Guardat: Enforcing data policies at the storage layer

Anjo Vahldiek-Oberwagner[1] Eslam Elnikety[1] Aastha Mehta[1] Deepak Garg[1]
Peter Druschel[1] Rodrigo Rodrigues[2] Johannes Gehrke[3,4] Ansley Post[5]

[1]MPI-SWS [2]NOVA LINCS/Nova University of Lisbon [3]Cornell [4]Microsoft [5]Google

Abstract

In today's data processing systems, both the policies protecting stored data and the mechanisms for their enforcement are spread over many software components and configuration files, increasing the risk of policy violation due to bugs, vulnerabilities and misconfigurations. Guardat addresses this problem. Users, developers and administrators specify file protection policies declaratively, concisely and separate from code, and Guardat enforces these policies by mediating I/O in the storage layer. Policy enforcement relies only on the integrity of the Guardat controller and any external policy dependencies. The semantic gap between the storage layer enforcement and per-file policies is bridged using cryptographic attestations from Guardat. We present the design and prototype implementation of Guardat, enforce example policies in a Web server, and show experimentally that its overhead is low.

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. Data processing 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.

In today's systems, the confidentiality and integrity of persistent data depend on many components of a system, as well as appropriate operator action. Applicable data protection policies may be implicit in code and configuration, and their specification and enforcement spread over different subsystems and multiple software layers, increasing the risk of policy violation due to bugs, vulnerabilities and misconfigurations. Reasoning about and debugging the policy in effect for a given collection of files can be challenging.

Guardat addresses these challenges by allowing users, developers and administrators to specify file protection policies *declaratively, concisely and separately from code*. This makes it easy to reason about policies and to write and debug them. Guardat enforces policies in the *storage layer* by mediating I/O. Compared to policy enforcement in higher layers, this reduces the scope for accidental policy circumvention due to bugs, misconfiguration, or operator error. Guardat also provides *cryptographic attestations* of files and their policies to bridge the semantic gap between per-file policies and block-level enforcement. Structurally, Guardat adds a declarative policy interpreter, per-file policy metadata, crypto and enforcement logic to the storage layer.

Guardat policies cover data integrity, confidentiality and access accounting. Following are example policies that can be enforced by Guardat to mitigate important threats: A policy can protect executable files from trojan horses by allowing only updates signed by a trusted party; an append-only policy can protect system logs from corruption and tampering; a policy preventing modifications for a specific period of time can protect backup data from accidental deletion or corruption; a policy requiring an authenticated secure session to read data can protect the confidentiality of a user's private data; and, a policy may stipulate that accesses to a file require a corresponding record be added to an append-only log file (mandatory access logging).

Unlike techniques that protect data in a system's main memory, including information flow control [30, 69], trusted computing [40], secure operating systems and secure hypervisors [12, 51], Guardat has low overhead, is readily deployable, and requires few (if any) changes to applications and operating systems. While Guardat does not protect transient data in main memory, it offers a low-cost safety net to protect persistent data from a wide range of threats, including a compromised system. Specifically, Guardat can protect the integrity of a system's long-term state, which can be recovered by rebooting. Moreover, Guardat can ensure the confidentiality of persistent data as long as the keys of users authorized to access the data are not compromised.

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3238-5/15/04.
http://dx.doi.org/10.1145/2741948.2741958

For instance, suppose a Guardat-enabled web server is compromised. Guardat can prevent access to private content and manipulation of content files without the owner's key. It can prevent the manipulation of log files and access control lists, and the installation of trojans without the sysadmin's key. Thus, Guardat minimizes damage during the attack (because it ensures the confidentiality of private data), and enables admins to quickly recover by patching the exploited vulnerability and rebooting (because it ensures the integrity of the server's persistent state, i.e., its executables, content files, access control lists and log files).

The contributions of this work include (1) the Guardat architecture and interface; (2) a declarative policy language, which balances expressiveness and efficient evaluation; (3) a prototype implementation of Guardat in the iSCSI Enterprise Target SAN server; and, (4) an experimental evaluation, which shows that Guardat policies can be enforced with low overhead. The following section provides an overview of the Guardat design and presents example policies. Section 3 describes the design of Guardat and its policy language in more detail. Section 4 presents experimental results with a prototype based on the iSCSI IET SAN server. We discuss implementation alternatives in Section 5, cover related work in Section 6 and conclude in Section 7.

2. Guardat overview

This section provides an overview of Guardat's design, and illustrates its capabilities with some example policies.

Principles. The design was guided by four principles:
1) Guardat policies are attached to files, separate from code, and specified in a custom *declarative policy language*. Therefore, the policy for a file's data can be specified concisely in one place and audited easily.
2) Guardat enforces policies in the *storage layer* to minimize the risk of policy circumvention. Our implementation of Guardat in a SAN server, for instance, allows a scalable configuration where policies are enforced by block servers in a machine room, while client computers and the enterprise network are untrusted.
3) Guardat policies state merely *what* accesses are allowed under which conditions, leaving it to untrusted code *how* to demonstrate compliance with a policy. This separation keeps the policy language small and policies concise, while shifting complexity to untrusted software and overhead to client computers.
4) Guardat relies on *cryptographic file attestations* to bridge the semantic gap between per-file policies and block-level enforcement. By requesting an attestation of a file's policy, name and content hash, an application can verify that Guardat associates data and policy correctly, independent of the filesystem or its metadata.

Overview. Guardat's program logic, called the Guardat controller or GDC, is integrated with a storage block device

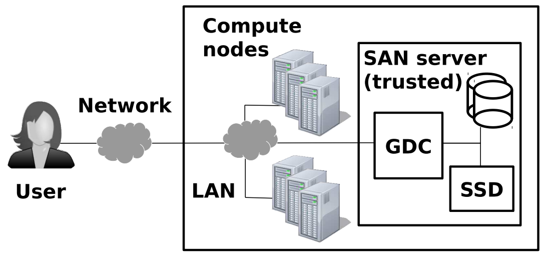

Figure 1. Guardat implementation in a SAN server

and enforces policies on every read and write. The GDC extends the standard block-device interface with a *file-level interface*, which allows higher software layers to (a) create, delete, read and update files using simple transactions, (b) associate policies with sequences of storage blocks that represent a file, (c) cryptographically authenticate and establish secure sessions, (d) provide credentials and other evidence of policy compliance, and (e) obtain attestations on stored files and their policies. The file-level interface can be used by a Guardat-aware filesystem, or by an application library in combination with a legacy filesystem via IOCTL calls.

Guardat maintains its own metadata for policy-protected files, allowing it to enforce file-level policies at the block level without depending on a filesystem. For each policy-protected file, Guardat maintains an ordered list of extents[1] that hold the file's data, a unique numeric identifier, a textual name string (typically used to store the file's pathname(s) - one for each hard link), and a pointer to a policy in effect for the file. The set of unique numeric identifiers forms a flat namespace, while the set of names typically encode a conventional namespace hierarchy maintained by an untrusted filesystem (file names). Each file can have its own policy but, typically, a collection of files share the same policy.

Besides one or more primary storage devices, Guardat 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 [50]. To authenticate itself as a legitimate Guardat device and sign attestations, the GDC includes a private key and a corresponding manufacturer-provided public key certificate.

Implementation, threat model and scope. In this paper, we focus on an implementation of Guardat within a storage area network (SAN) server for use in a data center (see Figure 1). We discuss implementation alternatives in Section 5.

The GDC, metadata and data must be protected from unauthorized access and undetected tampering. In our implementation, the SAN server includes the GDC, data and metadata storage devices, and is assumed to be physically protected, e.g., in a machine room with restricted access. In this scenario, Guardat policies are enforced regardless

[1] An extent is a sequence of storage blocks with consecutive indices.

of bugs, misconfigurations, or security incidents outside the SAN server, including incidents on any number of client machines, and regardless of actions by employees without access to the machine room.

We make standard assumptions about policies: correct policies must be installed when data is first stored, and external dependencies of policies like time servers, client authentication keys, and admin authentication keys must be trustworthy (in particular, admin authentication keys should be stored offline and protected physically). Under these assumptions, Guardat defends against threats to confidentiality and integrity of stored data. In addition, Guardat can protect the integrity and confidentiality of files transferred between Guardat devices, and between a Guardat device and a client device through a secure channel. 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 Guardat devices with independent failure modes.

2.1 Policy examples

Next, we illustrate Guardat's capabilities by presenting several example policies. We begin with a brief primer on the policy language; a more detailed description of the language follows in Section 3.3.

A Guardat policy contains four *rules*, one for each of the permissions **read**, **update**, **destroy** and **setpolicy**. Abstractly, the **read** rule represents the file's confidentiality policy; the **update** rule encodes the file's integrity policy; the **destroy** rule governs when the file's name can be recycled; and the **setpolicy** rule restricts policy changes.

Each rule specifies the *conditions* under which the respective permission holds. 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 *predicates* (e.g., fileNameIs) connected with conjunction ("and", written $\wedge$) and disjunction ("or", written $\vee$). Identifiers starting with uppercase letters (e.g., F in fileNameIs(F)) are variables. A variable stands for an arbitrary value and is instantiated during the evaluation of the first predicate in which it appears. Operationally, policy rules are clauses of Datalog [31]. Datalog is a standard foundation for access policies [6, 14, 42] known for its clarity, high-level of abstraction and ease of implementation.

In the following policy examples, if the **read** or **update** rule is omitted, then the permission is always allowed and if a **setpolicy** or **destroy** rule is omitted, then that permission is never allowed.

Protected executables. For an executable file, it is desirable to prevent 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 executable after any update must be signed by the software vendor (called "Vendor") as being version 10 or later. Moreover, any policy changes must be certified with the administrator's key, k_{ad}.

$$\begin{array}{l}\textbf{update} \text{ :- } \mathsf{fileNameIs}(F) \wedge \mathsf{fileNewLenIs}(L) \\ \quad \wedge\ (0,L)\ \mathsf{willHaveHash}\ Nh \wedge \mathsf{keyIs}(K, \text{“Vendor”}) \\ \quad \wedge\ K\ \mathsf{signs}\ okHash(F,N,Nh) \wedge (N \geq 10) \\ \textbf{setpolicy} \text{ :- } \mathsf{fileNameIs}(F) \wedge \mathsf{filenewPolIs}(Nph) \\ \quad \wedge\ k_{\mathsf{ad}}\mathsf{signs}\ goodPolicy(F,Nph)\end{array}$$

The first rule allows an update to the file only if there is a public key K belonging to "Vendor" (condition keyIs (K, "Vendor")), which signs that the file's new content hash, Nh, is the Nth version of the executable (condition K signs okHash(F, N, Nh)) and $N \geq 10$. The predicates keyIs(K, "Vendor") and K signs okHash(F, N, Nh) are verified from client-provided certificates signed by a certifying authority and the vendor, respectively. The second rule allows a change to the executable's policy only if the hash of the new policy, called Nph, was certified by the administrator (condition k_{ad} signs goodPolicy(F, Nph)).

Properties: As long as the integrity of the vendor's and the administrator'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). A variant of this policy can limit content on the system's boot sector to vendor-signed boot images, thus protecting the boot sequence from trojans and rootkits.

Append-only logs. The following policy restricts a file such that it may be extended by anyone but modified in-place (e.g., rotated) only by an administrator identified by the public key k_{ad}. The policy prevents accidental or malicious manipulation of system log files.

$$\begin{array}{l}\textbf{update} \text{ :- } \mathsf{sessionKeyIs}(k_{\mathsf{ad}}) \\ \quad \vee\ (\mathsf{fileCurrLenIs}(Lc) \wedge \mathsf{fileNewLenIs}(Ln) \\ \quad \wedge\ (Ln \geq Lc) \wedge \mathsf{txUpdatedExAre}(M) \\ \quad \wedge\ \mathsf{listsAreDisjoint}(M, [0, Lc]))\end{array}$$

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

Properties: As long as the integrity of the admin's key is maintained, the policy is enforced even if the system is compromised.

Protected backup. Backup files can be protected from accidental or malicious modification for a fixed period of time using the following policy.

$$\begin{array}{l}\textbf{update} \text{ :- } \mathsf{keyIs}(K, \text{“TimeServer”}) \\ \quad \wedge\ K\ \mathsf{signs}\ time(T)\ \mathsf{at}\ T_i \wedge (T + T_i > \mathsf{endT})\end{array}$$

The policy allows modification to the file only if the current time exceeds a pre-determined time endT. To enforce

time-based policies, Guardat relies on signed certificates from time servers and a short-range internal timer. In detail, the policy says that there should be a key K belonging to a time server (condition $\mathsf{keyIs}(K, \text{“TimeServer”})$), which issued a certificate T_i timer steps ago (condition K signs $time(T)$ $\mathtt{at}$ T_i) and the current time (calculated as $T + T_i$) exceeds the backup's expiration 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 read and write access to sensitive information like medical records be logged. A mandatory access logging (MAL) policy can be enforced by Guardat. Let P be a sensitive file that must be protected by MAL and let L be its log file. We assume that L is append-only, enforced by 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 or a read and a write entry in L, the order in which they were applied to P should be evident. 3) (Precision) Write entries in L that do not correspond to actual writes on P should be detectable. (Note: Requiring precision for reads would be pointless, because it is impossible to tell whether the data obtained in a read reached the user application before a crash. Instead, the presence of a read entry in L shows that the user application intended to and could have read the corresponding version of P.)

A policy that enforces these requirements can be encoded in the Guardat policy language. The representation has several straightforward but mundane details described in a technical report (TR) [58]. Here, we sketch the high-level idea. (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.)

First, P's **update** policy requires that a version number C_P embedded in P is incremented during each modification of P. Second, P's **update** and **read** policies require that a record describing the write or read operation on P, respectively, exists in L, with a version number equal to the current value of C_P. Third, L's **update** policy does not allow modification of records whose version numbers are less than or equal to the current value of C_P. The resulting policy satisfies the MAL requirements.
Properties: The policies enforce MAL even if the system is compromised.

Other policy idioms. Many other common policies can be expressed in Guardat. Examples include: (a) Role-based policies where access depends on the client's role in an organization (certificates can relate clients to roles), (b) Blacklist (whitelist) policies where access is denied (allowed) if the client's identity exists in a sorted file (the file's sortedness can also be enforced by a Guardat policy), and (c) History-based policies where access depends on past events that are visible to Guardat. The latter can be enforced by recording events in a dedicated log file and allowing access to the data file only when the log file is in certain states. The MAL policy is a simple history-based policy that allows access when the event of creating an appropriate log entry has occurred.

Expressiveness. As these examples demonstrate, the Guardat policy language can express rich state- and content-based policies like MAL, which prior declarative policy languages cannot. However, the language has limitations. It disallows recursively-defined predicates and, hence, cannot express layouts defined by iteration or recursion, e.g., it cannot express that the content of a file be well-formed XML. Such constraints may be checked by a trusted external verifier using certificates to communicate between the verifier and Guardat. A complete description of the policy language and its semantics is given in Section 3.3.

3. Guardat design

In this section, we describe the Guardat design in more detail, including the Guardat command interface and its use, as well as the policy language.

3.1 Overview

We begin with an overview of how a filesystem interacts with Guardat. The (untrusted) filesystem assigns names and storage blocks to a file and translates file requests into block requests using its metadata, as usual. Guardat maintains its own shadow metadata to look up the file and policy associated with a block request securely and efficiently. Guardat also assigns its own unique file identifiers, which can be reused only under policy control.

File attestations tie the GDC's view of a file as a sequence of extents to an application's view of a named file, thereby removing the need to trust the filesystem and its metadata. By requesting an attestation after a file is written or read, an application can verify that its view of the file is identical to the GDC's. Guardat has support for sparse files. The current design assumes that a block is assigned to at most one file; block sharing to support de-duplication, for instance, could be added easily.

3.2 Guardat commands

Guardat extends the standard block device interface with means to establish sessions, create, update and delete files, install policies, provide evidence of policy compliance, and obtain attestations. Untrusted code uses the interface to request file access while demonstrating to Guardat that the access satisfies the file's policy. Failure to demonstrate compliance will cause Guardat to deny the access. In the following, we describe the functionality provided by the interface. A full list of commands (17 in total) can be found in a technical report (TR) [58].

Session interface. A user application (also called a *client*) interacts with Guardat in a *session*. A secure, authenticated session must be used to access files whose policy requires client authentication. To access other files, no explicit session is required. Such use is conceptually treated as part of a default, untrusted session.

A session is established with a standard 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 client's public key (which acts as a client identifier) becomes available during every policy evaluation in the session; hence, Guardat can enforce policies that restrict access to a specific user. At the end of the handshake, Guardat returns a unique session identifier that links later commands to the session. Guardat can work with any client-side infrastructure for creating, managing and distributing public keys.

Transaction interface. Rich policies may require more than one read or write operation to transition a file from one compliant state to another. For instance, a file's integrity policy may require that each update increments an embedded version counter. For this purpose, Guardat supports *transactions* consisting of a sequence of reads and updates on a *single* file. Transactions are atomic: either all the updates are persisted or they are all discarded. Policies may refer to both the current and new content of a file in a transaction, as well as the content of other files. The policy is checked once at the end of the transaction, which commits if the policy check succeeds, and aborts otherwise.

We find this design useful in encoding policy state machines and access-accounting policies, as illustrated in Section 2.1. However, the design comes with a trade-off: To avoid buffering a potentially unbounded number of updates during a transaction, Guardat forbids destructive updates as part of a transaction. Instead, new content must be written to fresh (not currently allocated to a policy-protected file) extents on disk. This choice mirrors modern file system designs with copy-on-write block allocation, e.g., in WAFL, ZFS, and Btrfs [8, 23, 39]. Outside a transaction, destructive writes succeed if allowed by the policy. Guardat metadata changes are buffered in memory during a transaction.

Content caches. Guardat policies may be contingent on the current content of one or more files and the proposed new content of the updated file in the context of a transaction. To enable the efficient evaluation of such policies, two Guardat caches hold file content for use in policy evaluation. A per-session cache contains entries that refer to current file contents, either as a sequence of bytes at a given file offset and length, or as the hash of such a sequence. A per-transaction cache contains the same types of entries but refers to tentative updates to a file. Entries are added to the cache as a side-effect of read and write commands with appropriate flags. When a transaction commits, any entries in the transaction cache are moved into the session cache, and any existing session cache entries they supersede are evicted. When a transaction aborts, the entries in the transaction cache are discarded. To satisfy a policy that refers to current or pending file content, untrusted client code is expected to fill appropriate cache entries by issuing read/write commands before attempting a transaction commit.

Certificate interface. Cryptographically signed certificates represent facts asserted by a trusted third party, for instance, the wall clock time as reported by a trusted time server or the presence of a file on another Guardat device. The certificate interface has commands to obtain a fresh nonce to be included in a third-party certificate, and commands to add a signed third-party certificate to the Guardat cache for use in subsequent policy evaluations. Third-party certificates are described further in Section 3.3.

The call attest(fileName, nonce) returns a GDC-signed certificate that attests the existence of a file with its (set of) pathname(s), extents and policy. Optionally, the certificate may also include a hash of any of the file's contents. The attestation embeds a client-provided nonce. The **read** policy rule authorizes this call.

Replication/migration interface. A set of commands allow untrusted client software to securely manage the replication and migration of policy-protected files among Guardat devices, without access to their cleartext contents. A file copy succeeds only if the file's policy allows it, and if the integrity of the file's contents, name and policy are maintained during the transfer. The *pickle* operation invoked at a source Guardat device encrypts a file and its policy for a specific target Guardat device, while the *unpickle* operation installs the file at the target Guardat device. An attestation from the target Guardat device can then be used to prove to the source device that the file resides on the target device. A file's policy controls if, when and where a file can be migrated or replicated.

Application library. Guardat applications are linked with an untrusted library, which extends the POSIX API with commands to set policies, provide authentication credentials and certificates, and request attestations. The library also interposes the existing POSIX file API to perform actions required to satisfy a file's policy. It interacts with the GDC through IOCTL calls. We provide more details about an application library for a specific use case in Section 4.4.

Example usage. As an example, we show the sequence of steps required to update an executable file protected by the policy described in Section 2.1. First, a software update application (supd) supplies the required vendor certificate, which is passed by the Guardat library to the GDC to be cached. When supd opens the executable file for writing, the library starts a transaction with the GDC, and arranges that the hashes of all subsequent writes are added to the transaction cache. When supd is done writing and closes the file, the library asks Guardat to commit the transaction, which causes the GDC to evaluate the policy and commit if successful. Otherwise, the commit fails and the file is not modified.

Filesystem interoperability. Full interoperability with Guardat requires modest filesystem modifications to add session ids to the buffer cache tags for secure sessions, to associate write commands with appropriate transactions, and to enable policy-compliant file reallocation/defragmentation. In our prototype, we modified Linux's Btrfs for this purpose (the details of the Btrfs implementation are beyond the scope of this paper). However, unmodified filesystems can be used with many policies. In fact, all policies described in this paper except MAL operate with an unmodified ext4 filesystem. When an unmodified filesystem is used, the application library provides additional hashes of written data to enable the GDC to associate newly allocated blocks with files, while ext4 uses the standard block read and write commands. We refer the interested reader to our technical report (TR) for more details [58].

3.3 Guardat policy language

A brief overview of the Guardat policy language and several examples demonstrating its expressiveness were presented in Section 2.1. Here, we provide details of the language.

Types. The policy language supports three numeric types (boolean, integer and float), content hashes (SHA256), strings, public keys, lists of extents (each element of an extent list stores the logical byte offset within the file, physical block address and the length), variables and predicates.

Predicates. The Guardat policy language is based on standard Datalog but omits recursively-defined predicates for simplicity. Its expressiveness stems from custom predicates (40 in total) that are listed in Table 1. We divide the language's predicates into several categories. *Relational, arithmetic and list predicates* codify standard data operations like addition and subtraction of numeric types and disjointedness of extent lists. *Access predicates* provide the physical block addresses, the logical byte offset and the number of bytes accessed, giving policies control over block-level accesses outside of transactions. *Session predicates* provide authentication information for the current session and the current value of the internal timer. *File predicates* provide the accessed file's metadata (file name, length, extents and policy hash). *Transaction predicates* provide information about the metadata and policy updates during a transaction. *Content predicates* provide access to the per-session and per-transaction

Relational, arithmetic and list predicates			
eq(x,y)	x==y or x<-y	add(x,y,z)	x=y+z
neq(x,y)	x!=y	sub(x,y,z)	x=y-z
lt(x,y)	x<y	mul(x,y,z)	x=y*z
gt(x,y)	x>y	div(x,y,z)	x=y/z
le(x,y)	x<=y	rem(x,y,z)	x=y mod z
ge(x,y)	x>=y		

listGet(l, i, (o, b, len))	(o, b, len)==l(i) where i∈{0,. . .,|l|-1}
listLen(l, len)	len == | l |
listIsMember(l, x)	x ∈ l
listIsSubset(l1, l2)	l2 ⊆ l1
listsAreDisjoint(l1,l2)	l1 ∩ l2 == ∅
listIsPrefix(l, p)	l == [p | S] where S is suffix and | concatenates
listIsSuffix(l, s)	l == [P | s] where P is prefix and | concatenates
Access predicates (outside transactions)	
accStartBlkIs(b)	access starts at block b
accOffIs(o)	access offset at byte o
accLenIs(len)	access length is len
Certificate predicates	
keyIs(k, d)	Public key k is a signing authority for domain d (established by a standard certificate chain)
k signs rel($x_1, \ldots, x_n$) at T	k signed the relation rel($x_1, \ldots, x_n$) T counter ticks ago (only with nonce)
Session predicates	
sessionKeyIs(k)	k == current session's client authentication key
File predicates	
fileNameIs(s)	s == pathname of file
fileCurrLenIs(x)	x == file length
fileCurrExAre(l)	l == list of the file's extents
fileCurrPolIs(h)	h == file policy's hash
Transaction predicates	
txUpdatedExAre(l)	l == {x | x ∈ WriteSet}
txReadExAre(l)	l == {x | x ∈ ReadSet}
txReuseExAre(l)	l == CurrExtents ∩ NewExtents
txIsPickle(k)	current tx prepared pickled data for identity k
txIsUnpickle(k)	current tx holds unpickled data from identity k
fileNewLenIs(x)	x will be the new file length
fileNewExAre(l)	l will be the new list of file's extents
fileNewPolIs(h)	h will be the new file policy's hash
Content predicates	
(f,off,len) says rel($x_1, \ldots, x_n$)	$x_1, \ldots, x_n$ is the tuple at off,len in file f
(off,len) willSay rel($x_1, \ldots, x_n$)	ditto for the updated content of the current transaction
(f,off,len) hasHash h	hash of file f's content at off,len equals h
(off,len) willHaveHash h	ditto for the updated content in the current transaction

Table 1. Guardat policy language predicates

content caches. Finally, *certificate predicates* provide information about cached third-party certificates.

Third-party certificates. Section 2.1 illustrates several predicates that are verified and asserted by third parties (predicates keyIs, okHash, time). Such predicates are provided to Guardat as cryptographic certificates, signed by the respective third-parties. Guardat verifies every certificate provided to it using standard certificate chain verification [13] and makes the certificate's content and its signer's public key available to the policy interpreter through the predicate signs. Guardat relies on untrusted clients to provide relevant certificates before access. If certificates required for policy evaluation are missing, access is denied. When a certificate issuer is offline and previous certificates time out, access to files that rely on certificates from that issuer may be denied, but access to other files remains unaffected. To prevent replay attacks, each certificate must include either a recent Guardat-generated nonce or an explicit expiration time (time server certificates must contain a recent nonce).

Semantics. Guardat's policy language uses standard Prolog semantics (Datalog is a sublanguage of Prolog). These semantics have been studied extensively, both in the context of access control [6, 42] and more generally, so we review them only briefly. Predicates are evaluated left-to-right in a rule. Variables are implicitly existentially quantified. If a variable appears in many predicates joined by conjunction ($\wedge$), it gets bound to a concrete value (public key, file name, time point, etc.) when the first predicate in which it appears evaluates. Of all policy clauses joined by disjunction ($\vee$), only one has to evaluate affirmatively to allow access. The language is implemented using a stack machine, which is standard for languages like Prolog [60]. We describe the evaluation time complexity of the language in Section 4.2.

Usability. Declarative languages similar to ours are widely used as policy languages (e.g., XACML, SecPAL, Binder, SD3, and KeyNote [6, 7, 14, 28, 36]) due to their simplicity, which enables a very concise policy specification, as well as a very small interpreter, minimizing the TCB. A standard, imperative language could be used instead, but at the loss of the above mentioned benefits. Several security-oriented operating systems incorporate similar policy languages (e.g, Taos, Nexus and Singularity [51, 66, 67]). More broadly, the source of our policy language, Datalog, is an industry-strength alternative for SQL. Datalog is also used for other purposes like declarative specification of network protocols (languages NDlog and Overlog [32, 33]).

We believe that policies will be written mostly by privacy and security experts. For any application, there will be a limited number of basic useful policies, and most system administrators, users or developers will merely select from a library of policies, perhaps with minor customization or parameterization.

3.4 Summary

We close with a summary of the key ideas in Guardat's design. Attaching declarative policies to files enables the concise specification and auditing of rich data policies in one place. Enforcing policies at the block layer enables scalable system configurations that minimize the risk of policy circumvention. With our SAN server implementation, for instance, policies can be enforced in the machine room, while clients and enterprise network are untrusted. Guardat bridges the semantic gap between file-level policies and block-level enforcement by maintaining its own metadata associating blocks, files and policies, and providing cryptographic attestations for end-to-end verification of the policy in place for a file's data. Certificates allow policies to refer to facts attested by trusted third parties like time servers, other Guardat devices, or external verifiers. Transactions support rich policy state machines, where a transition between two policy-compliant states requires multiple read and write operations. The Guardat policy language and interface shift the burden of demonstrating policy compliance to untrusted software, thereby keeping policies concise and policy evaluation efficient, while shifting overhead to client computers.

4. Experimental evaluation

In this section, we describe a prototype implementation of Guardat within a SAN server. We evaluate its performance on a series of microbenchmarks, and in the context of a web server that enforces several of the policies explained in Section 2.1.

4.1 Prototype

Our prototype is based on the iSCSI Enterprise Target (IET) SAN server, which implements the server-side iSCSI protocol and provides SCSI block storage access via Ethernet. IET is in production use and available for many Linux distributions. It 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, which implements the Guardat interface and evaluates policies. The kernel module performs upcalls to determine if iSCSI block accesses should be allowed. The server is configured with a small SSD for storing Guardat metadata, as well as one or more magnetic disks or SSDs for the payload data.

The Guardat daemon maintains two B-tree index structures on the metadata SSD: a block-to-file index to find the file and policy associated with a given block number, and a name-to-file index to retrieve the file information (set of extents, policy, etc.) given a file id. For performance, the Guardat daemon maintains a write-through DRAM cache of B-tree nodes and policies, backed by the SSD. Updates are persisted on the SSD during a transaction commit.

When the kernel module receives a block access request, it passes the access type (read/write) and location (disk offset, length) to the multi-threaded Guardat daemon, which consults the block-to-file index. If the block location is not associated with a policy-protected file, the access is granted. Otherwise, the daemon evaluates the policy and returns the result to the kernel module. For read requests, the block read is scheduled while checking the permission to reduce latency. During a write request, the block write must be deferred until the Guardat daemon grants the permission.

To reduce the number of upcalls and policy evaluations, the kernel module maintains a cache of previous policy evaluation results of the form $\langle extent, permissions\rangle$. To feed this cache, the Guardat daemon always returns the largest extent encompassing the presently requested block for which the same permissions hold. The cache is flushed when a policy changes.

To support Guardat, we added about 20,000 LOC to the existing IET codebase, plus the OpenSSL and glib libraries that Guardat relies on. Despite the relatively large codebase of our proof-of-concept prototype, which includes a minimally configured Linux kernel (no remote login or user program execution), its attack surface is small. It consists of the IET management interface, the block-device interface, the Guardat interface extensions and the policy language.

Experimental setup. The Guardat-enhanced IET SAN server (based on version 1.4.20.3-9.6.1) [57] runs on a server connected to the client via one 10Gbit Ethernet link. 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 filesystems 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 E3-1225 3.1Ghz quad core CPU (AES and AVX instruction set) and 8GB main memory. The server has a 500GB disk drive dedicated to the server OS installation. Data blocks are stored either on a separate Seagate Barracuda 2TB 7200 rpm hard drive with a 64MB cache [49], or on a 512GB Samsung SSD [44]. The Guardat metadata is stored on a OCZ Deneva 2 C SLC 60GB (raw 64GB) SSD [37]. Only 4GB of that SSD is actually used for Guardat metadata. Guardat uses a DRAM metadata cache that holds 100K b-tree nodes.

The OpenSSL crypto library [38], Intel AES encryption library [25], and Intel's fast SHA256 implementation [26] are used for Guardat cryptographic operations.

4.2 Microbenchmarks

Read/write latency. We first examine the read/write latency of the Guardat prototype under synthetic workloads, using either a HDD or an SSD as the block store.

We use a 2TB image with 3.8 million files, each spanning a single 512KB extent. To use the same metadata size and access pattern on the HDD and SSD despite their different capacities, we access files allocated in the first 512GB of the image only. We compare the Guardat prototype with the original IET under three different configurations. **iSCSI:** The plain IET iSCSI implementation. **Guardatempty:** Guardat is used, but no files are protected by a policy. **Guardatfile:** An "allow all" policy is associated with each file. **Guardatpolicy:** Each file 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.

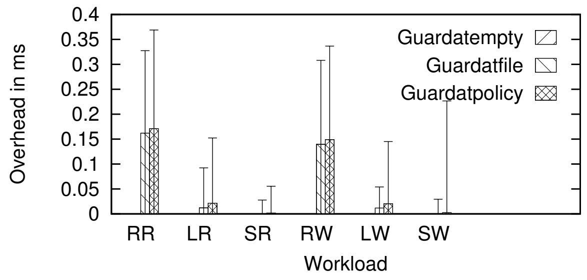

Figure 2. Absolute Guardat latency overhead

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 on the entire disk), and two access types (**R**ead and **W**rite). Each access reads or writes a single 512 byte block. For each access pattern in each configuration, we perform five experimental runs; each run has 20,000 accesses (a total of 100,000 accesses for each configuration). Each run starts at a randomly chosen location on the disk.

Figure 2 shows the absolute Guardat latency for metadata lookup and policy evaluation in the experiment. (Note that these results are independent of whether a SSD or HDD is used as the data store.) Error bars indicate the standard deviation. In the **Guardatempty** case, the userspace daemon spends 2.5µs upon the first access to check for a (non-existent) policy. A single entry is then added to the kernel module's permission cache, covering the entire disk and granting universal permission (no policies). Subsequent requests are granted from this cache at near zero cost making all bars invisible in Figure 2. In the other cases, the time spent by Guardat depends on the locality of the workload, which determines the hit rate in the kernel permission cache and the daemon's DRAM cache of b-tree nodes. For example, the Guardat overhead averages 2.2µs for **Guardatpolicy** in the sequential access cases, since caching is very effective. However, under the random workloads, **Guardatpolicy** has to perform on average 0.7 reads on its metadata SSD per check, increasing its overhead to 160µs. The variable number of metadata SSD accesses required for a given policy check explains the high variance.

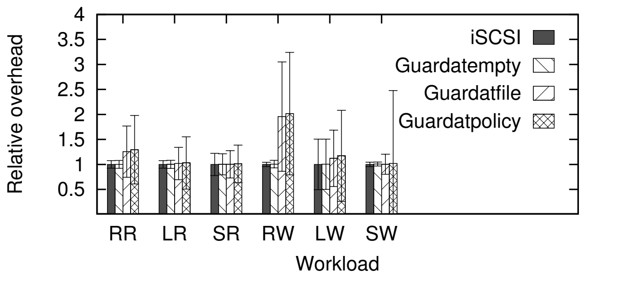

Figure 3. Latency with an SSD, relative to iSCSI

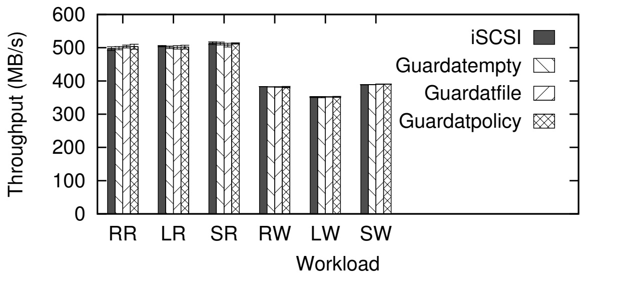

Figure 4. SSD I/O throughput

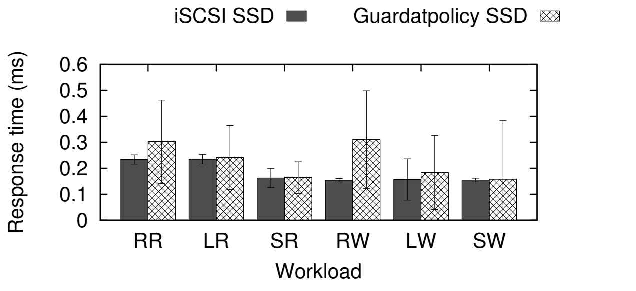

Figure 5. Absolute latency with SSD

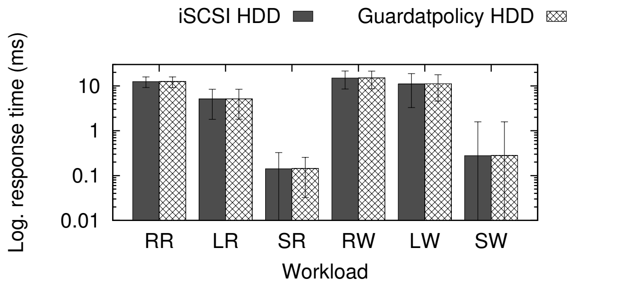

Figure 6. Absolute latency with HDD

Figure 3 shows the resulting average access latency with the SSD, relative to the plain iSCSI. Even with the fast SSD as a block store device, the Guardat latency overhead is generally low, but significant for random writes (2-fold increase). The fact that our block store SSD performs random writes much faster than random reads (153µs versus 233µs), presumably due to write buffering in its internal DRAM, combined with the fact that the policy check cannot be overlapped with the access during a write, contributes to this high relative overhead.

Note that the random access workload is extreme: The SSD block store device is very fast, we are measuring the latency of tiny accesses (512 bytes) at random locations over the entire disk, and there are many files and policies. Increasing the request size reduces the overhead. For example, with a 4K request size, the overheads decrease from 29.3% for **RR** and 101.6% for **RW** to 17.7% and 96.1%, respectively. With 128K requests, the overheads go further down to 0.9% and 23.5%, respectively. Moreover, as we show next, even under this workload the SSD retains much of its latency advantage over the HDD with Guardat, and Guardat's throughput overhead is very low on both the SSD and the HDD.

Figures 5 and 6 compare the absolute latencies achieved on a HDD and SSD with and without Guardat. Despite Guardat's large relative overheads for purely random writes, the SSD retains its towering latency advantage on such accesses over the HDD (note that the y-axis is different for SSD and HDD). With the magnetic HDD, the Guardat latency overheads for all configurations are negligible (below 1%).

Compared to a locally attached SSD, the average latency of a remotely connected iSCSI SSD increases by 0.051 ms, a little more than one network round trip (0.047 ms).

Read/write throughput. Next we examine the read/write throughput of the Guardat prototype, using the same configurations as the latency experiment. The test client issues four 128KB requests concurrently, which is sufficient to achieve maximal read and write throughput in the baseline iSCSI in all cases. For each access pattern in each configuration, we run the throughput test 5 times; each run issues a total of 20,000 accesses and starts at a random block within the disk.

Figure 4 shows the absolute throughput with the SSD. The results shown are the averages of 5 runs, where error bars indicate the standard deviation. The Guardat overhead is below 2% for all access patterns with the SSD. With the HDD, the overheads are in the same range.

The high latency overhead on random writes does not significantly affect the throughput because policy evaluation for different requests can be performed in parallel by the multi-threaded Guardat daemon, and overlapped with disk and SSD accesses to metadata and blocks.

Moreover, compared to a locally attached SSD, the throughput overhead is at most 3% for all iSCSI and Guardat configurations and workloads.

I/O performance summary. While Guardat adds little latency to HDD accesses and SSD accesses with good locality, it has a noticeable latency overhead on small, purely random writes to an SSD. However, this overhead diminishes quickly with larger request sizes and more locality, and can be overlapped with concurrent accesses, so that the SSD's throughput is not affected.

Policy size	**Domain size**				
	1	**2**	**4**	**8**	**16**
1	2.2	3.4	5.8	10.7	20.4
2	4.6	10.4	28.9	95.1	345.8
3	7.0	24.0	121.2	770.5	5,518.1
4	9.4	50.9	485.3	6,156.4	88,319.3
5	11.9	104.9	1,951.3	49,234.7	1,411,800.8

Table 2. Evaluation latency in µs for varying policy size (number of predicates and variables in the policy) and domain size (maximum number of cache entries)

Policy evaluation overhead. Consistent with Datalog, the theoretical worst-case evaluation time for a policy rule is in $O(m \cdot D^n)$, where m is the size of the rule (number of predicates), D is the size of the domain (bounded by the size of the Guardat cache) and n is the number of variables in the rule. In Table 2, we show the measured policy evaluation time for synthetic policies designed to extricate worst-case execution from our policy interpreter. D varies along columns of the table and m and n vary along rows ($m = n$ in all experiments). The results match the expected complexity $O(m \cdot D^n)$. The table indicates (correctly) that policy evaluation could be a substantial bottleneck for some policies but we do not observe this bottleneck in practice. The average policy evaluation latency of the most complex policy evaluated, MAL (Section 4.5) is only 27.7µs, even though the policy has $m = 4$, $n = 4$ and $D = 40$. This is because of a careful implementation of the policy interpreter to consider more recent cache entries first. Our other example policies evaluate even faster; the average evaluation time of the time-based policy from the latency experiment configuration **Guardatpolicy** is only 3.7µs.

Space requirements for metadata. 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. The snapshots are more than 10 years old at the time of this writing. Because the average file size in today's systems has likely increased, however, our analysis of Guardat's metadata requirements relative to the size of the data is conservative.

The required metadata can be accommodated in a solid state memory of 0.8% of the data size for 99.89% of the snapshots. As a point of reference, even commercially available hybrid disks provide at least 0.8% Flash [50] at the time of this writing. And, newer combinations of Flash/disk devices achieve much higher Flash to disk capacity ratios. For example, Apple's Fusion Drive [4] has a ratio of 128GB Flash for a 1TB HDD, which can easily accommodate all the snapshots. In all our experiments, which use other data sets, the metadata fit into only 0.2% of the data size.

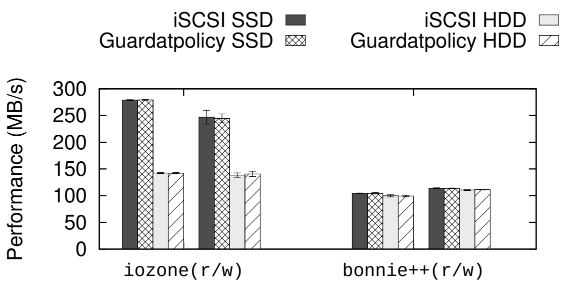

Figure 7. FS benchmarks read and write (r/w) performance

Flash memory wear. Because Flash memory can endure only a limited number of erase/program cycles, we must check that the SSD used to store metadata will not wear quickly. 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, 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 **Guardatpolicy** used above, 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. Enterprise environments typically deploy single-level cell (SLC) Flash memory, which has a nominal lifetime of 100,000 erase/program cycles. Using only 4GB of such memory we can accommodate up to 19.5M updates per day (225 per second). This is an extraordinarily high update rate that can accommodate even the most write-intensive applications. Cheaper multiple-level cell (MLC) and triple-level cell (TLC) Flash memory with nominal lifetimes of 10,000 and 1,000 erase/program cycles would support up to 1.95M and 195,000 metadata updates per day, respectively.

4.3 File system benchmarks

Next, we measure the performance of the Guardat prototype using the standard file system benchmarks `iozone` v3.429 and `Bonnie++` v1.03. The block store was formatted under ext4. `iozone` uses four worker threads to write 1GB sequentially to four separate files.[2] Later, each worker performs a sequential read of the file they previously wrote. Similarly, `Bonnie++` writes then reads 1GB each to 16 files. Figure 7 shows the performance for the baseline and Guardat under the **Guardatpolicy** configuration. The results shown are the averages of 5 runs and error bars indicate the

[2] We used the command `iozone -i 0 -i 1 -r 512k -I -c -e -T -t 4 -s 1g -F files`

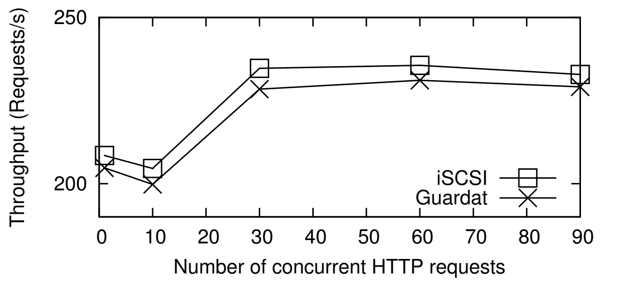

Figure 8. Web server throughput

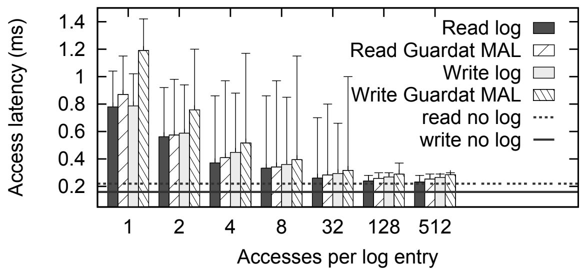

Figure 9. Latency with MAL, voluntary and no logging

standard deviation. The Guardat overheads are below 1.0% for both benchmarks on both the HDD and the SSD. Note that `Bonnie++` uses the C library functions getc and putc to perform file reads and writes, and is therefore unable to saturate the disks.

Similar to the throughput experiment, the iSCSI SSD results are close to those achieved with a locally attached SSD (at most 3.5% lower).

4.4 Use case: Web server

Next, we consider the performance of the Guardat prototype as part of a modified Apache Web server. The server holds a 220GB static snapshot of English language Wikipedia articles from 2008 [64] and Wikimedia images from 2005 [63], containing 15 million files with an average file size of 15KB and maximum file size of ~500KB. 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 [65]. 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 to protect the server's persistent state: **Content:** Require content updates signed by owners. We randomly assign one of 40,000 owners to each content file. **Executables/Config:** Require that updates to executable and configuration files be signed by the administrator. **Log files:** The Apache log files can only be appended, except with an administrator key used to rotate the log. To satisfy the log file policy, we added a total of 51 lines of code to Apache. This extra code issues Guardat commands to send content hashes to Guardat and flush application and filesystem caches (fflush & fsync) before every log file update. The policies protecting the content, executables and configuration files do not require any modifications to Apache.

Figure 8 shows the average throughput of three runs as a function of the number of concurrent HTTP accesses, for plain iSCSI and Guardat (standard deviation is below 0.5%). Each run loads 100,000 Wikipedia pages. The throughput overhead of the Guardat configuration over the unmodified iSCSI server is 1.95% at 60 concurrent requests, where iSCSI reaches its peak throughput, and always within 2.7%. This result shows that the Guardat overheads mostly overlap with other activities in the Web server. The 100,000 page requests result in approximately 350,000 Guardat reads, for an average of 3.5 reads per page. This shows that a substantial number of reads reach the Guardat device and are not absorbed by the filesystem buffer cache. In addition, Apache writes 2.7MB of log records in 170 transactions under the append-only policy. There are no updates to content, executables and configuration files, nor log rotations in the workload, but policies must still be checked during each access.

In terms of functionality, Guardat protects content, logs, configuration and executable files from tampering by unauthorized parties, which we confirmed through fault injection experiments.

4.5 Mandatory access logging (MAL)

In our final experiment, we perform accesses to a file with our mandatory access logging (MAL) policy. The policy requires an appropriate entry in a separate log file for an access to be allowed by Guardat. We use a 64MB primary file with or without the MAL policy in place. The primary file and the log file reside on different HDDs attached to the same Guardat IET server. The version counter embedded in the primary file is stored in Flash memory not used by Guardat. The client connects to the Guardat device and accesses the primary file in three different configurations. **no log:** file accessed without any logging and enforcement. **log:** accesses logged without policy enforcement. **Guardat-MAL:** accesses logged and policy enforced by Guardat.

Figure 9 shows the average access latency for 100,000 sequential 4KB reads and writes of the primary file, varying the number of accesses per recorded log entry from 1 to 512. Error bars indicate the standard deviation. In the case of a single access per log entry, enforcing the MAL policy increases the read/write latency by 11.5% and 50.6%, respectively, over voluntary logging. The higher cost for logged writes compared to reads reflects the need to update the version number. Both costs can be reduced by issuing version counter updates, log writes, and primary file accesses in parallel. Moreover, as shown in the figure, the cost of MAL can be amortized by logging several accesses in a single log entry, and approaches the cost of completely unlogged accesses for 512 accesses per log entry.

5. Discussion

Enforcement layer. Guardat enforces policies at the storage layer, which may seem surprising, given that policies are associated with files, not blocks. However, enforcement at this layer minimizes the risk of circumvention, and makes it easy to physically protect the trusted Guardat components in a machine room. For instance, Guardat is robust even to compromised SAN clients issuing illicit block requests. An implementation at a higher layer (e.g., NAS fileserver, VMM or client OS layer) is possible, but would would extend trust to additional, and likely more distributed, components. Moreover, Guardat is able to bridge the semantic gap between files and blocks without relying on the filesystem and its metadata, through its file-level interface and file attestations.

Implementation alternatives. Besides the Guardat prototype implementation in a SAN server, a GDC can be implemented within the microcontroller of a hybrid disk for use in an individual machine. A possible third implementation of the GDC is a trustlet within a VMM or operating system, isolated using trusted hardware features like Intel SGX [27] or ARM TrustZone [5].

In each implementation, the GDC, metadata and data must be protected from unauthorized access and undetected tampering. In our prototype, the SAN server must be physically protected, e.g., in a machine room where access is restricted to trusted staff. When the GDC is implemented as part of a microcontroller embedded in a hybrid disk, 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, Guardat policies are enforced as long as the microcontroller has not been physically tampered with (the disk enclosure may be 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. An implementation as a trustlet has similar security properties, except that the GDC executes on the main CPU and trust is derived from this CPU's isolation capabilities.

Support for databases. Some applications and systems increasingly rely on databases rather than files to represent their state. In databases, each row and each column may have a different policy, so enforcement at the file or block level is generally not appropriate. Table or column policies can be enforced with an appropriate file-based data model with Guardat in special cases. Nevertheless, we believe that Guardat can be generalized to fully support databases, but a design remains as future work.

6. Related work

Policy languages based on Datalog. Many declarative policy language are based on Datalog and resemble the Guardat policy language in syntax and semantics. Some examples are Soutei [42], Binder [14] and SecPAL [6]. Whereas these languages are generic, the Guardat policy language is domain-specific and contains custom-designed, storage-relevant predicates (Section 3.3). Soutei, Binder and SecPAL allow intensional (recursive, rule-defined) predicates, which the Guardat policy language omits to keep the implementation simple. These predicates can be added to Guardat without any conceptual challenges. DKAL [20] extends Datalog with declarative rules for exchanging authorization credentials in distributed systems. Such rules can be added to Guardat as well.

TCG storage work group specification. Although developed independently, the Guardat architecture bears some resemblance to storage work group standards of the trusted computing group (TCG) [55]. Similar to Guardat, the TCG standard prescribes session-based communication with storage devices and access control on all calls. 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 attestation of stored data. Implementations exist for a subset of the TCG specification [54], 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, integrity policies, or attestations, all of which Guardat does.

Trusted computing. Trusted computing (TC) relies on a trusted platform module (TPM) attached to a computer's motherboard to provide a hardware root-of-trust [40], while Guardat relies on a controller (GDC) attached to a storage device, enclosure or server. While TC provides remote attestation of the software executing on a computer, Guardat protects stored files and attests their state. TC provides sealed storage, where data is encrypted with a key stored in the TPM and released only when the computer runs a specific, trusted software configuration. Guardat instead enforces a declarative policy on all data accesses. Compared to TC, Guardat can reduce the size of the TCB and its attack surface. Depending on the policy, the TCB may be as small as the GDC. TC can complement Guardat: A Guardat policy for file access can require that trusted software, verified via TC remote attestation, execute on the client computer. Conversely, TC can be used to attest the GDC.

Related trusted computing proposals. Building on TC, semantic attestation [21] enforces properties of a computation by a runtime verification substrate within a VMM. Guardat provides a limited form of semantic attestation that enforces a data access policy, and does not require machine virtualization. With Excalibur [45] data can be bound cryptographically to a predicate on nodes (e.g., "this node is in Europe" or "this node is running Xen"). Guardat can implement a similar capability with the help of a trusted authority to certify the predicate. However, Guardat can enforce many other policies directly, without requiring an external trusted authority. Pasture [29] is a TPM-backed messaging and logging library that enforces MAL on data stored on an untrusted client machine. Furthermore, clients can delete unaccessed data in a way that provably prevents future access. In Section 2.1, we describe a similar MAL policy in Guardat. Provable deletion can be added, as described in our TR [58].

VMM/OS data protection. Overshadow [12] uses VMM interception of application-to-kernel switches to protect confidentiality and integrity of in-memory application data from a corrupted OS. Using memory-mapped files, the same protection extends to persistent files. Guardat enforces declarative policies (not considered in Overshadow) on persistent files. Overshadow's in-memory protection can be combined with Guardat for end-to-end enforcement of policies on data flowing through a system.

In InkTag [24], designated high-assurance processes (HAPs) are protected from the OS by the VMM, which verifies the OS's actions. The VMM also intercepts all I/O and enforces access control list-based protection on file accesses. Guardat supports richer policies. Protections provided by InkTag can be circumvented by rebooting into an OS without InkTag. Guardat protections cannot be bypassed by rebooting. InkTag requires changes to the OS and, depending on the application workload, may add 2-3x overhead. Guardat does not require any changes to the OS and incurs only moderate overheads even for very challenging workloads. Furthermore, Guardat provides policy protection even for remote clients, which Inktag does not.

The Nexus operating system [51], like the earlier Taos operating system [66], applies policy-based authorization on OS interfaces for file access, memory mapping, IPC and process management. The Nexus policy language, NAL, is similar to Guardat's [46]. Like Guardat, the untrusted application demonstrates policy compliance by providing credentials ahead of access. However, Guardat focuses exclusively on the storage subsystem and its policy language is more expressive for this subdomain, e.g., it can express the MAL policy, which NAL cannot. Moreover, Guardat is implemented in the storage layer. Nexus optionally provides data integrity by maintaining a Merkle hash tree over the entire filesystem and storing the root hash in a TPM. The same idea may be applied to Guardat.

DCAC [68] modifies the OS kernel to enforce attribute-based access control on files. In DCAC, processes have attributes (privileges) and file policies are conjunctions and disjunctions of these attributes. A process may create sub-attributes of any attribute it controls, and it may delegate these sub-attributes to other processes. DCAC can be used to build security primitives like process sandboxing and application-controlled ad hoc sharing. The same primitives can be built on Guardat, using application-created private keys instead of attributes for authorization. Additionally, Guardat can enforce data integrity, access logging and time-dependent access policies that DCAC cannot.

Protected storage. Butler et al. [9–11] 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. Guardat can also express such policies.

Commercially available self-encrypting disks [48] encrypt data to ensure its confidentiality when the device is lost or stolen. Our Guardat prototype includes this capability. 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 file 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) [2, 15, 18], access requests include a cryptographic capability 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 many policies without relying on an external policy manager; thus, Guardat can operate in an otherwise offline environment (unless a policy specifically delegates to an external verifier). Guardat can enforce content-based policies and attest files, which capability-based NAS cannot.

Type-safe disks (TSD) [52] 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 additionally supports cryptography and secure channels, which provide stronger protection against compromised hosts, buggy filesystems and operator mistakes. Also, Guardat's policy language can support rich policies beyond filesystem metadata integrity.

Storage systems such as Self Securing Storage (S4) [56] and NetApp's SnapVault [22] 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 2.1), as well as confidentiality and access accounting.

Protected filesystems. jVPFS [61, 62] 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 instead operates at the storage layer, and supports a much wider range of confidentiality and integrity policies.

SQCK [19] states a filesystem's metadatay invariants as SQL queries, and checks/repairs these invariants off-line. Recon [16] enforces declarative invariants on a filesystem's metadata at runtime. Guardat instead enforces *data* confidentiality and integrity, and does not rely on correct filesystem metadata.

PCFS [17] and PFS [59] enforce declarative integrity and confidentiality policies at the filesystem-level. Unlike Guardat, PCFS and PFS cannot enforce policies that depend on the content or size of files, do not attest stored files, and can be bypassed by booting into a different configuration. PFS uses the NAL policy language, which we discussed earlier. PCFS uses a formal logic with more connectives than the Guardat policy language. However, the logic is undecidable, which increases the clients' work in establishing policy compliance.

Extended storage functionality. Commercial hybrid disks [50] 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 storage with an object-based interface [35]. These systems offer capability-based security for whole objects, which we already compared to. Some Guardat commands are also object-based, and could therefore be integrated with an object-based storage standard. Seagate's recent Kinetic Open Storage Platform [47] is based on storage devices with Ethernet interfaces and in-built key-value stores and secure data migration abilities (similar to Guardat pickle/unpickle commands). Unlike Guardat, access control relies on a trusted library outside the drive. Several storage subsystems like active disks [43], semantically smart disks [53] and differentiated storage services [34] include program logic to improve performance. Guardat addresses the orthogonal concerns of data confidentiality, integrity and access accounting.

Pennington et al. [41] 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 the policies can be richer than the rules of an IDS system.

7. Conclusion

To the best of our knowledge, Guardat is the first system that enforces, at the storage layer, rich per-file confidentiality, integrity and access accounting policies, and attests the state of files. Enforcement at the storage layer reduces the risk of policy circumvention due to software bugs, misconfigurations and operator error. The Guardat policy language, although based on well-understood foundations, provides domain-specific predicates to enforce rich confidentiality, integrity and access accounting policies based on a wide range of conditions, including client authentication, trusted wall-clock time, and the state (content) of files, even at sub-file granularity. Guardat ensures the confidentiality and integrity of a system's persistent state and data, yet is easy to deploy and amenable to an efficient implementation, as demonstrated by an experimental evaluation.

Acknowledgments

We would like to thank our shepherd Michael Swift and the anonymous reviewers for their helpful feedback. Lorenzo Alvisi, Bobby Bhattacharjee, and Ted Wobber had provided valuable suggestions based on earlier drafts. This work was supported by the Max Planck Society, the European Research Council (ERC imPACT SyG and ERC Starting Grant No. 307732), and a Google Faculty Research Award to Peter Druschel and Deepak Garg.

References

[1] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *Trans. Storage 3*, 3 (2007).

[2] AGUILERA, M. K., JI, M., LILLIBRIDGE, M., MACCORMICK, J., OERTLI, E., ANDERSEN, D. G., BURROWS, M., MANN, T., AND THEKKATH, C. Block-level security for network-attached disks. In *Proc. of the 2nd USENIX FAST* (2003).

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

[4] APPLE INC. Fusion Drive. `https://support.apple.com/en-us/HT202574`.

[5] ARM. ARM Security Technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, ARM Technical White Paper, 2009.

[6] BECKER, M. Y., FOURNET, C., AND GORDON, A. D. Design and semantics of a decentralized authorization language. In *Proc. of the 20th IEEE Computer Security Foundations Symposium* (2007).

[7] BLAZE, M., FIEGENBAUM, J., AND IOANNIDIS, J. The Keynote trust-management system version 2. See `http://www.ietf.org/rfc/rfc2704.txt`, 1999.

[8] BTRFS. Btrfs. `https://btrfs.wiki.kernel.org/index.php/Main_Page`, 2014.

[9] Butler, K., McLaughlin, S., Moyer, T., and McDaniel, P. New security architectures based on emerging disk functionality. IEEE Computer Society.

[10] Butler, K. R. B., McLaughlin, S. E., and McDaniel, P. D. Rootkit-resistant disks. In *Proc. of the ACM CCS* (2008).

[11] Butler, K. R. B., McLaughlin, S. E., and McDaniel, P. D. Kells: a protection framework for portable data. In *Proc. of the ACSAC* (2010).

[12] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th ACM ASPLOS* (2008).

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

[14] DeTreville, J. Binder, a logic-based security language. In *Proc. of the IEEE S&P* (2002).

[15] Factor, M., Naor, D., Rom, E., Satran, J., and Tal, S. Capability based secure access control to networked storage devices. In *Proc. of the 24th IEEE MSST* (2007).

[16] Fryer, D., Sun, K., Mahmood, R., Cheng, T., Benjamin, S., Goel, A., and Brown, A. D. Recon: Verifying file system consistency at runtime. *Trans. Storage 8*, 4 (Dec. 2012).

[17] Garg, D., and Pfenning, F. A proof-carrying file system. In *Proc. of the 31st IEEE S&P* (2010).

[18] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A cost-effective, high-bandwidth storage architecture. In *Proc. of the 8th ACM ASPLOS* (1998).

[19] Gunawi, H. S., Rajimwale, A., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Sqck: A declarative file system checker. In *Proc. of the 8th USENIX OSDI* (2008).

[20] Gurevich, Y., and Neeman, I. Dkal: Distributed-knowledge authorization language. In *Proc. of the 21st IEEE Computer Security Foundations Symposium* (2008).

[21] Haldar, V., Chandra, D., and Franz, M. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *Proc. of the 3rd USENIX Virtual Machine Research And Technology Symposium* (2004).

[22] Hayakawa, M. WORM Storage on Magnetic Disks Using SnapLock Compliance and SnapLock Enterprise. Tech. Rep. TR-3263, Network Appliance, 2007.

[23] Hitz, D., Lau, J., and Malcolm, M. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter Technical Conference* (1994).

[24] Hofmann, O. S., Kim, S., Dunn, A. M., Lee, M. Z., and Witchel, E. Inktag: Secure applications on an untrusted operating system. In *Proc. of ACM ASPLOS* (2013).

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

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

[27] Intel Corp. Software Guard Extension Programming Reference. `http://software.intel.com/sites/default/files/329298-001.pdf`, 2012.

[28] Jim, T. SD3: A trust management system with certified evaluation. In *Proc. of the IEEE S&P* (2001).

[29] Kotla, R., Rodeheffer, T., Roy, I., Stuedi, P., and Wester, B. Pasture: Secure offline data access using commodity trusted hardware. In *Proc. of the 10th USENIX OSDI* (2012).

[30] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information flow control for standard OS abstractions. In *Proc. of 21st ACM SOSP* (2007).

[31] Li, N., and Mitchell, J. C. Datalog with constraints: A foundation for trust management languages. In *Proc. of the 5th Symposium on Practical Aspects of Declarative Languages* (2003).

[32] Loo, B. T. *The Design and Implementation of Declarative Networks*. PhD thesis, University of California, Berkeley, 2006.

[33] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing declarative overlays. In *Proc. of the 20th ACM SOSP* (2005).

[34] Mesnier, M., Chen, F., Luo, T., and Akers, J. B. Differentiated storage services. In *Proc. of the 23rd ACM SOSP* (2011).

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

[36] OASIS. eXtensible Access Control Markup Language (XACML). Online at `http://www.oasis-open.org/committees/xacml`.

[37] OCZ Technology Inc. Deneva 2 data sheet. `http://ocz.com/enterprise/download/product-briefs/deneva2_cs_slc_product_brief.pdf`, 2011.

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

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

[40] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping trust in commodity computers. In *Proc. of the 31st IEEE S&P* (2010).

[41] Pennington, A. G., Griffin, J. L., Bucy, J. S., Strunk, J. D., and Ganger, G. R. Storage-based intrusion detection. *ACM Trans. Inf. Syst. Secur. 13*, 4 (Dec. 2010).

[42] Pimlott, A., and Kiselyov, O. Soutei, a logic-based trust-management system. In *Proc. of the 8th International*

Symposium on Functional and Logic Programming (FLOPS) (2006).

[43] RIEDEL, E., FALOUTSOS, C., GIBSON, G., AND NAGLE, D. Active disks for large-scale data processing. *IEEE Computer 34*, 6 (2001).

[44] SAMSUNG. 830 SSD data sheet. `http://www.samsung.com/us/system/consumer/product/mz/7p/c1/mz7pc128nam/830.pdf`, 2011.

[45] SANTOS, N., RODRIGUES, R., GUMMADI, K. P., AND SAROIU, S. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proc. of the 21st USENIX Security Symposium* (2012).

[46] SCHNEIDER, F. B., WALSH, K., AND SIRER, E. G. Nexus authorization logic (nal): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.* (June 2011).

[47] SEAGATE TECHNOLOGY LLC. Kinetic Open Storage Platform. `http://www.seagate.com/solutions/cloud/data-center-cloud/platforms`.

[48] SEAGATE TECHNOLOGY LLC. Self-encrypting hard disk drives in the data center. Tech. Rep. TP583, 2007.

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

[50] SEAGATE TECHNOLOGY LLC. Momentus XT Data Sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_momentus_xt.pdf`, 2012.

[51] SIRER, E. G., DE BRUIJN, W., REYNOLDS, P., SHIEH, A., WALSH, K., WILLIAMS, D., AND SCHNEIDER, F. B. Logical attestation: An authorization architecture for trustworthy computing. In *Proc. of 23rd ACM SOSP* (2011).

[52] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-safe disks. In *Proc. of the 7th USENIX OSDI* (2006).

[53] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *Proc. of the 2nd USENIX FAST* (2003).

[54] 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.

[55] STORAGE WORK GROUP OF THE TRUSTED COMPUTING GROUP. TCG storage architecture core specification. `http://www.trustedcomputinggroup.org/resources/tcg_storage_architecture_core_specification`, 2011.

[56] STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A. N., AND GANGER, G. R. Self-securing storage: Protecting data in compromised systems. In *Proc. of the 4th USENIX OSDI* (2000).

[57] THE ISCSI ENTERPRISE TARGET PROJECT. `http://iscsitarget.sourceforge.net/`, 2011.

[58] VAHLDIEK-OBERWAGNER, A., ELNIKETY, E., METHA, A., GARG, D., DRUSCHEL, P., RODRIGUES, R., GEHRKE, J., AND POST, A. Guardat: Enforcing data policies at the storage layer. Tech. Rep. 002, MPI-SWS, 2014. `http://www.mpi-sws.org/cont/tr/2014-002.pdf`.

[59] WALSH, K., AND SCHNEIDER, F. B. Costs of security in the PFS file system. Tech. rep., Computing and Information Science, Cornell University, 2012.

[60] WARREN, D. H. D. An abstract Prolog instruction set. Tech. Rep. Technical Note 309, SRI International, 1983.

[61] WEINHOLD, C., AND HÄRTIG, H. VPFS: Building a virtual private file system with a small trusted computing base. In *Proc. of the 3rd ACM EuroSys* (2008).

[62] WEINHOLD, C., AND HÄRTIG, H. jVPFS: Adding robustness to a secure stacked file system with untrusted local storage components. In *Proc. of the USENIX ATC* (2011).

[63] WIKIMEDIA FOUNDATION. Image Dump. `http://archive.org/details/wikimedia-image-dump-2005-11`, 2005.

[64] WIKIMEDIA FOUNDATION. Static HTML dump. `http://dumps.wikimedia.org/`, 2008.

[65] WIKIMEDIA FOUNDATION. Page view statistics April 2012. `http://dumps.wikimedia.org/other/pagecounts-raw/2012/2012-04/`, 2012.

[66] WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. Authentication in the Taos operating system. *ACM Trans. Comput. Syst. 12*, 1 (1994).

[67] WOBBER, T., YUMEREFENDI, A., ABADI, M., BIRRELL, A., AND SIMON, D. R. Authorizing applications in singularity. In *Proc. of the 2nd ACM EuroSys* (2007).

[68] XU, Y., DUNN, A. M., HOFMANN, O. S., LEE, M. Z., MEHDI, S. A., AND WITCHEL, E. Application-defined decentralized access control. In *Proc. of USENIX ATC* (2014).

[69] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in HiStar. In *Proc. of 7th USENIX OSDI* (2006).