[11] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280. `http://www.ietf.org/rfc/rfc5280.txt`, 2008.

[12] John DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Security and Privacy (S&P)*, 2002.

[13] Michael Factor, Dalit Naor, Eran Rom, Julian Satran, and Sivan Tal. Capability based secure access control to networked storage devices. In *Proceedings of the 24th IEEE Mass Storage Systems and Technologies (MSST)*, 2007.

[14] Deepak Garg and Frank Pfenning. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[15] Ron Garret. A Time Machine time bomb. `http://blog.rongarret.info/2009/09/time-machine-time-bomb.html`.

[16] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th ACM ASPLOS*, 1998.

[17] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium*, 2004.

[18] Mark Hayakawa. WORM Storage on Magnetic Disks Using SnapLock Compliance and SnapLock Enterprise. Technical Report TR-3263, Network Appliance, 2007.

[19] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter Technical Conference*, 1994.

[20] Intel Corp. AESNI library. `http://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library/`, 2011.

[21] Intel Corp. Fast SHA256. `http://download.intel.com/embedded/processor/whitepaper/327457.pdf`, 2012.

[22] Ramakrishna Kotla, Tom Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. Pasture: Secure offline data access using commodity trusted hardware. In *Proceedings of the 10th USENIX OSDI*, 2012.

[23] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th Symposium on Practical Aspects of Declarative Languages*, 2003.

[24] M. Mesnier, G.R. Ganger, and E. Riedel. Object-based storage. *Communications Magazine*, 41(8), 2003.

[25] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the 23rd ACM SOSP*, 2011.

[26] Microsoft Corp. What is volume shadow copy service? `http://technet.microsoft.com/en-us/library/cc757854(WS.10).aspx`.

[27] Microsoft Corp. Windows Backup and Restore. `http://www.microsoft.com/athome/setup/backupdata.aspx#fbid=l7X90d97alI`.

[28] OCZ Technology Inc. Deneva 2 data sheet. `http://www.oczenterprise.com/ssd-products/deneva-2-c-sata-6g-2.5-slc.html`, 2011.

[29] OpenSSL Cryptographic library. `http://www.openssl.org/docs/crypto/crypto.html`, 2012.

[30] Oracle Corporation. Solaris ZFS. `http://www.oracle.com/us/products/servers-storage/storage/storage-software/031857.htm`.

[31] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[32] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM SIGMOD*, 1988.

[33] Adam G. Pennington, John Linwood Griffin, John S. Bucy, John D. Strunk, and Gregory R. Ganger. Storage-based intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 13(4):30:1–30:27, December 2010.

[34] Andrew Pimlott and Oleg Kiselyov. Soutei, a logic-based trust-management system. In *Proceedings of the 8th International Symposium on Functional and Logic Programming (FLOPS)*, 2006.

[35] E. Riedel, C. Faloutsos, G.A. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6), 2001.

[36] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[37] Seagate Technology LLC. Self-encrypting hard disk drives in the data center. Technical Report TP583, 2007.

[38] Seagate Technology LLC. Barracuda Data Sheet. `http://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barracuda-xt-ds1696.3-1102us.pdf`, 2011.

[39] Seagate Technology LLC. Momentus XT Data Sheet. `http://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/momentus-xt-data-sheet-ds1704-4-1209-us.pdf`, 2012.

[40] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th USENIX OSDI*, 2006.

[41] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX FAST*, 2003.

[42] Storage Work Group of the Trusted Computing Group. Self-encrypting drives take off for strong data protection. `http://www.trustedcomputinggroup.org/community/2010/03/selfencrypting_drives_take_off_for_strong_data_protection`, 2011.

[43] Storage Work Group of the Trusted Computing Group. TCG storage architecture core specification. `http://www.trustedcomputinggroup.org/resources/tcg_storage_architecture_core_specification`, 2011.

[44] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th USENIX OSDI*, 2000.

[45] The iSCSI Enterprise Target project. `http://iscsitarget.sourceforge.net/`, 2011.

[46] Kevin Walsh and Fred B. Schneider. Costs of security in the PFS file system. Technical report, Computing and Information Science, Cornell University, 2012.

[47] Carsten Weinhold and Hermann Härtig. jVPFS: Adding robustness to a secure stacked file system with untrusted local storage components. In *Proceedings of the USENIX ATC*, 2011.

[48] Carsten Weinhold and Herrmann Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM/SIGOPS EuroSys*, 2008.

[49] Wikimedia Foundation. Image Dump. `http://archive.org/details/wikimedia-image-dump-2005-11`, 2005.

[50] Wikimedia Foundation. Static HTML dump. `http://dumps.wikimedia.org/`, 2008.

[51] Wikimedia Foundation. Page view statistics April 2012. `http://dumps.wikimedia.org/other/pagecounts-raw/2012/2012-04/`, 2012.

A Details of Guardat API calls

This section summarizes the Guardat API calls and describes the session handshake protocol.

Session API We describe the handshake protocol that authenticates the client and Guardat to each other and establishes session keys. We use some abbreviations. C and G denote the client and Guardat respectively. For a principal A, K_A denotes its public key and K_A^{-1} denotes its corresponding private key. Guardat's public key K_G is assumed to be known to everyone (through a manufacturer-provided certificate). The client and Guardat use two freshly chosen random nonces, denoted N_C and N_G respectively. $\text{sig}(K, M)$ denotes the signature on message M with private key K. $\text{enc}(K, M)$ denotes the encryption of message M with public key K. (M_1, M_2) denotes concatenation of messages M_1 and M_2. $A \to B : M$ means that A sends message M to B. Our handshake protocol is:

$$\begin{aligned} C \to G :&\ K_C, \text{sig}(K_C^{-1}, \text{enc}(K_G, N_C)) \\ G \to C :&\ \text{sig}(K_G^{-1}, (0, \text{enc}(K_C, (N_C, N_G)))) \\ C \to G :&\ \text{sig}(K_C^{-1}, (1, \text{enc}(K_G, (N_C, N_G)))) \end{aligned}$$

At the end of the protocol, both C and G have seen their nonces signed by the other party; this authenticates them to each other. The sequence numbers 0 and 1 in the second and third messages distinguish the two messages from each other, preventing a man in the middle from causing confusion through replays. The session keys are derived by the client and Guardat using key derivation functions on the concatenation of the two nonces, (N_C, N_G).

The handshake protocol is implemented using two API calls, named handshake1 and handshake2. The first call, handshake1, corresponds to the first message in the protocol and its return value corresponds to the second message. The second call, handshake2, corresponds to the third message and its return value is just a confirmation that the session has been established. (The return value is irrelevant to security of the protocol, but it is needed to tell the client to proceed.)

- handshake1(message): The message should be of the form of the first message in the protocol. If correct, the return value is the second message of the protocol and a session id to link the second call.

- handshake2(sessionId, message): The message should be of the form of the third message in the protocol. The return value is either success or failure. If the value is success, then sessionId is used as the identifier for the rest of the session.

The API call endSession(sessionId) ends a session.

Object API To allow flexible content hashing during readBatched() and fresh() calls, Guardat provides hash computation buffers to which the client can selectively choose to add data during these calls. Once the client has added all the data it needs to a buffer, it finalizes the buffer, which moves the hash of the content in the buffer to either the session cache or the batch cache (depending on whether the hash buffer has current or new object content). In the implementation, each non-finalized hash buffer is an open hash computation and newly appended content is hashed immediately. Hash buffers are accessible through the following API.

- initHash(batchId, currOrNew): Initialize a new hash buffer for the object associated with the open batch batchId. The Boolean flag currOrNew indicates whether the buffer will hold a hash of the object's current version or its updated version. This choice is enforced in calls that add to the buffer. Returns a unique identifier for the hash buffer, hashId.

- closeHash(hashId): Finalize the hash buffer hashId and move the hash in it to the session cache if the buffer has current object content or to the batch cache if the buffer has new object content.

API calls to start and end batches and to read and update objects were described in Section 3.1. We summarize them below with details of caching flags that we omitted from Section 3.1.

- openBatch(sessionId, objName): Start batch on object with name objName in session identified by sessionId. If objName does not exist, create it. Returns a new batchId on success.

- readBatched(batchId, off, len, buf, cacheFlags, cacheIntervals, hashId): Read len bytes from logical offset off of the object associated with batchId and return the result in buf. The cacheFlags indicate whether or not the read content should be added to the session cache and whether or not it should be added to a hash buffer. If either is the case, then cacheIntervals specifies which logical ranges of the read content need to be added. If content is to be added to a hash buffer, hashId identifies the buffer. This call evaluates the **read** policy rule.

- reuse(batchId, off, len, off'): Take content in the logical range [off,off+len-1] from the current version of the object associated with batchId and insert it at offset off' in the new version.

- fresh(batchId, off, len, buf, off', cacheFlags, cacheIntervals, hashId): Write len bytes from buffer buf to the extent starting at byte offset off on disk and add the resulting extent to the new version at offset off'. The arguments cacheFlags, cacheIntervals and hashId have roles similar to those in readBatched(), but the batch cache, and not the session cache, is affected.

- setPolicy(batchId, new_policy): Set the policy of the new version to new_policy.

- endBatch(batchId): Close the batch identified by batchId. Evaluates the **update** policy rule

and, if the policy is being modified, also the **setpolicy** rule.

- destroy(objName): Destroy the metadata of the empty object named objName. Evaluates the **destroy** policy rule.

Certificate API The certificate API generates nonces, attests content and forwards third-party certificates to Guardat for policy evaluation.

- getNonce(sessionId): Return a new nonce that is to be embedded in a third-party certificate later.
- setCertificate(sessionId, certificate): Provide certificate for use in later policy evaluation.
- attest(sessionId, objName, attestFlags, hashId, nonce): Attest objName's metadata and optionally a hash of selected content. The argument attestFlags specifies which of the following need to be attested: the list of physical extents, policy, policy hash and content hash. If content hash is to be attested, hashId points to an entry in the session cache which has that hash. Guardat returns a single certificate containing all requested vectors and the client-provided nonce. The **read** policy rule of objName is evaluated.

Backwards compatibility Standard block-device API calls read(blk,cnt,buf) (read cnt disk blocks starting at block blk into buffer buf) and write(blk,cnt,buf) can be used to read and write disk extents. The **read** or **update** policy rule of all extents that overlap the accessed extents is evaluated.

B Details of the Guardat policy language

We summarize in this section predicates available in the Guardat policy language. In addition to these predicates, policies may use any uninterpreted predicates established through third-party certificates.

The following universal predicates are available in all policy rules.

- object_name_is(O): The object being accessed has name O.
- session_is(K): The current session has been authenticated with public key K.
- time_is(T): The internal timing counter is currently T.
- guardat_key_is(K): The public key of this Guardat installation is K.
- Arithmetic, string comparison: $t_1 == t_2$, $t_1 < t_2$, $t_1 \leq t_2$.
- is_subset(R_1, R_2): Range set R_1 is a subset of range set R_2.
- disjoint(R_1, R_2): Range sets R_1 and R_2 are disjoint.
- (obj, off, len) says C: The content at offset off in object obj is C. (Based on the session cache.)
- (obj, $((\text{off}_1, \text{len}_1), \ldots, (\text{off}_n, \text{len}_n))$) hasHash H: The bytes stored in the given list of (off,len) pairs in object obj have hash H. (Based on the session cache.)

Specific contextual predicates are available in each policy rule. We list these below, categorized by the policy rules. The **destroy** rule admits no contextual predicates.

Read rule The following predicates are available in the **read** policy rule.

- isAttest(): True if the **read** rule is being evaluated in an attest() call, and false otherwise.
- access_locations_are(R): The set of logical (off,len) pairs read from the object is R.
- access_physical_extents_are(E): The set of physical extents read is E.
- access_length_is(L): The number of bytes read is L.

Update/setpolicy rule The rules for **update** and **setpolicy** evaluate in the same call, endBatch(), and, hence, they admit the same contextual predicates with only one exception that is shown later. When the **update** rule evaluates in the call write(), some of these predicates always evaluate to false. These predicates are marked with *.

- isCreate*(): True if and only if the batch executed on an object that did not already exist.
- current_length_is(L): The length of the current version of the object is L bytes.
- new_length_is*(L): The length of the new version of the object is L bytes.
- current_physical_extents_are(E): The current version of the object spans the set E of physical extents.
- new_physical_extents_are*(E): The new version of the object spans the set E of physical extents.
- updated_locations_are(M): The set of logical (off, len) pairs updated during the batch is M.
- read_locations_are*(R): The set of logical (off, len) pairs read during the batch is R.
- current_pol_hash_is(H): The current policy has hash H.
- new_pol_hash_is*(H): The new policy has hash H.
- (off, len) willsay* C: The new content at offset off is C. (Based on the batch cache.)
- $((\text{off}_1, \text{len}_1), \ldots, (\text{off}_n, \text{len}_n))$ willHaveHash* H: The new bytes stored in the given list of (off,len) pairs have hash H. (Based on the batch cache.)

The following predicate is available only in the **update** policy rule, not the **setpolicy** rule, and can be used to distinguish evaluation of **update** in the endBatch() call from that in the write() call.

- isWrite(): True if and only if evaluation is part of the write() call.

C Compatibility with existing filesystems

We sketch how our Guardat prototype can be used with an existing, unmodified filesystem, which is not aware of Guardat and issues only ordinary block reads and writes. In this compatibility mode, applications are linked with a library, which implements the standard POSIX filesystem interface, and provides additional operations for applications to authenticate, set a policy for a file, provide certificates required by policies, and request attestations. The library interacts with the Guardat userspace daemon directly and makes API calls to associate block read and write operations issued by the filesystem with an object, client session and batch. We note that the library is untrusted and does not require extra privileges. In particular, the library only executes Guardat calls on behalf of applications that the applications are allowed to execute themselves.

To determine if a block read operation is allowed, the Guardat daemon maps the requested block number to the associated object (if one exists) using the block-to-object B-tree. To further be able to map the read operation to an authenticated session, we impose the limitation that only a single batch or session can be open for a given object at any given time in compatibility mode.

Write operations may refer to an extent not currently associated with any object. (When a file is extended, the filesystem allocates new blocks.) Therefore, prior to writing new data to a file, the application library provides the Guardat daemon with a vector of hashes of aligned blocks containing the new data. This vector enables the daemon to associate subsequent writes issued by the filesystem with the correct object, offset, session and batch. In order to avoid ambiguity, two blocks with the same hash value may not be outstanding at the same time. The daemon enforces this condition by temporarily refusing to accept a block hash vector that contains an element that is already present among the current set of outstanding vectors.

When the kernel module receives a write command, it computes the hash of blocks to be written, and

sends the hash to the daemon along with the request. The daemon matches write requests with the list of hashes provided by the compatibility library. Computing hashes in the kernel avoids sending data from the kernel to the userspace daemon.

The compatibility mode has limitations. As mentioned above, only a single session and batch may be active for any given object, which can lead to some loss of performance in workloads with concurrent accesses to the same file. Also, because the filesystem is unaware of Guardat, any attempt by the filesystem to relocate a file with an associated integrity policy may fail. As a result, defragmentation of an unmodified filesystem requires a modified defragmentation utility. Object data encrypted with a session key must be communicated between library and the Guardat daemon without going through the iSCSI driver, to avoid polluting the filesystem's buffer cache with session-encrypted data. These limitations can be lifted by modifying a filesystem to use the extended Guardat API, which is a subject of ongoing work.