Techniques to Protect Confidentiality and Integrity of Persistent and In-Memory Data

A dissertation submitted towards the degree
Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Anjo Lucas Vahldiek-Oberwagner

Saarbrücken
October, 2018

Date of Colloquium:	February 5^{th}, 2019
Dean of Faculty:	Prof. Dr. Sebastian Hack
Chair of the Committee:	Prof. Dr. Gert Smolka
Reporters	
First Reviewer:	Prof. Peter Druschel, Ph.D.
Second Reviewer:	Deepak Garg, Ph.D.
Third Reviewer:	Stefan Saroiu, Ph.D.
Academic Assistant:	Engel Lefaucheux, Ph.D.

Abstract

Today computers store and analyze valuable and sensitive data. As a result we need to protect this data against *confidentiality* and *integrity* violations that can result in the illicit release, loss, or modification of a user's and an organization's sensitive data such as personal media content or client records. Existing techniques protecting confidentiality and integrity lack either efficiency or are vulnerable to malicious attacks. In this thesis we suggest techniques, Guardat and ERIM, to efficiently and robustly protect *persistent* and *in-memory* data.

To protect the confidentiality and integrity of *persistent* data, clients specify per-file policies to Guardat declaratively, concisely and separately from code. Guardat enforces policies by mediating I/O in the *storage layer*. In contrast to prior techniques, we protect against accidental or malicious circumvention of higher software layers. We present the design and prototype implementation, and demonstrate that Guardat efficiently enforces example policies in a web server.

To protect the confidentiality and integrity of *in-memory* data, ERIM isolates sensitive data using Intel Memory Protection Keys (MPK), a recent x86 extension to partition the address space. However, MPK does not protect against malicious attacks by itself. We prevent malicious attacks by combining MPK with call gates to trusted entry points and ahead-of-time binary inspection. In contrast to existing techniques, ERIM *efficiently* protects frequently-used session keys of web servers, an in-memory reference monitor's private state, and managed runtimes from native libraries. These use cases result in high switch rates of the order of 10^5–10^6 switches/s. Our experiments demonstrate less then 1% runtime overhead per 100,000 switches/s, thus outperforming existing techniques.

Kurzdarstellung

Computer speichern und analysieren wertvolle und sensitive Daten. Das hat zur Folge, dass wir diese Daten gegen *Vertraulichkeits-* und *Integritätsverletzungen* schützen müssen. Andernfalls droht die unerlaubte Freigabe, der Verlust oder die Modifikation der Daten. Existierende Methoden schützen die Vertraulichkeit und Integrität unzureichend, da sie ineffizient und anfällig für mutwillige Angriffe sind. In dieser Doktorarbeit stellen wir zwei Methoden, Guardat und ERIM, vor, die *persistente* Daten und Daten *im Arbeitsspeicher* effizient und widerstandsfähig beschützen.

Um die Vertraulichkeit und Integrität *persistenter* Daten zu schützen, verknüpfen Nutzer für jede Datei Richtlinien in Guardat. Guardat überprüft diese Richtlinien für jeden Zugriff und setzt diese im Speichermedium durch. Im Gegensatz zu existierenden Methoden, beschützt Guardat vor mutwilligem Umgehen. Wir beschreiben die Methode, eine Implementierung und evaluieren die Effizienz von Beispielrichtlinien.

Um die Vertraulichkeit und Integrität von Daten *im Arbeitsspeicher* zu schützen, isoliert ERIM sensitive Daten mit Hilfe von Intel Memory Protection Keys (MPK), eine neue x86 Erweiterung, um den Arbeitsspeicher aufzuteilen. Da MPK allerdings nicht gegen mutwillige Angriffe schützt, verhindert ERIM diese, indem es MPK mit widerstandsfähigen Wechseln der Speicherbereiche und einer Binärcodeüberprüfung kombiniert. Im Gegensatz zu existierenden Methoden, beschützt ERIM effizient häufig genutzte Sitzungsschlüssel, Zustandsvariablen eines Referenzmonitors und verwaltete Laufzeitumgebungen von nativen Bibliotheken. Unsere Experimente zeigen, dass weniger als 1% Laufzeitmehraufwand je 100.000 Wechseloperationen pro Sekunde notwendig sind.

Publications

Parts of this thesis have appeared in the following publications.

- "Guardat: Enforcing data policies at the storage layer". Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Ansley Post, Rodrigo Rodrigues, Johannes Gehrke. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

- "ERIM: Secure and Efficient In-process Isolation", Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Deepak Garg, Peter Druschel. Under review and technical report (arXiv:1801.06822), 2018.

Additional publications not included in this thesis.

- "Protecting Data Integrity with Storage Leases", Anjo Vahldiek, Eslam Elnikety, Ansley Post, Peter Druschel, Rodrigo Rodrigues. MPI-SWS Technical Report 2011-008.

- "Thoth: Comprehensive Policy Compliance in Data Retrieval Systems", Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, Peter Druschel. In *Proceedings of the USENIX Security Symposium*, 2016.

- "Light-Weight Contexts: An OS Abstraction for Safety and Performance", James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, Peter Druschel. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

- "Pesos: Policy Enhanced Secure Object Store", Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, Christof Fetzer. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018.

Fynn, Julius, Timon and Kerstin.

Acknowledgments

I would like to extend my thanks to many people. They so generously contributed to the work presented in this thesis and helped me during the ups and downs of graduate life.

Firstly, I would like to express my sincere gratitude to my advisers Peter Druschel and Deepak Garg for their continuous support. Their guidance helped me in research and writing of this thesis. I'm very grateful to them for giving me the time to raise my family and be with my children.

I would like to thank the rest of my thesis committee: Paul Francis, and Stefan Saroiu, for their insightful comments and encouragement, but also for the hard questions which helped me to further improve my research from various perspectives.

I have been very fortunate to collaborate with fantastic fellow PhD students, Eslam Elnikety and Aastha Metha. With them, the countless and tiring deadlines turned into adventure trips. I cannot imagine going through graduate life without their friendship and support.

Besides Eslam and Aastha, I had the great pleasure to collaborate with Bobby Bhattacharjee, Nuno Duarte, Johannes Gehrke, James Litton, Rodrigo Rodrigues, and Ansley Post.

Furthermore, I would like to thank fellow students and postdocs at MPI-SWS for creating a great place to work, in particular Arpan, Bimal, Bilal, Cheng, Ekin, Ezgi, Felipe, Filip, Georg, Jan-Oliver, James, Juhi, Manohar, Natacha, Nancy, Nuno, Oana, Paarijaat, Pedro, Pramod, Reinhard, Scott, and Viktor.

A special thanks to the staff at MPI-SWS, Mary-Lou, Claudia, Brigitta, and Annika, and in particular to Carina and Chris for their fantastic efforts and technical know-how as well as Rose for her guidance on scientific writing and presenting.

I would like to thank my family for their support and strengthening words. Without them I could not have accomplished this work. Fynn, Julius and Timon for their enthusiasm and belief that every problem has a simple solution. Finally, I am deeply thankful to Kerstin for all her love, support, her ultimate support during this endeavor and three awesome kids!

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Today computers assist people in most daily activities such as social interactions, learning, and information sharing. People entrust computer systems with their valuable data, e.g., personal media content, financial and health records, and cryptographic keys. Computer systems ought to protect the confidentiality and integrity of such data. Confidentiality guarantees that only authorized reads of the data succeed. Integrity prevents unauthorized updates to the data.

Violating the confidentiality and integrity of sensitive data can result in its leak, loss or modification. As a result clients and organizations may face the loss of highly sentimental data and reputational or financial loss. Common causes of data confidentiality and integrity violations [15] include software bugs, security vulnerabilities, misconfiguration and operator error. First, a bug, e.g., in an application may overwrite existing files, violating integrity. Second, security vulnerabilities in online services may be used by malicious attackers to extract sensitive data such as cryptographic keys, violating confidentiality. Third, misconfigurations may lead to accidental data reads, violating confidentiality. Fourth, an administrative operator of a system may accidentally delete data from the system, violating integrity.

To prevent these violations, many techniques to protect data confidentiality and integrity have been proposed. Model checking, language-based static analysis, testing and reference monitors are broad classes of such techniques. Model checking ensures that an application follows a given specification, and thus provides the

strongest guarantees compared to the remaining guarantees. However, model checking of everyday software (e.g., operating systems or web browsers) is complex and consequently difficult and expensive, which limits the use of model checking to specific components in high-risk applications. Language-based static analysis enforces specific program invariants over source code and marks invariant violations such as bugs and vulnerabilities during development. Due to the approximation of runtime values, analysis tools suffer in practice from high false positives rates. In addition, limited support for multi-language software systems hinders their broad adoption [104]. Testing, on the other hand, provides a reasonable and best-effort coverage of violations over a subset of application inputs. Although widely used, testing-based approaches do not provide formal guarantees, since testing every possible input is usually unfeasible, especially when considering malicious attacks. Thus, none of these techniques provides the ability to enforce confidentiality and integrity systematically across applications and independent of the application implementation.

In contrast, reference monitors [6] enforce confidentiality and integrity of data by observing applications at runtime, mediating relevant events (such as I/O or memory accesses) and denying accesses which violate confidentiality or integrity. Treating the application as a black box allows reference monitors to enforce confidentiality and integrity independent of the application implementation and application size, and systematically across applications. This allows adoption across a wide range of use cases including legacy applications with no access to source code. Compared to model checking or static analysis, which prove correctness of an application, reference monitors reduce the proof of correctness to a smaller and simpler piece of code, namely the reference monitor. In contrast to the other techniques, reference monitors induce runtime overhead on the production systems to which they are applied. Reducing this overhead is an important design consideration.

Reference monitors have been applied to various use cases. When protecting data confidentiality and integrity, we differentiate between protecting *in-transit*, *persistent*, or *in-memory* data.*In-transit* data is typically protected using cryptographic methods which encrypt the contents providing confidentiality, and sign the contents providing integrity. For secure communication today's computer systems rely on the SSL/TLS standard with readily available implementations in several cryptographic libraries such as OpenSSL [96]. However, for both, *persistent* and *in-memory* data, existing techniques do not efficiently and comprehensively protect data confidentiality and integrity [134, 45, 110, 124, 48, 3, 74, 59, 29, 68, 75, 113, 58, 38, 29].

To protect the confidentiality and integrity of *persistent* data, existing monitoring techniques [48, 134, 45, 118, 139, 19, 74, 5, 124, 110] mediate application I/O by in a library, file system, operating system, hypervisor, or in the storage layer. However, mediation in a layer other than the storage layer can be easily bypassed, and none of the storage layer techniques support general confidentiality and integrity policies. Hence, a strong and general policy enforcement technique for persistent data is currently missing.

To protect the confidentiality and integrity of *in-memory* data from accesses by an untrusted application, prior work relies on memory isolation through language and runtime [38, 72, 129, 143, 68], process-based [74, 59, 29, 20], or randomization-based techniques [113, 58]. First, language and runtime techniques isolate by inserting checks into the application binary to protect against arbitrary data accesses. Although robust against malicious attacks, these techniques suffer from runtime overheads to perform the checks. Second, process-based isolation splits the execution of an application into separate hardware-protected processes. Similar to language and runtime isolation, process-based isolation is robust against malicious attacks. The efficiency depends on the cost of context switches between application processes, which is usually high. Third, randomization-based isolation uses the huge address space to hide sensitive data at a random location. While randomization-based

techniques are efficient (no checks or high switch costs), malicious attackers can find the secret location and break guarantees [113, 60, 39, 49, 94]. No existing technique efficiently isolates memory with low runtime overhead, and offers strong protection against malicious attackers. As a result, no existing isolation technique is sufficient for several important use cases such as protecting cryptographic keys or native libraries in managed runtimes.

Contributions: This dissertation contributes Guardat, which efficiently enforces *expressive confidentiality and integrity policies* at the storage layer protecting persistent data, and ERIM, which *strongly and efficiently* isolates in-memory data.

Guardat: In contrast to previous techniques [48, 134, 45, 118, 139, 19, 74] that intercept at the application or a system software layer, Guardat protects the confidentiality and integrity of *persistent* data at the storage layer. Thus, Guardat minimizes the size and attack surface of the trusted computing base (TCB) relied upon for enforcement.

Protecting at the storage layer limits available access information to block addresses. As a result, existing storage layer techniques techniques [5, 124, 110] do not enforce generic confidentiality and integrity policies. To overcome the lack of client information such as file names and access credentials, clients communicate with Guardat through secure channels, tunneling through untrusted system layers like the operating system. Clients use this communication to send additional information such as file names or access credentials. Using this information, Guardat enforces confidentiality and integrity for every data access while relying only on its own enforcement logic.

With Guardat, confidentiality and integrity requirements are specified as per-file policies by users, developers, or administrators. Policies specify the conditions under which a file may be read, updated, or have its policy changed. These conditions, written in a declarative language, may depend on client authentication, the initial

and final states of the file (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 file complies with the policy.

For example, users can rely on Guardat to mitigate serious threats: To prevent the insertion of malicious code in executables, a policy can protect executable files by allowing only updates signed by a trusted party; to prevent system logs from corruption and tampering, a policy can protect log files by making them append-only; to prevent the illicit release of a user's private data, a policy can protect the user's data by requiring an authenticated secure session to read data; to prevent arbitrary file accesses and allow auditing of accesses, a policy can protect files by requiring a mandatory log entry before accessing a file.

We evaluate the efficiency of Guardat using the policy examples described above. We show that Guardat enforces confidentiality and integrity policies with low overhead. When protecting a web server's content from accesses by unauthorized users, and binaries from unauthorized updates, the throughput overhead is less than 1% compared to no protection.

ERIM: ERIM is a framework for strong, efficient isolation of in-memory data. It allows partitioning an application into a trusted and an untrusted component within a single address space. For this, ERIM relies on Memory Protection Keys (MPK) [64], a recent x86 extension to partition the address space into up to 16 disjoint memory domains. With ERIM the trusted and the untrusted component's data reside in different domains and ERIM controls access to each domain. A new *user-mode* CPU instruction (WRPKRU) switches access permissions to domains efficiently (about 60 cycles per switch), without kernel intervention. Although efficient, this instructions allows malicious attackers to escalate their access permissions. Hence, by itself MPK

is not sufficient to guarantee security against malicious or compromised untrusted components.

ERIM's contribution is to build secure memory isolation using MPK by (1) providing call gates to securely transfer control to the trusted component at predefined entry points without kernel intervention, and (2) use binary inspection to remove exploitable binary code ensuring that the switch instruction cannot be exploited. As a result, to gain access to secret data, an untrusted component has to invoke a call gate transferring control to the trusted component. In contrast to prior techniques, ERIM's memory isolation significantly reduces the switch cost between the trusted and untrusted component, does not slow down untrusted component like language-based techniques, and protects against malicious attackers.

We apply ERIM's design to challenging and previously high-overhead use cases [74, 29, 72]. First, we isolate frequently used OpenSSL session keys of a web server (nginx) and show scalability. Second, we isolate the safe region in an implementation of code-pointer integrity (CPI) [72]. Third, we isolate a managed runtime (node.js) from an untrusted native library (SQLite). Our results show that ERIM provides robust memory isolation with a low overhead of less than 1% for 100,000 switches per second.

Overview: In the remainder of this thesis we further describe the background and related work (Chapter 2), followed by detailed description of the design, implementation, and evaluation of Guardat (Chapter 3) and ERIM (Chapter 4). Finally, we conclude and describe future work (Chapter 5).

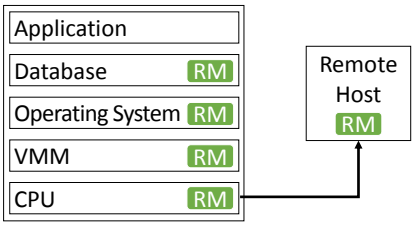
CHAPTER 2

Background

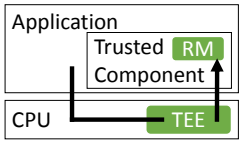
In this chapter we provide an overview of the background work on reference monitors and briefly describe existing techniques to protect confidentiality and integrity of *persistent* and *in-memory* data. This chapter is only meant to server as a background material for understanding the thesis. A detailed comparison to existing work is provided in Sections 3.8 and 4.6.

Reference monitoring enforces security policies at run time without insisting that the application be bug-free. Reference monitors intercept all relevant operations, evaluate each operation against the required policy and deny operations when violations are imminent. We summarize state-of-the-art techniques for reference monitoring.

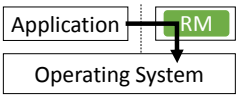
Figure 2.1 depicts possible implementation scenarios for reference monitors. Reference monitors have been implemented at different abstraction layers within the software and hardware stack (see Figure 2.1a). Each abstraction layer guards access to the resources provided to higher layers. Reference monitors in higher layers (e.g., application, database or file system) rely on protection guarantees provided by lower layers, increasing the TCB and risk of circumvention of the reference monitor. While monitoring at an abstraction layer, reference monitors can be implemented by isolating software components in trusted execution environments (TEE) [82] (see Figure 2.1b) or a separate application process [22] (see Figure 2.1c), by sandboxing



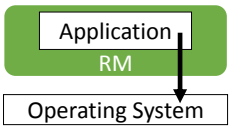
(a) Reference monitor at each level of abstraction



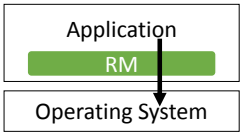
(b) Reference monitor in trusted execution environment (TEE)



(c) Reference monitor in separate application



(d) Sandboxing application inside a reference monitor



(e) Inlined reference monitor

Figure 2.1: Reference monitor implementation scenarios

an application [143] (see Figure 2.1d), or by inlining monitors into the application itself [1, 72, 29] (see Figure 2.1e).

Enforcement techniques in non-application layers efficiently mediate all accesses to relevant resources (e.g., memory or files). Usually lower abstraction layers, such as the operating system (OS) or virtual machine monitor (VMM), intercept events with coarse-grain information from the application. Each layer abstracts information with help from the application. For instance, implementing a per file confidentiality policy is only possible within the file system layer or above. At these layers the accessed file and its associated policy is still available.

Monitoring at the application layer offers the most detailed information about the application state and execution at the cost of a larger TCB and risk of circumvention. Inlining the mediation and enforcement into the application [37, 72] offers the ability to protect the integrity of the control flow of an application at the cost of additional checks for every indirect jump and return. Enforcing such application level guarantees

at a lower layer (e.g., the OS) is infeasible, since every check would incur high switch costs between the layer and the application.

While numerous reference monitoring techniques have been suggested, this dissertation focuses on protecting the confidentiality and integrity of *persistent* and *in-memory* data. We describe next the state of the art in protecting persistent and in-memory data.

2.1 Protecting persistent data

In the following we describe techniques to protect persistent data from illicit release, corruption or deletion due to bugs, misconfigurations, operator error or malicious attacks. We do not consider hardware failures, since replication (such as RAID [98]) or data encryption mitigate these threats easily.

In general, the data confidentiality and integrity guarantees in today's computer systems depend on, and are spread across the application, a database management system, the OS (including the file system) and virtual machine monitors. For example, each layer enforces its own user access control protecting against illicit accesses and in some cases also keeps data hashes to protect the integrity. Compared to application layer protection, lower layers provide a stronger protection against circumvention, but typically do not provide a generic policy enforcement and instead focus on specific uses and policies.

Hypervisor/OS data protection Nexus [118] and TAOS [139] are two OS-level techniques that enforce authorization policies on OS interfaces (e.g., files, inter-process communication, memory mappings or process management) protecting data confidentiality. Nexus optionally maintains a Merkle hash tree of the file system to provide data integrity. In contrast to Nexus and TAOS which enforce policies, Dune [19] and lwC [74] are frameworks to build a reference monitor at the OS abstraction layer mediating the system call interface and both show use cases to

protect data confidentiality and integrity. Although these systems improve the confidentiality and integrity protection of persistent data, they can be circumvented by accessing the data directly at the storage layer.

Protected file systems Beyond the access control in existing commercial file systems, jVPFS [134] combines a small, trusted file system with a conventional untrusted file system to ensure data confidentiality and integrity. It furthermore encrypts data, maintains hash trees and logs data accesses. PCFS [45] enforces declarative confidentiality and integrity policies. These systems suffer from possible circumvention by directly accessing the data at the storage layer.

Protected storage Self-encrypting disks [110], dm-crypt [32] and Bitlocker [87] encrypt data on disk, protecting its confidentiality and integrity. Web storage services [5] enforce access control on client data using user identities, groups and roles. In addition, data is encrypted for secure storage and confidential transit within the system. Storage systems like Self Securing Storage (S4) [124] maintain shadow copies of overwritten data allowing rollback in case of accidental or malicious corruption of data. These systems provide a specific guarantee instead of a system which enforces a large class of configurable policies.

Remote/Cloud storage In capability-based network attached storage [48, 40, 3], every request requires a capability from an external policy manager to allow the access. Separately creating these capabilities from the storage location limits the efficiency of enforceable policies. For example, content-dependent policies would increase the overheads to access the content at the policy manager and result in doubling the number of data accesses.

Due to the recent trend to offload data storage to an untrusted cloud storage service, several systems [128, 116, 79] propose additional techniques to protect data confidentiality and integrity. Shu et al. [116] introduce a trusted proxy server in between a client and a cloud storage service protecting data confidentiality and

integrity by authenticating clients and deploying an efficient Merkle hash tree. The recently suggested Pesos object store [70] builds on Guardat's policy language and interpreter to enforce confidentiality and integrity policies over objects stored in an untrusted cloud environment. It relies on trusted execution environments (TEE), namely Intel's SGX [63], and a storage device (Seagate's Kinetic disks) providing a secure channel to the TEE to ensure confidentiality and integrity of data stored in the cloud. Similarly to Guardat, Pesos associates policies with objects and intercepts every object access (mainly get/put requests) to evaluate the policy.

2.2 Protecting sensitive in-memory data

Enforcing security invariants while trusting only a small portion of an application's code generally requires isolating sensitive data so it cannot be leaked or corrupted by untrusted code, and facilitating switches to trusted code that has access to the isolated state. We discuss techniques to isolate and securely transfer control provided by different abstraction layers (operating systems, hypervisors, applications) and different techniques (compilers and language runtimes, and binary rewriting). Briefly, these techniques are either not highly efficient or do not offer strong protection. The techniques that offer strong protection either rely on costly context switches between user- and kernelspace or add runtime overhead on untrusted applications as checks are inlined into the binary code.

OS-based techniques A simple way to provide data isolation is to split application components into separate processes. This approach is feasible only if the rate of cross-component invocations is relatively low, so that the substantial inter-process communication and context switching overheads are tolerable.

Novel kernel abstractions like light-weight contexts (lwCs) [74] and secure memory views (SMVs) [59], combined with additional compiler support as in Shreds [29] or runtime analysis tools as in Wedge [20], have reduced the cost of data isolation

to the point where isolating OpenSSL long-term signing keys is feasible with little overhead [74].

Mimosa [50] relies on the Intel TSX hardware transactional memory to protect private cryptographic keys from software vulnerabilities and cold-boot attacks. Private keys are stored in memory in encrypted form. Accesses to the private key are performed within a transaction that first decrypts the private key using a register-based master key, performs the access, wipes the cleartext key, and then commits. The cleartext key never exists outside an uncommitted transaction and TSX ensures that an uncommitted transaction's data is never exposed to DRAM or other cores.

Virtualization-based techniques In-process data isolation can also be provided by a hypervisor. Dune [19] enables user-level processes to implement isolated compartments by leveraging the Intel VT-x x86 virtualization ISA extensions [64]. Koning et al. [68] sketch how to use the VT-x VMFUNC instruction to switch extended page tables in order to achieve in-process data isolation. SeCage [75] similarly relies on VMFUNC to switch between isolated compartments; it also provides static and dynamic program analysis based techniques to automatically partition monolithic software into compartments. TrustVisor [81] uses a thin hypervisor and nested page tables to support isolation. Similar to OS-based techniques, high switch costs between components allows an efficient adoption only in use cases with low switch rates.

Language and runtime techniques Data isolation can be provided as part of a memory-safe programming language with low overhead. The cost of this isolation is low if most of the checks can be done statically at compile time. However, only applications written in such languages benefit, the isolation depends on the compiler and runtime correctness, and can be undermined by libraries written in unsafe languages.

Software fault isolation (SFI) [129, 143] provides data isolation in unsafe languages through runtime memory access checks inserted by the compiler or by rewriting executable binaries. However, SFI imposes an overhead on all execution of untrusted code, not just on control transfers.

The memory-safe Rust language [103] allows unsafe code sections. Almohri et al. [4] split the address space of a Rust program to isolate the unsafe code from the remaining program.

Koning et al. [68] present a general isolation technique, called MemSentry, which instruments programs using an LLVM pass. The instrumentation can rely on several isolation techniques to ensure that only legitimate accesses to isolated data are allowed. One of the isolation techniques used by MemSentry is MPK, and experimental results show that this technique is the most efficient in situations where isolated data is located in a few contiguous regions and accesses are frequent. However, MemSentry requires a complementary control-flow integrity technique to prevent bypassing of the instrumentation. Similar to Memsentry, a recently suggested memory isolation, called IMIX, [43] also relies on a complementary control-flow integrity technique which adds substantial runtime overhead. In contrast, ERIM does not require control-flow integrity lowering the runtime overhead.

Shreds [29] provides a compiler and runtime library which protect the confidentiality and integrity of variables and code blocks. Developers annotate the source code to mark variables containing private data and code blocks computing on private data. The compiler inserts necessary runtime checks and initial isolation of variables and code blocks. The evaluation shows high switch cost of about 16ms which is only marginally faster than thread or process switches. This limits Shreds applicability to use cases with infrequent switching. We apply ERIM to more demanding use cases that switch more frequently and as a result, would have high overheads if isolated with Shreds.

Hardware-based trusted execution environments Intel's SGX [63] and ARM's TrustZone [12] ISA extensions allow (components of) applications to execute with hardware-enforced isolation that provides isolation even from the operating system. The overheads for switching to a secure component are similar to other efficient hardware-based isolation mechanisms [68].

ASLR Address space layout randomization (ASLR) is widely used to mitigate code reuse exploits such as buffer overflow attacks [113, 58]. By randomizing the layout of code and data in a process's address space, ASLR seeks to make it more difficult for attackers to reuse code as part of an exploit. In practice, ASLR has proved vulnerable to attacks that learn the location of code and data dynamically [113, 60, 39, 49, 94]. ASLR-Guard [78] seeks to increase the difficulty of such attacks by encrypting pointers to the ASLR region stored in memory.

Intel MPX, MPK, and ARM memory domain Intel introduced the MPX ISA extension in the Skylake CPU series [64]. MPX provides architectural support for bounds checking. A compiler can use up to four bounds registers, and each register can store a pair of 64-bit starting and ending addresses. Specialized instructions efficiently check a given address and raise an exception if the bounds are violated. By itself, MPX cannot enforce security invariants, but it can aid SFI-based techniques for data isolation [68].

Recent Intel CPUs provide support for memory protection keys (MPK) [64]. Protection keys provide a second level of page access permission controlled by the application itself. To access a page, both the OS page tables and an additional user-space controlled register must allow the access, else a page fault occurs. By itself, MPK is not suitable for security applications, because malicious code could raise its privileges. ERIM combines MPK with binary inspection to provide secure isolation.

ARM memory domains [11] are similar to Intel MPK, except that changing domain permissions requires a costly and privileged (kernel) operation. They cannot be used to implement isolation with low-cost switching like MPK.

CHAPTER 3

Guardat: Enforcing data policies at the storage layer

This chapter describes Guardat, a system to enforce confidentiality and integrity policies on *persistent* data. Briefly, the problem is that computer and storage systems increase in complexity and so does the risk to data confidentiality and integrity 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 the third party. Data stored in third-party platforms rely on the trust and reliability of the third-party provider. Today's systems enforce the applicable security policy of a file implicitly in their code. Furthermore, the policy specification and enforcement may spread over different subsystems, increasing the risk of circumvention and misconfiguration.

Guardat introduces a reference monitor at the storage layer to tackle these challenges. It provides a single-point of policy specification, configuration and enforcement at the storage layer relying only on its own policy interpreter, enforcement logic and explicit policy dependencies, thus minimizing the TCB and attack surface.

The following sections describe Guardat's design and API, its declarative policy language, example use cases, an implementation, related work and an experimental evaluation of a prototype implementation.

3.1 Design

Guardat's 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.

Enforcement at the storage layer is preferable, since it minimizes the risk of circumvention, and makes it easy to physically protect the trusted Guardat components in a machine room. A design enforcing at a higher layer (e.g., NAS file server, VMM or client OS layer) would extend trust to additional, and likely more distributed, components. Moreover, Guardat is able to bridge the semantic gap between files and blocks as without relying on the untrusted filesystem and its metadata.

Data stored in Guardat is organized into files. For each policy-protected file, Guardat maintains its own shadow metadata, consisting of an ordered list of extents, a unique numeric identifier, a textual name string (typically used to store the file's pathname(s)—multiple in the case of hard links), and a reference to a policy in effect for the file. The set of numeric identifiers form a flat namespace, while the set of names typically encode a conventional namespace hierarchy maintained by an untrusted filesystem. Each file can have its own policy but, typically, a collection of files share the same policy.

The policy of a file consists of four rules, one for each of the permissions **read**, **update**, **destroy** and **setpolicy**. Each rule specifies conditions on the context and environment under which the respective permission holds. 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 identifier (name) can be recycled; and the **setpolicy** rule describes when the policy can be changed. Storage commands that read or update a file or its metadata check conditions of the corresponding policy rules.

Guardat integrates with filesystems. The (untrusted) filesystem as usual assigns names and storage blocks to a file and translates file requests into block requests using its metadata. Guardat uses 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.

Guardat's program logic, called the Guardat controller or GDC, is integrated with a storage block device 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 sets of extents (files) using simple transactions, (b) associate policies with files, (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.

3.2 Threat model

The GDC, metadata and data must be physically protected from unauthorized access and undetected tampering. In our implementation (see Section 3.6.1), data and metadata storage devices are assumed to be physically protected, e.g., in a machine room with restricted access. Guardat policies are enforced, subject to external policy dependencies, regardless of bugs, misconfigurations, or security incidents outside the storage device, including incidents on any number of client machines.

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 can often 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. 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 Guardat devices with independent failure modes.

3.3 Interface

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. In the following, we describe the functionality provided by the interface. Table 3.1 shows all Guardat API calls.

3.3.1 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 (*sId*) that links later commands to the session. In the description of the remaining interface, we omit

Session API:

message, *sId* **handshake1**(message)	Initiates the session establishment.
int **handshake2**(*sId*, message)	Finalizes the session establishment.
int **endsession**(*sId*)	Terminates the session *sId*.

Transaction API:

tId **openTx**(*sId*, objname)	Starts a transaction on file named objname.
int **endTx**(*sId*, *tId*)	Commits a transaction.
int **setPolicy**(*sId*, *tId*, *pId*)	Set policy *pId* for objname.
int **reuse**(*sId*, *tId*, off, len, off')	Takes content of interval [off, off + len - 1] and inserts content at off'.
int **fresh**(*sId*, *tId*, b, len, buf, off [, cacheflag])	Write content to block and add to objname at offset off.
buf **readTx**(*sId*, *tId*, off, len [, cacheflag])	Reads from objname at offset off.

File/Policy API:

pId **createPolicy**(*sId*, policy)	Stores policy and returns a unique identifier *pId*.
int **destroy**(*sId*, objname)	Deletes objname's metadata and content.

Content Hashing API:

hId **initHash**(*tId*, curOrNew)	Creates hash identifier for current or new objname.
certificate **closeHash**(*hId*)	Computes hash, creates certificate and stores hash in cache.

Certificate API:

nonce **getNonce**(*sId*)	Returns pseudo-random nonce value.
int **setCertificate**(*sId*, certificate)	Provide certificate to Guardat.
certificate **attest**(*sId*, objname, nonce)	Creates a certificate attesting the state of objname.

Replication/Migration API:

buf **pickle**(*sId*, objname, targetGdKey)	Creates an encrypted buffer buf including the content and policy of file objname which can only be encrypted by a Guardat device with targetGdKey as public key.
int **unpickle**(*sId*, buf, objname)	Decrypts buf to extract content c and policy p; creates policy p and file named objname with content c; associates the previously created policy.

Table 3.1: Guardat Interface Calls

the *sId* argument as it appears in every call. Guardat can work with any client-side infrastructure for creating, managing and distributing public keys.

3.3.2 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 3.5. 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 filesystem designs with copy-on-write block allocation, e.g., in WAFL, ZFS, and Btrfs [56, 125, 23]. Outside a transaction, destructive writes succeed if allowed by the policy.

The transaction API adds 5 new commands: openTx, endTx, reuse, fresh and readTx, setPolicy. The call **openTx**(*sId*, objname) starts a new transaction on the file named objname. Generating objname is up to the (untrusted) higher layers, e.g., the filesystem. If objname does not exist, a new empty file is created and given this name (this is the only way to create a file in Guardat). The call returns a transaction id (*tId*) that links later calls to the transaction and the session. A file is updated by

reusing content from its current version and adding fresh content to create a new version. The call **reuse**(*tId*, 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**(*tId*, blk, len, buf, off) writes len bytes from buffer buf to the extent starting at byte number b 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 file (including the file being modified). The new version of the file may be given a new policy with the call **setPolicy**(*tId*, *pId*). The call buf **readTx**(*tId*, off, len) reads len bytes of the file starting at logical offset off in the file and returns the result to the buffer buf. The read rule of the file's policy is evaluated before reading to buf; if it denies access, the call fails. This enforces data confidentiality. Note that we allow byte-level addressing on files, so policies can be very fine-grained.

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

3.3.3 File/Policy interface

While file creations are implemented as transactions, file destruction and policy creation exist as additional calls. The **destroy**(objname) call removes the content and metadata of the file named objname from Guardat after successfully evaluating the destroy permission of the associated policy. To reduce the required metadata space, Guardat allows multiple files to be protected by the same policy. Therefore,

the **createPolicy** call returns a policy Id (*pId*) which can be used multiple times in **setPolicy** calls during a transaction.

3.3.4 Content cache interface

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, write, fresh or readTx commands with appropriate flags (cacheflag). 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.

In order to iteratively build content hashes, Guardat offers the **initHash** call to start the hash computation. If the returned identifier (*hId*) is specified as cacheflag during a read, write, fresh or readTx call, then the respective content is added to the hash computation. After a client finishes the read/write sequence, she closes the hash via the **closeHash** call which finalizes the hash computation and stores the result in the respective session or transaction cache for later use during policy evaluation. In addition a cryptographically signed certificate including the computed hash, file name and a hash of the associated policy is returned.

3.3.5 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 (**getNonce**) to be included in a third-party certificate, and commands to add a signed third-party certificate (**setCertificate**) to the Guardat cache for use in subsequent policy evaluations. Third-party certificates are described further in Section 3.4.

The call **attest**(objname, 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.

3.3.6 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** call invoked at a source Guardat device encrypts a file and its policy for a specific target Guardat device, while the **unpickle** call 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.

3.3.7 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 3.7.4.

3.3.8 Example usage

As an example, we show the sequence of steps required to update an executable file protected by the policy described in Section 3.5. First, a software update application (supd) supplies the required vendor certificate, which is passed by the Guardat library to the GDC to be cached (via **setCertificate**). 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.

3.4 Policy language

Guardat file access policies are specified in an expressive and simple declarative language. Each file'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.

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 [73]. Datalog is a standard foundation for writing access policies [18, 33, 100], known for its clarity, high-level of abstraction and ease of implementation.

3.4.1 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.

3.4.2 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 3.2. 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 content caches. Finally, *certificate predicates* provide information about cached third-party certificates.

Relational, arithmetic and list predicates							
eq(x,y)	x==y	le(x,y)	$x \leq y$	add(x,y,z)	x=y+z	mul(x,y,z)	x=y·z
	or x<-y	ge(x,y)	$x \geq y$	sub(x,y,z)	x=y-z	div(x,y,z)	x=y/z
neq(x,y)	x!=y	lt(x,y)	$x < y$			rem(x,y,z)	x = y mod z
		gt(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 3.2: Guardat policy language predicates

We divide the language's predicates into three categories. *Universal predicates* are available in all policy rules and provide knowledge of the public key that authenticated the session (predicate sessionKeyIs), the name(s) of the file being accessed (predicate fileNameIs), and any content already buffered in the session cache, including the content of other files. Access to the content of other files is necessary for enforcing many policies, including mandatory access logging (MAL) (see Section 3.5).

Rule-specific predicates are available in particular policy rules. In the **read** rule, such predicates provide the length of the read and its logical and physical offsets (hence, policies may be specified at byte-granularity). In the **update** rule, such predicates provide the current and new extents of the file, the current and new file sizes, and the new file contents buffered in the transaction cache. Hence, policies may compare old and new file contents, e.g., the MAL policy requires that the file version number increment at each update.

Finally, Guardat policies may mention arbitrary *external predicates* established through third-party cryptographic certificates.

3.4.3 Third-party certificates

Guardat verifies every certificate provided to it using standard certificate chain verification [30] 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 the required certificates 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 using a pseudorandom number generator, or an explicit expiration time (time server certificates must contain a recent nonce). 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.

3.4.4 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 [18, 100] 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 [131]. We describe the evaluation time complexity of the language in Section 3.7.2.4.

3.4.5 Usability

Declarative languages similar to ours are widely used as policy languages (e.g., XACML, SecPAL, Binder, SD3, and KeyNote [92, 18, 33, 66, 21]) 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 [139, 118, 140]). 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 [76, 77]).

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.

3.5 Policy examples

We illustrate Guardat's capabilities by presenting example policies to protect executables, log files and backups. If the **read** or **update** rule of a policy is omitted, then the permission is always allowed and if a **setpolicy** or **destroy** rule is omitted, then that permission is never allowed.

3.5.1 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}.

$$
\begin{aligned}
&\textbf{update} \ \text{:-} \ \mathsf{file\_name\_is}(F) \ \wedge \ \mathsf{new\_length\_is}(L) \ \wedge \\
&\quad (0, L) \ \mathsf{willHaveHash} \ Nh \ \wedge \ \mathsf{key\_is}(K, \text{“Vendor”}) \ \wedge \\
&\quad K \ \mathsf{signs} \ \mathsf{ok\_hash}(F, N, Nh) \ \wedge \ (N \geq 10) \\
&\textbf{setpolicy} \ \text{:-} \ \mathsf{file\_name\_is}(F) \ \wedge \\
&\quad \mathsf{new\_pol\_hash\_is}(Nph) \ \wedge \\
&\quad k_{\mathsf{ad}} \ \mathsf{signs} \ \mathsf{good\_policy}(F, Nph)
\end{aligned}
$$

The first rule allows an update to the file only if there is a public key K belonging to "Vendor" (condition $\mathsf{key\_is}(K, \text{“Vendor”})$), which signs that the file's new content

hash, Nh, is the Nth version of the executable (condition K signs ok_hash(F, N, Nh)) and $N \geq 10$. The predicates key_is$(K,$ "Vendor") and K signs ok_hash(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, has been certified by the administrator (condition k_{ad} signs good_policy(F, 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). 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.

3.5.2 Append-only logs

The following policy specifies an append-only file that 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.

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

The policy allows an update if either the session is authenticated by the administrator (condition session_is(k_{ad})) or the file's new length Ln exceeds its current length Lo and the first Lo 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.

3.5.3 Protected backup

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

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

The policy allows modification to the file only if the current time exceeds a predetermined time endT. To enforce such policies, Guardat relies on signed certificates from time servers and a short-range internal timing counter. In detail, the policy says that there should be a key K belonging to a time server (condition key_is(K, “TimeServer”)), which issued a certificate that the time was T when the Guardat internal counter had value T_i (condition K signs time(T) at T_i), the current internal counter value is T_j (condition count_is(T_j)) and the current time (calculated as $T + T_j - T_i$) exceeds the backup 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.

3.5.4 Mandatory access logging (MAL)

Legislation and organizational policies often mandate that all read and write access to sensitive information like medical records be logged. Although application-level solutions to enforce such mandatory access logging (MAL) exist, enforcing the policy in Guardat is desirable because it would increase security.

For this exposition, let P be the sensitive file which must be protected by MAL and let L be its log file. We assume that the log file is append-only, through the policy described earlier. The MAL requirement is three-fold:

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.

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.

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.

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.

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: $\mathsf{may\_read}(K, S)$, which indicates that the client with public key K has potentially read the set S of (off,len) ranges from P; and $\mathsf{change}(K, S, H)$, 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 $\mathsf{may\_read}(K, S)$ 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 $\mathsf{is\_session}(K)$ 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_read 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 3.7.5.

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 transation 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 files 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 transaction cache.

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 and committed to Guardat. 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 transactions 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 3.7.5. Applications that cannot accurately estimate their read-sets ahead of time can simply create blanket may_read entries that cover the entire file and periodically *commit read-only transactions* accompanied by special log entries that specify precisely what has been read in the transaction. The precise read set is available to Guardat during a commit transaction, so the log entry's accuracy can be verified. This mode of use requires a second counter in log entries and the sensitive file to count read-only transactions.

3.5.5 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 using Guardat policies), 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 only when the event of creating an appropriate log entry has occurred.

3.5.6 Expressiveness

As these examples demonstrate, the Guardat policy language is expressive. It can express content-based policies like MAL that prior work on 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, or by extending the language with recursive predicates.

3.6 Implementation

This section describes the prototype implementation of Guardat in a SAN server and present implementation alternatives of the Guardat design.

3.6.1 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. Figure 3.1 depicts the component level design. The server accesses an SSD for the Guardat metadata and one or more payload disks which are either magnetic- or flash-based. 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, 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,

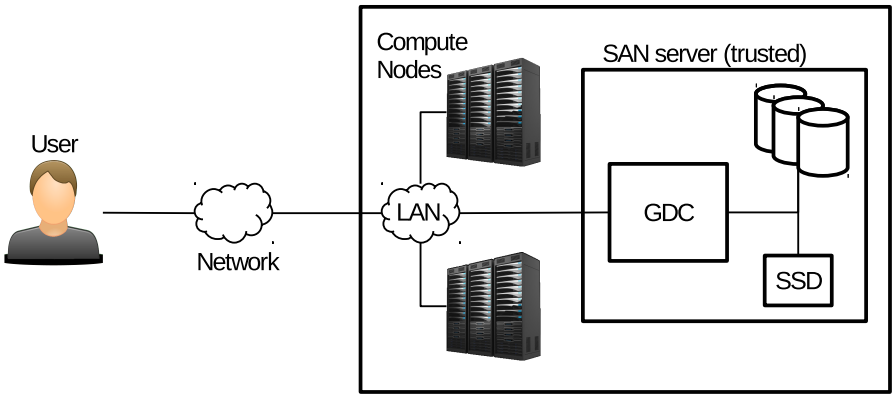


Figure 3.1: Guardat implementation in a SAN server

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. This optimization avoids policy re-evaluation and saves the communication cost between kernel module and the Guardat daemon in many cases.

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 minimally configured Linux kernel, the resulting attack surface is likely to be significantly smaller than that of the systems and applications built on top of Guardat in most cases. Our Guardat implementation adds less than 20,000 LOC to the existing IET codebase, plus the OpenSSL and glib libraries it relies on.

3.6.2 Implementation alternatives

Guardat can be implemented in different ways depending on the deployment and threat model. The GDC can be implemented using the following mechanisms:

(a) In a SAN server for use in a data center, as described in the previous prototype section.

(b) Integrated with the microcontroller of a hybrid disk for use in an individual machine.

(c) In a trustlet within a virtual machine monitor or operating system, isolated using trusted hardware features like Intel SGX [63] or ARM TrustZone [12].

Table 3.3 lists examples of deployment scenarios, their threat models and trust assumptions. As described in the threat model each implementation must protect the GDC, metadata and data from unauthorized physical access and undetected tampering

Implementation (a) relies on physical protection, e.g., in a machine room with access only by trusted employees. A possible deployment scenario at a Cloud provider protects user data from bugs and misconfigurations in its infrastructure and from opportunistic access by employees. The user must trust the Cloud provider to prevent physical access to the SAN server by all but trusted employees.

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 au-

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

Table 3.3: Guardat deployment scenarios and trust assumptions

thenticated 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.

Implementation (c) has similar security properties, except that the GDC executes on the main CPU and trust is derived from this CPU's trusted isolation capabilities.

3.6.3 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. Unmodified filesystems can be used with many policies. In fact, all policies described in Section 3.5 except MAL operate with an unmodified ext4 filesystem.

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 transaction. We note that the library is untrusted and does not require extra privileges. In particular, the library only executes those 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 transaction or session can be open for a given object at any given time in compatibility mode. Confidential objects cannot in general be accessed through the file system, because that system's block cache may deliver data encrypted for one client to another client. Therefore, an application library reads such objects through the IOCTL interface, by first looking up the object's list of extents, and then issuing reads via IOCTL.

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 transaction. 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 transaction 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. Finally, an unmodified filesystem that overwrites blocks in place cannot be used with certain integrity policies in compatibility mode. These limitations can be lifted by modifying a filesystem to use the extended Guardat API.

3.6.4 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. Recent work [84] enforces similar data confidentiality and integrity policies over databases.

3.7 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 3.5.

3.7.1 Experimental setup.

The Guardat-enhanced IET SAN server (based on version 1.4.20.3-9.6.1) [65] 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 [111], or on a 512GB Samsung SSD [106]. The Guardat metadata is stored on a OCZ Deneva 2 C SLC 60GB (raw 64GB) SSD [93]. 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 [96], Intel AES encryption library [61], and Intel's fast SHA256 implementation [62] are used for Guardat cryptographic operations.

3.7.2 Microbenchmarks

3.7.2.1 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.

Each configuration is exercised with two access types (**R**ead and **W**rite) and three different access patterns:

Sequential: Blocks are accessed in order of increasing block id.

Local: Each accessed block chosen randomly within 40,000 block ids of the previous block.

Random: Each accessed block chosen randomly on the entire disk.

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

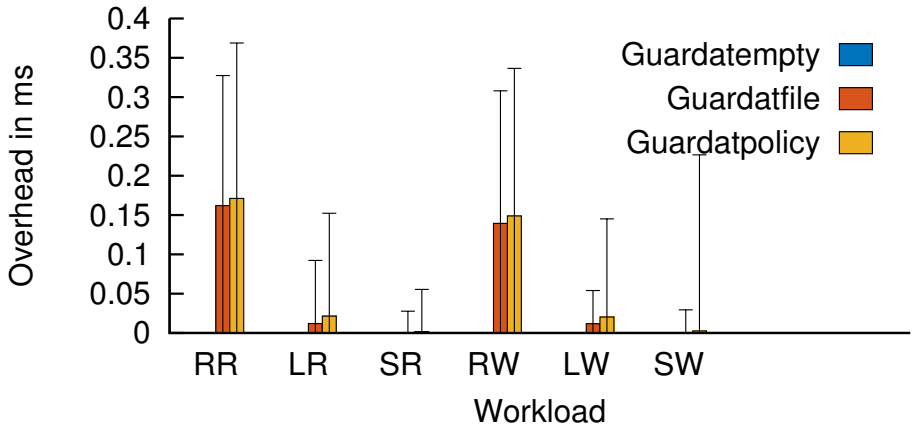


Figure 3.2: Absolute Guardat latency overhead. (Overhead of Guardatempty invisible <2.5 µs.)

(a total of 100,000 accesses for each configuration). Each run starts at a randomly chosen location on the disk.

Figure 3.2 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 *blue* bars invisible in Figure 3.2. 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.

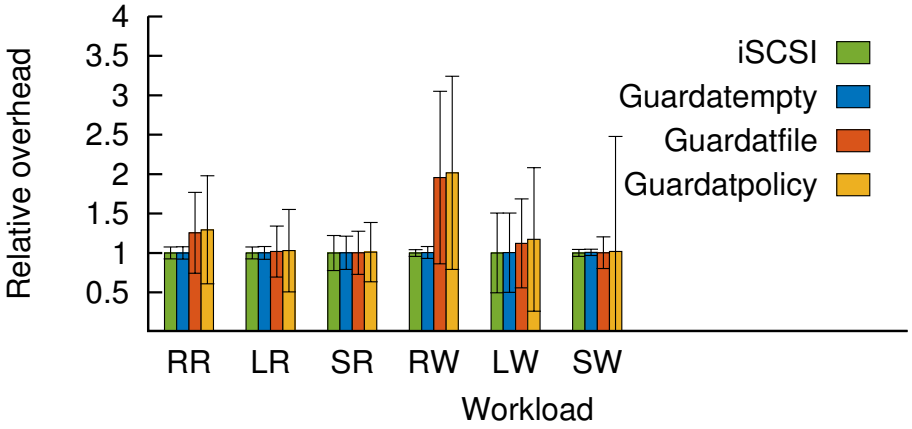


Figure 3.3: Latency with an SSD, relative to iSCSI

Figure 3.3 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 3.4 and 3.5 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

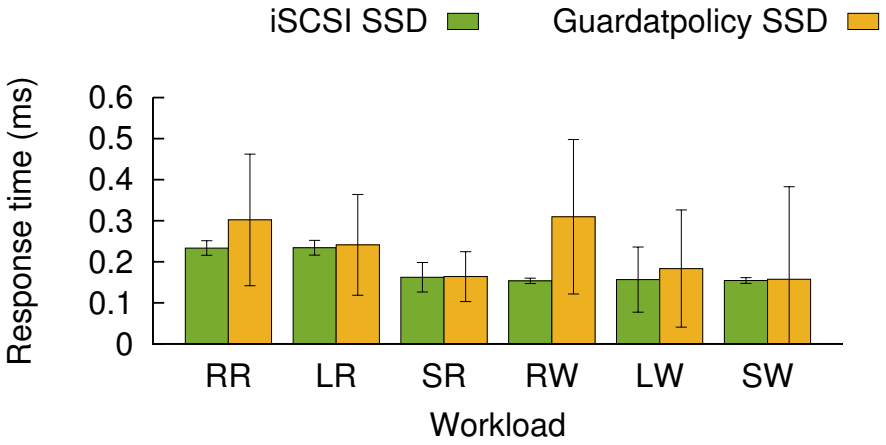


Figure 3.4: Absolute latency with SSD

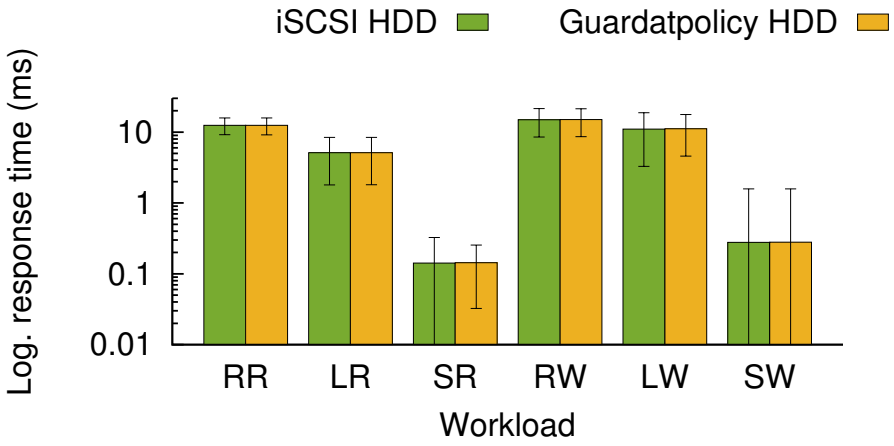


Figure 3.5: Absolute latency with HDD

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).

3.7.2.2 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

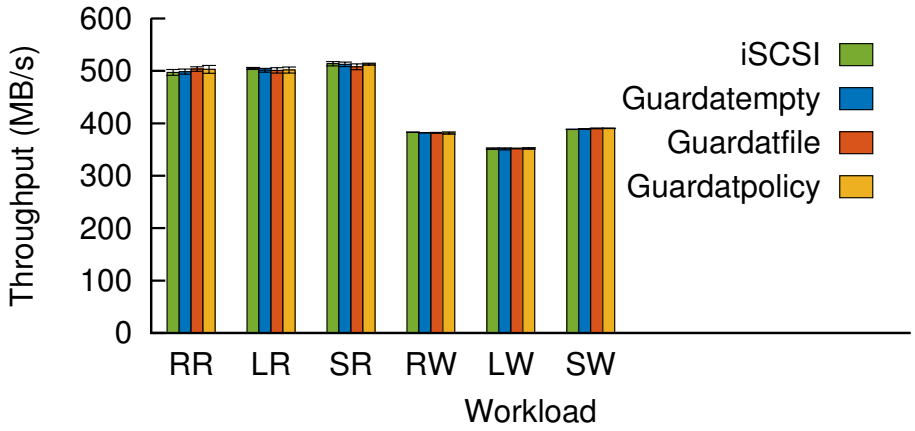


Figure 3.6: SSD I/O throughput

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 3.6 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.

3.7.2.3 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

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

more locality, and can be overlapped with concurrent accesses, so that the SSD's throughput is not affected.

3.7.2.4 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 3.4, 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 3.7.5) 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.

3.7.2.5 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. [2]. 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 [112] at the time of this writing. Newer combinations of Flash/disk devices achieve much higher Flash to disk capacity ratios and this trend is projected to continue given the price and space reduction rates of flash memory. For example, Apple's Fusion Drive [10] 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.

3.7.2.6 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

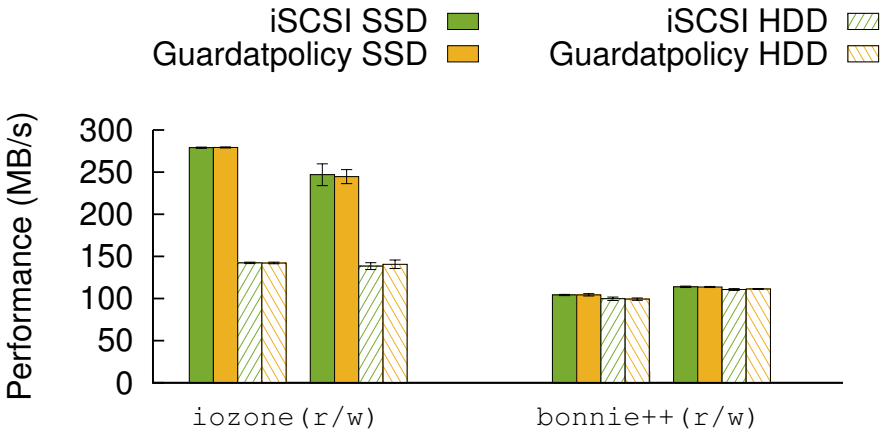


Figure 3.7: FS benchmarks read and write (r/w) performance

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.

3.7.3 Filesystem benchmarks

Next, we measure the performance of the Guardat prototype using the standard filesystem 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.[1] Later, each worker performs a sequential read of the file they previously wrote. Similarly, `Bonnie++` writes then reads 1GB each to 16 files. Figure 3.7 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 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).

3.7.4 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 [137] and Wikimedia images from 2005 [136], 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 [138]. 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.

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

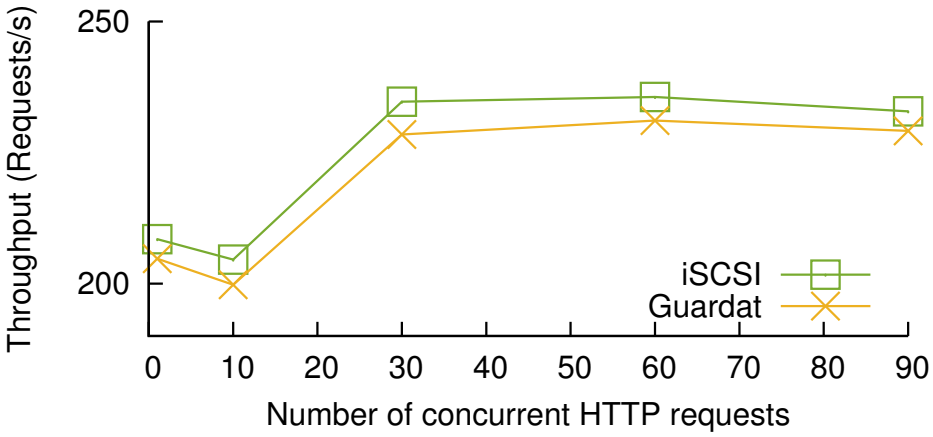


Figure 3.8: Web server throughput

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 3.8 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.

3.7.5 Mandatory access logging

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. (horizontal lines)

log: Accesses logged without policy enforcement.

Guardat MAL: Accesses logged and policy enforced by Guardat.

Figure 3.9 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

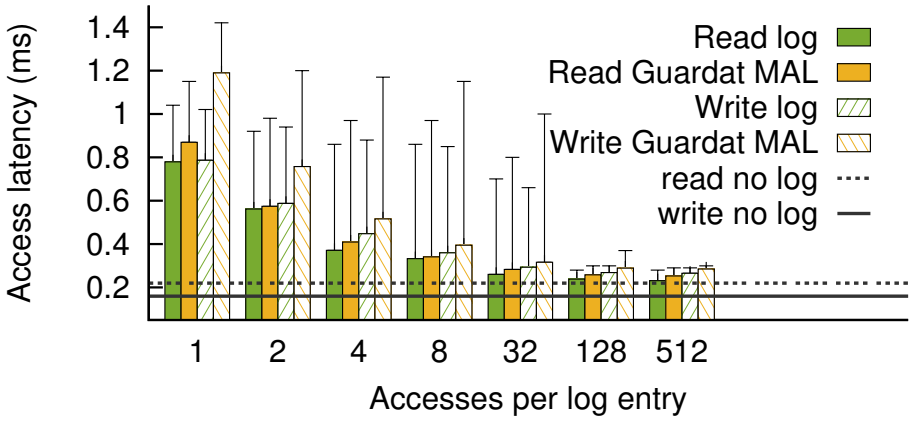


Figure 3.9: Latency with MAL, voluntary and no logging

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.

3.8 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 [100], Binder [33] and SecPAL [18]. Whereas these languages are generic, the Guardat policy language is domain-specific and contains custom-designed, storage-relevant predicates (Section 3.4). 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 [52] 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) [127]. 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 [123], 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 [97], 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 [54] 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 [107] 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 [69] 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 3.5.4, we describe a similar MAL policy in Guardat.

VMM/OS data protection. Overshadow [28] 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 [57], 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 [118], like the earlier Taos operating system [139], 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 [108]. 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 [142] 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. [27, 25, 26] 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 [110] 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 [5] 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) [48, 3, 40], 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) [119] 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.

To address the specific problem of accidental or malicious corruption of backup data, one possibility is to restrict the ability to overwrite that data. The simplest form is to use a write-once storage medium like a DVD. Similarly, Venti [101] is a centralized storage service that implements a write-once interface and is intended as a storage back end for archival data. Internally, data is stored on a RAID disk group. Data blocks with identical content are coalesced prior to being stored. Write-once solutions, however, will accumulate stored data over time. To avoid this inflexibility, where data needs to be retained forever and the storage medium can never be reused, one can disable writes for a given time period. This can be achieved in various ways.

Storage systems such as Self Securing Storage (S4) [124] and NetApp's SnapVault [55] 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 3.5), as well as confidentiality and access accounting.

Protected filesystems. jVPFS [134, 133] 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 [51] states a filesystem's metadata invariants as SQL queries, and checks/repairs these invariants off-line. Recon [44] 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 [45] and PFS [130] 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.

Protecting data availability. Storage systems like RAID [98], snapshotting filesystems [56, 125, 89] and some backup utilities [9, 88] 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 [46]). 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 [112] 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 [86]. 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 [109] 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 [102], semantically smart disks [120] and differentiated storage services [85] include program logic to improve performance. Guardat addresses the orthogonal concerns of data confidentiality, integrity and access accounting.

Pennington et al. [99] 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.

3.9 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 our experimental evaluation.

CHAPTER 4

ERIM: Secure and Efficient In-process Isolation

The previous chapter described the design, implementation and evaluation of Guardat, a storage layer reference monitor enforcing confidentiality, integrity and accounting policies over persistent data. It enforces these policies independent of higher abstraction layers providing a strong threat model relying on a small TCB and attack surface.

However, the set of enforceable policies are limited by the observable events at the storage layer. Guardat has no control over data released to an application with sufficient access credentials. As a results an application may leak data over the network, due to bugs, malicious attacks or misconfigurations.

In contrast to Guardat, ERIM mediates an untrusted application's execution, intercepting relevant application operations like accesses to private data or operating system services. To mediate untrusted applications, ERIM partitions sensitive data and code into an isolated and trusted component, thereby limiting the effects of bugs and vulnerabilities in the untrusted component to data accessible in the untrusted application only. For instance, isolating cryptographic keys from the remaining application can thwart vulnerabilities like the OpenSSL Heartbleed bug [90]; isolating jump tables can prevent attacks on the integrity of an application's control flow; and isolating a managed language's runtime can protect its security invariants from bugs and vulnerabilities in co-linked native libraries.

Isolation foremost requires *memory isolation*, which prevents an untrusted component from directly accessing the private memory of other components. Broadly speaking, memory isolation can be enforced using one of two approaches. First, we may instrument the code of untrusted components with bounds checks prior to indirect memory accesses, ensuring that memory of other components is not accessed directly, as in SFI [129]. However, this approach imposes overhead on all execution of untrusted components due to bounds checks, and it requires an additional technique to prevent circumvention of the bounds checks in the face of control-flow hijacks [68]. The total overhead is commonly of the order of tens of percent points.

The second approach is to use hardware support for memory isolation such as OS or hypervisor (extended) page tables [20, 29, 74, 19]. Here, fast access checks in hardware prevent a component from accessing the memory of other components, but there is an overhead on *switches* between components, since hardware privileges must be changed, e.g., by switching page tables and possibly invalidating TLB entries. Recent work on *in-process isolation* such as Wedge [20], Shreds [29], and light-weight contexts (lwCs) [74] has reduced the cost of hardware-based isolation somewhat. Nonetheless, switching still requires a system call and its cost is significant (at 1 us per switch [74] a conservative switch rate of 100,000 times a seconds amounts to 10% overhead).[1]

Consequently, there is need for an isolation technique that does not impose continuous overhead *while* a component executes and that also has very low *switching cost* on component transition. ERIM achieves this goal by building on a recent x86 ISA extension called memory protection keys or MPKs, also simply called protection keys [64]. MPKs allow tagging each page with one of 16 domains, thus partitioning a process' address space into disjoint domains. A special per-core register, PKRU,

[1]Using x86 memory segmentation instead of page tables, as in Native Client [143], can reduce the switch cost. However, support for segmentation with 64-bit addressing is limited and Native Client has been deprecated in favor of the memory-safe language WebAssembly [53].

determines which domains are accessible. Switching permissions requires only writing the PKRU register with a user-mode instruction, which is a relatively quick operation (11–260 cycles on current Intel CPUs in our experiments).

However, since the PKRU-update instruction is user-mode, MPK by itself in insufficient for security: Compromised or malicious components can execute the instruction to gain unauthorized access to the memory of other components. To prevent this, ERIM additionally relies on binary inspection to ensure that all occurrences of this instruction (called WRPKRU) in the binary are *safe*, i.e., they cannot be exploited to gain unauthorized access. By design, this property holds even if there is a control-flow hijack in the untrusted component. Hence, there is no need to complement ERIM with control-flow integrity, which would add overhead.

ERIM distinguishes itself from prior work on applications that have very high switching rates (~10^5/s or more) and that additionally spend a nontrivial amount of time in untrusted components. There are many such applications. We evaluate our prototype of ERIM on three such applications. First, in the web server nginx, we show that ERIM can isolate *session keys*. Protecting session keys is a meaningful goal, since attacks targeted at individual users' privacy only need to compromise session keys. Second, we show that ERIM can efficiently isolate the safe region in code-pointer integrity [72]. Third, we show that ERIM can be used to isolate a managed language runtime from possibly buggy native libraries. In all cases, we observe switching rates of orders at least 10^5 times/s per core. ERIM provides strong hardware isolation with overheads less than 1% for every 100,000 switches/s, which is considerably lower than that of existing techniques.

The following sections describe the design, three use cases, a prototype implementation, related work and the evaluation using three use cases.

4.1 Design

Like prior work, ERIM enables a *trusted* application component to isolate sensitive data from the rest of the *untrusted* application. Unlike prior work, ERIM supports such isolation with *low overhead* even at *high switching* rates between the untrusted application and the trusted component, and without relying on any other (possibly expensive) protection mechanism, e.g., control-flow integrity. By way of example, the trusted component may be a crypto library that wants to isolate cryptographic keys, an inlined reference monitor that wants to isolate sensitive meta data (such as a taint map or jump tables), or it may be a managed language runtime that wants to isolate from a buggy native library. We use the letter T to denote the trusted component and U to denote the remaining untrusted application.

The main primitive ERIM provides is memory isolation—it reserves a region of the address space accessible exclusively from the trusted component T. This reserved region is denoted M_T and it can be used by T to store sensitive data. The rest of the address space, denoted M_U, holds the application's regular heap and stack and is accessible from both U and T. ERIM prevents U from having direct access to M_T; access to M_T is enabled atomically with a control transfer to designated entry points in T, and disabled when T returns control to U. More precisely, ERIM enforces the following invariants:

(1) While control is in U, access to M_T remains disabled.

(2) Access to M_T is enabled atomically with a control transfer to a designated entry point in T and disabled when T transfers control back to U.

The first invariant provides isolation of M_T from U, while the second invariant prevents U from confusing T into accessing M_T improperly by jumping into the middle of M_T's code. Due to the second invariant, ERIM does not need support from a solution for control-flow integrity for security.

Control transfers from U to T and back, with the corresponding enabling and disabling of access to M_T are facilitated by special sequences of ERIM-provided code, dubbed *call gates*. Call gates are implemented in a manner that prevents exploitation of their binary code for elevating privileges.

A call gate enables access to M_T, executes a specified entry point of T, then disables access to M_T when transferring control to the trusted component and disables access on the way back. A call gate transfers control only to a designated entry point in the trusted component. This entry point may be a function (if the trusted component is a library) or a specific sequence of instructions (if the trusted component is inlined into the application).

By design, ERIM imposes negligible overhead on the execution of code within the untrusted application and within the trusted component, and its call gates are very fast. Additionally, ERIM's isolation is strong—it is derived directly from a hardware security feature and it is absolute, not probabilistic (unlike address space layout randomization (ASLR)). Both, fast switching and the strong isolation, make ERIM suitable to protect sensitive data in high-performance applications, even those that switch between the application and the library very frequently.

4.1.1 Threat model

ERIM makes no assumptions about the untrusted component (U) of an application. U may behave arbitrarily and may contain memory corruption and control-flow hijack vulnerabilities that may be exploited during its execution.

However, ERIM assumes that the trusted component T's binary does not have such vulnerabilities and does not compromise sensitive data by calling back into U while access to M_T is enabled, through information leaks, or by mapping executable pages with unsafe/exploitable occurrences of the WRPKRU instruction.

The hardware, the OS kernel, and a small library added by ERIM to each process that uses ERIM are trusted to be secure. We also assume that the kernel enforces standard DEP—an executable page must not be simultaneously mapped with write permissions. ERIM relies on a list of legitimate entry points into T provided either by the programmer or the compiler, and this list is assumed to be correct (see Section 4.1.5). The OS's dynamic program loader/linker is trusted to invoke ERIM's initialization function before any other code in a new process.

Side-channel and rowhammer attacks, and microachitectural leaks, although important, are beyond the scope of this work. However, ERIM is compatible with existing defenses.

4.1.2 Intel Memory Protection Keys (MPK)

To realize its goals, ERIM uses the recent MPK extension to the x86 ISA [64]. MPK allows associating one of 16 protection keys with each memory page, thus partitioning the address space into up to 16 *domains*. A per-core register, called PKRU, determines the current access permissions (read, write, neither or both) on each domain for the code running on that core. Access checks against the PKRU are implemented in hardware and impose no overhead on program execution.

Changing access privileges requires writing new permissions to the PKRU register with a *user-mode* instruction, WRPKRU. This instruction is relatively fast (11–260 cycles on current Intel CPUs), does not require a syscall, changes to page tables, a TLB flush, or inter-core synchronization.

Since WRPKRU can be executed in user-mode, untrusted code can execute it at any point to elevate privileges and MPK cannot provide any memory security against untrusted application code by itself. To get this protection, ERIM combines MPK with additional binary inspection to ensure that any WRPKRU occurrences

on executable pages are *safe*, i.e., they cannot be exploited to improperly elevate privilege.

The mainstream Linux kernel fully supports page-table entries tagged with MPK domains, syscalls to tag the entries with specific domains and restores PKRU registers upon context switches. Since hardware PKRU checks are disabled in kernel mode, the kernel has also been modified to check PKRU permissions explicitly before accessing any userspace pointer. To eliminate the risk of signal handlers elevating privileges, the kernel updates the PKRU register to its initial set of privileges (only read/write access to domain 0) before throwing a signal to the userspace.

4.1.3 High-level overview of the design

ERIM can be configured to provide either complete isolation of M_T from U (confidentiality and integrity), or only write protection (only integrity). For simplicity, we describe the design for complete isolation first. Section 4.1.7 describes how to configure ERIM slightly differently to provide write protection only.

ERIM's isolation mechanism is conceptually simple: It maps T's reserved memory, M_T, and the application's general memory, M_U, to two different MPK domains. It manages MPK permissions (the per-core PKRU registers) to ensure that M_U is always accessible, while M_T is never accessible when control is in U. It allows U to securely transfer control to T and back via *call gates*. A call gate enables access to M_T using the WRPKRU instruction and immediately transfers control to a specified entry point of T, which may be an explicit or inlined function. When T is done executing, the call gate disables access to M_T and returns control to U. This enforces ERIM's two invariants (1) and (2) from Section 4.1. Call gates operate entirely in user-mode (they don't use syscalls) and are described in Section 4.1.4.

Preventing WRPKRU exploitation A key difficulty in ERIM's design is preventing the untrusted U from exploiting WRPKRU instructions on executable pages

in the address space to elevate privileges, e.g. using control-flow hijack or code-injection attacks. To prevent such exploits, ERIM relies on *binary inspection* to enforce the invariant that only *safe* WRPKRU occurrences appear on executable pages. A WRPKRU occurrence is safe if it is immediately followed by one of the following:

(A) A pre-designated entry point of T.

(B) A specific sequence of instructions that checks that the permissions set by the WRPKRU do not include access to M_T and terminates the program otherwise.

A safe WRPKRU occurrence cannot be exploited to execute untrusted code with access to M_T. If the occurrence satisfies (A), then it does not give control to U at all; instead, it enters T at a designated entry point. If the occurrence satisfies (B), then it would terminate the program immediately were it used by a control-flow hijack to enable access to M_T.

ERIM's call gates use only safe WRPKRU occurrences and, therefore, pass our binary inspection. Our modified kernel *inspects* any page prior to mapping it in executable mode, enforcing the invariant that all occurrences of WRPKRU on executable pages are safe. Section 4.1.5 provides details of this kernel binary inspection mechanism.

Creating safe binaries An important question is how to construct binaries that do not have unsafe WRPKRUs. On x86, an *inadvertent* or unintended executable WRPKRU may arise spanning the bytes of two adjacent instructions or as a subsequence in a longer instruction. To eliminate inadvertent WRPKRUs, we develop a binary rewriting mechanism that rewrites any sequence of instructions containing an inadvertent WRPKRU to a functionally equivalent sequence without any WRPKRUs. Similarly, the mechanism also alters deliberate uses of WRPKRU which voluntarily switch domains by inserting privilege checks. The mechanism can be deployed as

```
  xor ecx, ecx                                                    1
  xor edx, edx                                                    2
  mov PKRU_ALLOW_TRUSTED, eax                                     3
  WRPKRU // copies eax to PKRU                                    4

// Execute trusted component's code                             6

  xor ecx, ecx                                                    8
  xor edx, edx                                                    9
  mov PKRU_DISALLOW_TRUSTED, eax                                  10
  WRPKRU // copies eax to PKRU                                    11
  cmp PKRU_DISALLOW_TRUSTED, eax                                  12
  je continue                                                     13
  syscall exit // terminate program                               14
continue:                                                         15
// control returns to the untrusted application here            16
```

Listing 4.1: Call gate implementation in assembly. The code of the trusted component's entry point may be inlined by the compiler on line 6, or there may be an explicit direct call to it.

a compiler pass, integrated with our binary inspection, or by statically rewriting binaries prior to their use as explained in Section 4.2

4.1.4 Call gates

A call gate transfers control from U to T, enabling access to M_T, then runs code from a designated entry point of T, and later returns control to U after disabling access to M_T. This requires two WRPKRUs. The primary challenge in designing the call gate is ensuring that both these WRPKRUs are safe in the sense explained in Section 4.1.3.

Listing 4.1 shows the assembly code of a call gate. WRPKRU expects the new PKRU value in the eax register and requires ecx and edx to be 0. The call gate works as follows. First, it sets PKRU to enable access to M_T (lines 1–4). The macro PKRU_ALLOW_TRUSTED is a PKRU setting that allows access to M_T. Next, the call gate passes control to the designated entry point of T (line 6). The entry point's code may be invoked either by a direct call, or it may be inlined here.

After T has finished, the call gate sets PKRU to disable access to M_T (lines 8–11). The macro PKRU_DISALLOW_TRUSTED is a PKRU setting that excludes access to M_T. Next, the call gate checks that the PKRU was actually loaded with PKRU_DISALLOW_TRUSTED (line 12). If this is not the case, it terminates the program (line 14), else it returns control to U (lines 15–16). It may seem that the check on line 12 is pointless since it will always succeed (eax is set to PKRU_DISALLOW_TRUSTED on line 10). While this will be the case under normal operation, the check prevents *exploitation* of the WRPKRU on line 11 with control flow hijack attacks (explained next).

Safety Both occurrences of WRPKRU in the call gate are safe—neither can be exploited by a control flow hijack to get unauthorized access to M_T. Specifically, the first occurrence of WRPKRU (line 4) is of form (A)—there is a control transfer to a specific, designated entry point of T right after the WRPKRU. This occurrence cannot be exploited to transfer control to anything else. The second occurrence of WRPKRU (line 11) is followed by a check that terminates the program if the new permissions include access to M_T. If, as part of an attack, the execution jumped directly to line 11 with PKRU_ALLOW_TRUSTED in eax, the program would be terminated on line 14.

Efficiency A call gate's overhead on a roundtrip from U to T is two WRPKRUs, a few very fast, standard register operations and one conditional branch instruction. This overhead is very low compared to other hardware isolation techniques that rely on inter-process communication, syscalls or hypervisor trampolines to change privileges.

Use considerations ERIM's call gate omits features that some readers may naturally expect. These features have been omitted to avoid having to pay their overhead when they are not needed. First, the call gate does not include support to pass parameters from U to T or to pass a result from T to U. Instead, parameters and

return values can be passed via a designated shared buffer in M_U (both U and T have access to M_U). Second, the call gate does not scrub registers when switching from T to U. Consequently, if T uses confidential data, it must scrub any secrets from registers before returning to U. Further, because T and U share the call stack, T must also scrub secrets from the stack prior to returning. Alternatively, T can allocate a private stack for itself in M_T, and T's entry point can switch to that stack as soon as it is invoked. This will prevent T's secrets from being written to U's stack in the first place. (Such a private stack is also necessary for multi-threaded applications; see Section 4.1.7).

4.1.5 Binary inspection

Next, we describe ERIM's binary inspection. This mechanism prevents U from mapping any executable pages with unsafe WRPKRU occurrences. The mechanism relies on a simple kernel modification that prevents U, but not T, from mapping any page with execute permissions using calls like `mmap` and `mprotect`. (Whether such a call is made by U or T is easily determined by examining the PKRU register.) Instead, any such page is mapped read-only, and the kernel records in a buffer shared with T that the page is *supposed* to be executable pending inspection.

If and when control transfers to such a page, a fault occurs. The fault traps to a dedicated signal handler, which ERIM installs when it initializes (a further kernel modification prevents U from overriding this signal handler). This signal handler calls a T function which checks that the faulting page is pending inspection and, if so, it *scans* the page and the beginnings and ends of surrounding pages for occurrences of WRPKRU. For every WRPKRU, it checks that the WRPKRU is safe, i.e., either condition (A) or condition (B) from Section 4.1.3 holds. If so, the handler remaps the page with the execute permission and resumes execution of the faulting instruction, which will now succeed. If not, the program is terminated. This mechanism has very

low overhead in practice (it scans an executable page at most once—when the page is first used), it enforces that WRPKRU occurrences on executable pages mapped by U are always safe, and it is fully transparent to U's code if all its WRPKRUs are already safe.

To check for condition (A), ERIM must be provided a list of designated entry points of T. The source of this list depends on the nature of T and is trusted. If T consists of library functions, then the programmer marks these functions, e.g., by including a unique character sequence in their names. If the functions are not inlined by the compiler, their names will appear in the symbol table. If T's functions are subject to inlining or if they are generated by a compiler pass, then the compiler must be directed to add their entry locations to the symbol table with the unique character sequence. In all cases, ERIM can identify designated entry points by looking at the symbol table and make them available to the signal handler.

Condition (B) is checked easily by verifying that the WRPKRU is immediately followed *exactly* by the instructions on lines 12–15 of Listing 4.1. These instructions ensure that the WRPKRU cannot be used to enable access to M_T and continue execution.

Security We briefly summarize how ERIM attains security. The binary inspection mechanism prevents U from mapping any executable page with an unsafe WRPKRU. T does not contain any executable unsafe WRPKRU by assumption. Consequently, only safe WRPKRUs are executable in the entire address space at any point, and they transfer control to one of T's designated entry points, which are safe by assumption. Safe WRPKRUs preserve ERIM's two security invariants (1) and (2) *by design*. Hence, M_T remains isolated from U.

4.1.6 Process lifecycle with ERIM

Besides the simple kernel changes mentioned in Section 4.1.5, all of ERIM is implemented as a runtime library that is linked into a process binary either statically or at load time through LD_PRELOAD. This library provides a memory allocator for domain M_T (this allocator can be used by T to allocate objects in M_T) and, importantly, an initialization function, which is invoked by the standard OS loader before U's main().

The initialization function, called init here, creates the memory domain M_T and maps memory to it (M_U occupies the default MPK domain, which is automatically created with the process). It then loads T's code and data from a dynamic link library. Next, init scans the code of T for unsafe WRPKRUs, sets up call gates to enable control transfer to T's entry points, and installs the signal handler mentioned in Section 4.1.5. Finally, init scans U's code, disables access to M_T and transfers control to U's main().

After main() has control, U executes almost as usual. It maps and unmaps memory in the domain M_U using the standard system memory allocator. However, to access T's exported services, U must invoke a call gate to enable access to M_T and invoke a T entry point. Hence, U's binary must be constructed to invoke call gates to T at appropriate points. This is done using a combination of two techniques. First, LD_PRELOAD can be used to re-link explicit T function calls to a library of wrappers that invoke a call gate. Second, T invocations from functions inserted by the compiler can be made to directly invoke the call gate by modifying these functions. We use this method in our application of ERIM to CPI [72] to invoke a call gate at every sensitive region update.

4.1.7 Other considerations

Multi-threaded applications ERIM's design works as-is with multi-threaded applications because MPK uses a per-core PKRU register. Threads are created as usual, e.g., using libpthread. As the PKRU register is saved during context switches, a new thread starts executing with its parent's PKRU register. However, multi-threading imposes an additional requirement on T (not on ERIM): In a multi-threaded application, it is essential that T allocate a private stack in M_T (not M_U) for each thread and execute its code on these stacks. This is easy to implement by switching stacks at T's entry points. Not doing so and executing T on standard stacks in M_U runs the risk that, while a thread is executing in T, another thread executing in U may corrupt or read the first thread's stack frames. This can potentially destroy T's integrity, leak its secrets and hijack control while access to M_T is enabled. By executing T on stacks in M_T, such attacks are prevented.

ERIM for integrity only Some applications care only about the integrity of data, but not its confidentiality. Examples include CPI, which needs to protect only the integrity of code pointers. In such applications, efficiency can be improved by allowing U to *read* M_T directly, thus avoiding the need to invoke a call gate for reading M_T. The ERIM design we have described so far can be easily modified to support this. Only the definition of the constant PKRU_DISALLOW_TRUSTED in Listing 4.1 has to change to also allow read-only access to M_T. With this change, read access to M_T is always enabled.

Just-in-time (jit) compilers using ERIM Existing jit compilers allocate new code pages as writable, and alter the page permissions to execute-only once the compilation finishes. At this time ERIM's kernel module maps the page without execute permission. When execution reaches the newly compiled code a segmentation fault occurs. ERIM's dynamic binary inspection scans the page and only enables the

execute bit if no unsafe WRPKRU exists. This mechanism is safe, but may lead to program crashes as the jit compiler does not necessary emit WRPKRU free code.

ERIM-aware jit compilers can emit WRPKRU free binary code by relying on the rewrite strategy described in Section 4.2, and inserting call gates when necessary. An additional optimization could inform ERIM's binary inspection mechanism at the end of the jit compiler's pipeline to scan the page for WRPKRUs and enable the execute permission. This lowers the number of segmentation faults, but requires jit compilers to support ERIM.

In addition to supporting ERIM, jit compilers can prevent memory-corruption attacks [47] from, e.g., corrupting the jit compiler's state using ERIM's memory isolation. ERIM's memory isolation can efficiently protect the jit compiler's state by isolating the jit compiler in the trusted domain, while the application runs in the untrusted domain. As a result, ERIM prevents the untrusted application from accessing the jit compiler's state preventing memory-corruption attacks. Compared to existing work [42] which relies on Intel SGX to isolate the compiler's state, ERIM's isolation is highly efficient.

OS privilege separation (extension) The design described so far provides memory isolation. Some applications, however, require privilege separation between T and U with respect to OS resources. For instance, an application might need to restrict the filesystem name space accessible to U or restrict the system calls available to U. For example, suppose that the goal of using ERIM is to hide a long-term cryptographic key that is also backed to a file. Unless U's access to the file is also restricted, no amount of memory isolation can effectively hide the key.

ERIM's design can be easily extended with a few additional kernel changes to support privilege separation with respect to OS resources. First, during process initialization, ERIM's init function can instruct the kernel to restrict U's access rights. After this step, the kernel refuses to grant access to restricted resources whenever

the present value of the PKRU is not PKRU_ALLOW_TRUSTED, indicating that the syscall does not originate from T. To gain access to restricted resources, U has to invoke T, which can act as a reference monitor.

4.2 Rewriting inadvertent WRPKRUs

For security, our binary inspection (see Section 4.1.5) requires binaries to have only safe WRPKRU occurrences. WRPKRUs emitted purposefully by a compiler can be made safe by changing the compiler slightly to insert the check on lines 12–15 of Figure 4.1 after every potentially unsafe WRPKRU. Inadvertent WRPKRUs—those that occur unintentionally as parts of longer x86 instructions or spanning two consecutive x86 instructions—are more interesting. In this Section, we describe a rewrite strategy to eliminate such WRPKRUs. The strategy is *complete*: Any sequence of x86 instructions containing an inadvertent WRPKRU can be rewritten to a functionally equivalent sequence without any WRPKRUs.

4.2.1 Rewrite strategy

WRPKRU is a 3 byte instruction, 0x0F01EF. WRPKRU sequences that span two or more instructions can be "broken" by inserting a 1 byte nop like 0x90 between any two consecutive instructions. 0x90 does not coincide with any individual byte of WRPKRU (0x0F, 0x01 and 0xEF), so this insertion cannot generate a new WRPKRU.

A WRPKRU sequence that lies entirely within a longer instruction can be eliminated by finding an equivalent sequence of instructions. Doing so systematically requires understanding x86 instruction coding. An x86 instruction consists of:

(i) An opcode field possibly with prefix.

(ii) A MOD R/M field that determines the addressing mode and includes the code for a register operand.

(iii) An optional SIB field that specifies registers for indirect memory addressing.

(iv) Optional displacement and/or immediate fields which specify constant offsets for memory operations and other constant operands.

Our strategy for rewriting an instruction containing WRPKRU as a subsequence depends on the fields with which the WRPKRU subsequence overlaps. Table 4.1 summarizes our strategy. If the WRPKRU sequence lies entirely in the opcode field, then the instruction *is* WRPKRU. As explained earlier, this case is handled by adding a check (B) after the instruction to make it safe.

If the sequence overlaps with the MOD R/M field, we change the register code in the MOD R/M field, which eliminates the WRPKRU sequence. This change requires a free register. If one exists, we use it, else we rewrite to push an existing register to the stack, use it in the instruction, and pop it back. (Lines 2 and 3 in Table 4.1.)

If the sequence overlaps with the displacement or the immediate field, we change the mode of the instruction to use a register instead of a constant. The constant is computed in the register before the instruction (lines 4 and 6). If a free register is unavailable, we push and pop one. Two instruction-specific optimizations are possible. If the instruction is jump-like, then the jump target can be relocated in the binary; this changes the displacement in the instruction, eliminating the need for a free register (line 5). If the instruction is an associative operation such as addition, then the operation can be performed in two increments without an extra register (line 7).

We never rewrite the SIB field. This does not affect the completeness of our technique since any WRPKRU must overlap with at least one non-SIB field (the SIB field is 1 byte long while WRPKRU is 3 bytes long).

Overlap with	Cases	Rewrite strategy	ID	Example
Opcode	Opcode = WRPKRU	Insert privilege check after WRPKRU	1	
Mod R/M	Mod R/M = 0x0F	Change to unused register + move command	2	add ecx, [ebx + 0x01EF0000] → mov eax, ebx; add ecx, [eax + 0x01EF0000];
		Push/Pop used register + move command	3	add ecx, [ebx + 0x01EF0000] → push eax; mov eax, ebx; add ecx, [eax + 0x01EF0000]; pop eax;
Displacement	Full/Partial sequence	Change mode to use register	4	add eax, 0x0F01EF00 → (push ebx;) mov ebx, 0x0F010000; add ebx, 0x0000EA00; add eax, ebx; (pop ebx;)
	Jump-like instruction	Move code segment to alter constant used in address	5	call [rip + 0xffef010f] → call [rip + 0xffef0100]
Immediate	Full/Partial sequence	Change mode to use register	6	add eax, 0x0F01EF → (push ebx;) mov ebx, 0x0F01EE00; add ebx, 0x00000100; add eax, ebx; (pop ebx;)
	Associative opcode	Apply instruction twice with different immediates to get equivalent effect	7	add ebx, 0x0F01EF00 → add ebx, 0x0E01EF00; add ebx, 0x01000000

Table 4.1: Rewrite strategy for intra-instruction occurrences of WRPKRU

4.2.2 Implementing the rewriting

For binaries that can be (re)compiled from source, rewriting can be added to the codegen phase of the compiler, which converts the intermediate representation (IR) to machine instructions. Whenever codegen outputs an inadvertent WRPKRU, the surrounding instructions in the IR can be replaced with equivalent WRPKRU-free instructions as described above, and codegen can be run again on the updated IR.

For binaries that cannot be recompiled, the rewrite strategy can be integrated with our binary inspection handler (Section 4.1.5). If the handler discovers an unsafe WRPKRU on an executable page during its scan, it can overwrite the page with 1-byte trap instructions, make it executable, and store the original page in reserve without enabling it for execution. Subsequently, if there is a jump into the executable page, a trap occurs and the trap handler discovers an entry point into the page. It can then disassemble the *reserved page* from that entry point on, rewriting any discovered WRPKRU occurrences, and copy the WRPKRU-free instruction sequences back to the executable page. To prevent other threads from executing partially overwritten instruction sequences, we actually rewrite a fresh copy of the executable page with the WRPKRU-free sequences, and then swap this rewritten copy for the executable page. This technique is transparent to the application, has an overhead proportional to the number of entry points into offending pages (we disassemble from every entry point only once) and maintains the invariant that only safe WRPKRU sequences are executable.

In contrast to rewriting at runtime, a binary can be statically rewritten to remove all inadvertent WRPKRUs. Compared to a compiler or runtime approach, static binary rewriting does not rely on source code availability and does not imposes additional runtime overhead. Its drawback is the dependence on a static rewrite tool which can successfully rewrite a binary. In order to successfully rewrite a binary,

tools like Dyninst [34] require a full disassembly of the binary. Recently Bauman et al. [16] have proposed a static rewrite technique removing this dependency.

Our static rewrite approach, similar to the binary inspection, performs a simple linear scan of the binary to find all those inadvertent occurrences of the 3-byte WRPKRU sequence in executable sections. Next, using any binary rewriting tool, e.g., Dyninst [34], we disassemble the binary to the extent possible, and rewrite instructions to eliminate these inadvertent occurrences. We use the previously described rewriting strategy (see Section 4.2.1 or table 4.1). Occurrences of WRPKRU in parts that we cannot disassemble are handled by the binary inspection and rewriting at runtime as described in the previous Section.

We evaluate the effectiveness of statically rewriting binaries in Section 4.5.1.4.

4.3 Use Cases

ERIM differs from prior work by providing efficient isolation in applications where switches between trusted and untrusted components are very frequent, of the order of 10^5 or 10^6 times a second. We describe three such use-cases here, and show in Section 4.5 that ERIM's overhead is low on all of them.

4.3.1 Isolating cryptographic keys in web servers

Isolating *long-term* SSL keys to protect from web server vulnerabilities such as the Heartbleed bug [90] is well-studied [74, 75]. However, long-term keys are accessed relatively infrequently (only a few times per user session). *Session keys* that are accessed far more frequently (up to 10^6 times a second per core in a high throughput web server like nginx) have not been isolated so far. Isolating sessions keys is also relevant as these keys protect the confidentiality of individual users. No existing technique can isolate *session keys* without significant overhead.

Taking ERIM's efficient isolation into account, an ERIM-protected component can isolate the cryptographic keys and cryptographic methods. This results in a small TCB and attack surface. OpenSSL does not implement isolation within the library, hence we partitioned OpenSSL's low-level crypto library (libcrypto) to isolate the session keys and basic crypto routines, which run as T, from the rest of the web server, which runs as U. The outer layer of OpenSSL provides the high-level SSL/TLS interface, whereas the inner, isolated layer securely stores the cryptographic keys and performs cryptographic operations. When using this ERIM-protected OpenSSL within a server application, a new SSL/TLS session creates a session key within the T. Messages of this session can only be en-/decrypted within the T. This efficiently protects the cryptographic keys of server applications from memory vulnerabilities.

4.3.2 CPI/CPS

Code-pointer integrity (CPI) [72] is a compiler transform that prevents control-flow hijacks by isolating sensitive objects—code pointers and objects that can lead to code pointers—in a *safe region* that cannot be written without bounds checks. CPS is a lighter, less-secure variant of CPI that isolates only code pointers. Switching rates to the safe region can be very high in CPI, of the order of 10^6 switches per second on standard benchmarks. A key question in CPI/CPS is how to isolate the safe region. The original paper uses ASLR on x86-64 for its evaluation. ASLR has almost no runtime overhead, but it is now known to be ineffective for data isolation [113, 60, 39, 49, 94].

We show that ERIM can provide strong isolation for the safe region at low cost. To do this, we override the CPI/CPS-enabled compiler's intrinsic function for writing the sensitive region to use a call gate around an inlined sequence of T code that performs a bounds check before the write. (MemSentry [68] also proposes the use

of MPKs for isolating the safe region, but does not actually build or evaluate this use-case.)

4.3.3 Native libraries in managed runtimes

Applications running on managed runtimes such as a Java or JavaScript VM often rely on third-party *native code* libraries. A relevant security goal is to isolate the managed runtime from bugs and vulnerabilities in the native libraries. ERIM can be used for this purpose by mapping the managed runtime to T and the native library(ies) to U. We test this by isolating a native SQLite plugin from Node.js. SQLite and Node.js are, respectively, a state-of-the-art C database library and a state-of-the-art managed runtime for JavaScript [121, 91].

4.4 Implementation

We have implemented a prototype of ERIM on Linux. The prototype includes a 77 line Linux Security Module (LSM) that intercepts all mmap and mprotect calls to prevent U from mapping pages in executable mode, and prevents U from overriding the binary inspection handler. We also added 26 LoC in kernel hooks needed for this module. Our implementation also includes the ERIM runtime library, which provides a memory allocator over M_T, call gates, the ERIM initialization code, and binary inspection. These comprise 569 LoC.

Separately, we have implemented the rewriting logic to eliminate inadvertent WRPKRU occurrences (about 2250 LoC). While we have not yet integrated the logic into either a compiler or our inspection handler, we have integrated it into a standalone binary rewriting tool that uses Dyninst [34] to disassemble binaries. The binaries used in our evaluation do not have any unsafe WRPKRU occurrences and do not load any libraries at runtime.

Call type	Cost (cycles)
Inlined call (no switch)	5
Direct call (no switch)	8
Indirect call (no switch)	19
Inlined call + switch	60
Direct call + switch	69
Indirect call + switch	99
getpid system call	152
lwC switch [74] (Skylake CPU)	6050

Table 4.2: Cycle counts for basic call and return

4.5 Evaluation

We evaluate ERIM on microbenchmarks and on the three applications mentioned in Section 4.3. We perform our experiments on Dell PowerEdge R640 machines with 16-core MPK-enabled Intel Xeon Gold 6142 2.6GHz CPUs (with Turbo Boost and SpeedStep disabled), 384GB memory, 10Gbps Ethernet links, running Debian 8. For the CPI experiment, we use the Levee prototype v0.2 available from `http://dslab.epfl.ch/proj/cpi/` and Clang v3.3.1 including its CPI compile pass, runtime library extensions and link-time optimization. For the nginx experiment, we use nginx v1.12.1 and OpenSSL v1.1.1 and the ECDHE-RSA-AES128-GCM-SHA256 cipher. For the managed language runtime experiment, we use Node.js v9.11.1 and SQLite v3.22.0. For a comparison base line we use SQLite compiled to WebAssembly via emscripten v1.37.37's WebAssembly backend [36].

4.5.1 Microbenchmarks

4.5.1.1 Switch cost

We performed a microbenchmark to measure the overhead of invoking a function with and without a switch to a trusted component. The function adds a constant to an integer argument and returns the result. Table 4.2 shows the cost of invoking

the function, in cycles, as an inlined function (I), as a directly called function (DC), and as a function called via a function pointer (FP). For reference, the table also includes the cost of a simple syscall (getpid) and the cost of a switch on lwCs, a recent in-process isolation mechanism based on standard page table protections [74].

In our microbenchmark, calls with an ERIM switch are between 55 and 80 cycles more expensive than their no-switch counterparts. The most expensive indirect call costs less than the simplest system call (getpid). ERIM switches are up to 100x faster than lwC switches.

Because the CPU must not reorder loads and stores with respect to a WRPKRU instruction, the overhead of an ERIM switch depends on the CPU pipeline state at the time of the WRPKRUs in the switch. In experiments described later in this Section, we observed average overheads ranging from 11 to 260 cycles per switch. At a clock rate of 2.6GHz, this corresponds to overheads between 0.04% and 1.0% for 100,000 switches per second, which is significantly lower than the overhead of any bounds-check, kernel- or hypervisor-based isolation.

4.5.1.2 Emulating MPK's switch cost

Following we describe how to emulate the WRPKRU instruction. This enables us to compare against techniques who's environment does not support MPK. The WRPKRU instruction moves the value of the eax register to the PKRU register. However, since the instruction impacts the validity of subsequent loads/stores, the instruction cannot be re-ordered relative to surrounding load/store instructions in the execution pipeline. We emulate the cost of WRPKRU using a sequence of xor instructions that have no net functional effect (except consuming CPU cycles), followed by RDTSCP, which causes a pipeline stall and prevents instruction re-ordering. The emulation code is shown in Listing 4.2.

```
for(i = 0; i < 5; i++) {
    xor eax, ecx
    xor ecx, eax
    xor eax, ecx
}
rdtscp
```

Listing 4.2: WRPKRU emulation using RDTSCP and XorSwitch

Benchmark	Switches/sec	CPI Overhead (%)	
		ERIM	EMUL
403.gcc	13,454,647	22.3	22.68
445.gobmk	1,055,994	1.77	1.76
447.dealII	1,270,582	0.56	0.17
450.soplex	408,192	0.6	2.56
464.h264ref	1,684,572	1.22	0.86
471.omnetpp	36,578,718	144.02	142.26
482.sphinx	1,148,883	0.84	0.65
483.xalancbmk	21,448,977	52.22	51.74

Table 4.3: Domain switch rates of selected SPEC CPU benchmarks and overheads for ERIM-CPI and EMUL-CPI, relative to standard CPI.

Validation To validate that our emulation estimates overheads close to those of the actual WRPKRU instruction, we re-run the CPI/CPS benchmarks of Section 4.5.2 with WRPKRU emulation in place of the actual WRPKRU instruction. Figure 4.1 reproduces ERIM's relative overheads on various benchmarks from Figure 4.2 but additionally lists the relative overheads using the WRPKRU emulation (lines EMUL-CPI and EMUL-CPS). Table 4.3 lists the precise overheads for CPI on benchmarks that have high switching rates. As can be seen, the overheads of the emulation are very close to actual ERIM's overheads on all benchmarks.

Note from Table 4.3 that our emulation is not perfect, but quite close to the actual in terms of overhead. Emulating the performance of WRPKRU perfectly is difficult since emulation cannot exactly reproduce the effects of WRPKRU on the execution pipeline. (WRPKRU must prevent the reordering of loads and stores with respect to itself.) Depending on the specific benchmark, our emulation slightly over-

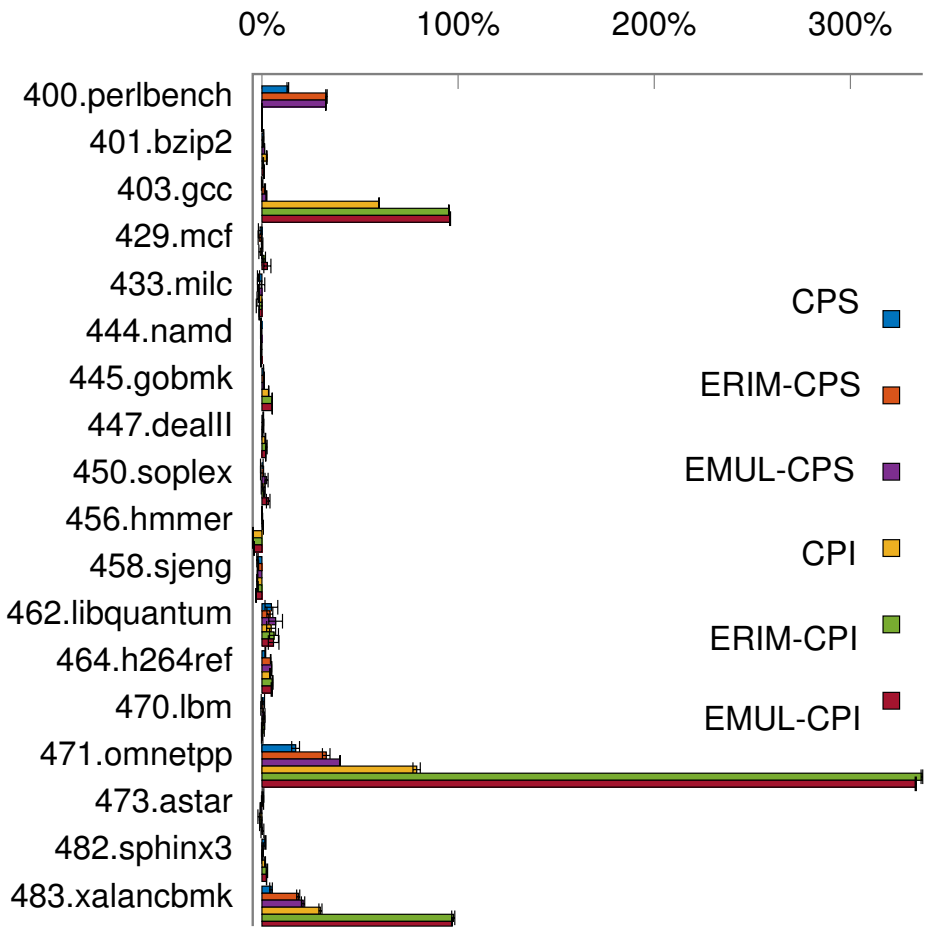


Figure 4.1: SPEC CPU overhead for CPI/CPS with ERIM and an emulation of WRPKRU (EMUL-CPI/EMUL-CPS), relative to no protection. Error bars show standard deviation over 5 runs.

or under-estimates the actual performance impact of WRPKRU due to the reason mentioned above. We also observed that emulations of WRPKRU using LFENCE or MFENCE (the latter was suggested by [68]) in place of RDTSCP incur too little or too much overhead relative to the actual WRPKRU.

4.5.1.3 Binary inspection

To determine the cost of ERIM's binary inspection, we measured the cost of scanning the binaries of each of the 17 applications in the SPEC 2006 CPU benchmark, which

range in size from 9 to 3918 4KB pages and, when compiled with CPI (see next Section), contain between 35 and 63765 WRPKRU instructions. The overhead is largely independent of the number of WRPKRU instructions and ranges between 3.5 and 6.2 microseconds per page. Even for the largest binary, this amounts to only 17.7 milliseconds, a tiny fraction of the typical runtime of a process.

4.5.1.4 Statically rewriting binaries

Here, we analyze how often the WRPKRU opcode sequence (0x0F01EF) occurs in existing binaries and we evaluate the effectiveness of our static binary rewriting.

We analyzed binaries from several Linux distributions to find occurrences of WRPKRU opcodes in executable memory. Note that we have to consider not only code sections of ELF files, but also read-only data marked as executable by the standard GNU linker (PT_LOAD segments with execute-bit).

We analyzed all binaries of the Debian 8, Ubuntu 14.04 and Ubuntu 16.04 repositories as well as a compiled hardened Gentoo. Hardened Gentoo is a source distribution with additional compilation flags (e.g. -PIE for position independent code) to improve security. We also recompiled and relinked a hardened Gentoo with the GNU ELF linker (hardened Gentoo Gold) to generate three load segments (page protections R, RX, RW) in the ELF file. Creating a read-only load segment eliminates exploitable WRPKRU opcodes in static data sections, since they are no longer executable.

Table 4.4 summarizes our findings. First, WRPKRU opcodes occur mainly in Debian- and Ubuntu-based binaries. Almost no WRPKRUs appear in Hardened Gentoo, since it compiles executables as position independent code which changes direct function calls into indirect (rip-dependent) calls reducing the likelihood of a WRPKRU sequence occurring in constant offsets. The hardened Gentoo distribution has 9 occurrences of WRPKRU in executable data segments, which do not appear in Gentoo Gold.

Distribution		Debian 8	Ubuntu 14	Ubuntu 16	Hardened Gentoo	Hardened Gentoo Gold
ELF files		61364	69830	79169	10212	10212
ELF files with WRPKRUs		182 (.30%)	224 (.32%)	219 (.28%)	9 (.09%)	0 (.0%)
Executable WRPKRUs		301	458	273	16	0
WRPKRUs in code section		69 (22.9%)	76 (16.6%)	101 (37.0%)	0	0
Disassembled by Dyninst		58 (84%)	63 (82.9%)	91 (90%)	0	0
Inter-instruction	Number	35 (60%)	37 (59%)	43 (47%)	0	0
	Rewritable by split	35 (100%)	37 (100%)	43 (100%)	0	0
Intra-instruction	Number	23 (40%)	26 (41%)	48 (53%)	0	0
	Rewritable by rule 7	23 (100%)	26 (100%)	48 (100%)	0	0

Table 4.4: Analysis inadvertent WRPKRU opcodes in Linux distributions and ability to statically rewrite

The majority of WRPKRU opcodes are found in non-code sections (like read-only static data) that are executable, due to their inclusion in the read-execute load segment. By linking these binaries with the GNU ELF linker and generating three PT_LOAD segments, we can render these WRPKRU opcodes non-exploitable.

Using the DynInst [34] tool, we attempted to disassemble the code sections (e.g. text, init, fini, plt) that contain WRPKRU opcodes. Of the 69, 76, and 101 such instances in Debian 8, Ubuntu 14, and Ubuntu 16, DynInst was able to successfully disassemble the surrounding code in 84, 82.9, and 90% of cases, respectively.

Next, we divided all occurrences where DynInst was able to disassemble the surrounding instructions into those where the sequence appears either within or across an x86 instruction. Both occur with similar frequency in the set of analyzed binaries. No binary contained a legitimate WRPRKU instruction, since no software uses MPK natively at the time of this writing. All instances that appear across two x86 instructions were eliminated by inserting a NOP instruction. We were able to rewrite all remaining, intra-instruction instances using rule 5 in Table 4.1. Neither of these rewrites affects the performance of the rewritten programs.

The remaining 11, 13, and 10 instances in the three Linux distributions, respectively, could not be rewritten because DynInst was unable to reconstruct the call graph and disassemble the surrounding code successfully. We rely on the binary inspection technique described in Section 4.1.5 to remove these instances.

4.5.2 Protecting sensitive data in CPI/CPS

We use ERIM to isolate the safe region of CPI and CPS [72] in a separate domain. We modified CPI/CPS's LLVM compiler pass to emit additional ERIM switches, which bracket any code that modifies the safe region. The switch code, as well as the instructions modifying the safe region, are inlined with the application code. In addition, we implemented simple optimizations to safely reduce the frequency of

ERIM domain switches. For instance, the original implementation initializes sensitive code pointers to zero during initialization. Rather than generate a domain switch for each pointer initialization, we generate loops of pointer set operations that are bracketed with a single pair of ERIM domain switches. This is safe, because the loop relies on direct jumps and the code to set a pointer is inlined in the loop's body. In total, we modified 300 LoC in LLVM's CPI/CPS pass.

Like the original CPI/CPS paper [72], we compare the overhead of the original and our ERIM-protected CPI/CPS system on the SPEC CPU 2006 benchmarks, relative to a base line compiled with Clang without any protection. The original CPI/CPS system is configured to use ASLR for isolation, the default technique used on x86-64 in the original paper. ASLR imposes almost no switching overhead, but also provides no security [113, 60, 39, 49, 94].

Figure 4.2 shows the average runtime overhead of 10 runs of the original CPI/CPS (lines "CPI/CPS") and CPI/CPS over ERIM (lines "ERIM-CPI/CPS"). All overheads are normalized to the unprotected SPEC benchmark. We were unable to obtain results for 400.perlbench for CPI and 453.povray for both CPS and CPI. The 400.perlbench benchmark does not halt when compiled with CPI and SPEC's result verification for 453.povray fails due to unexpected output. These problems exist in the code generated by the Levee CPI/CPS prototype with CPI/CPS enabled (-fcps/-fcpi), not our modifications.

4.5.2.1 CPI

The geometric means of the overheads (relative to no protection) of the original CPI and ERIM-CPI across all benchmarks are 4.7% and 5.3%, respectively. The relative overheads of ERIM-CPI are low on all individual benchmarks except gcc, omnetpp, and xalancbmk.

To understand this better, we examined the switching rates across benchmarks. Table 4.5 shows the switching rates for benchmarks that require more than 100,000

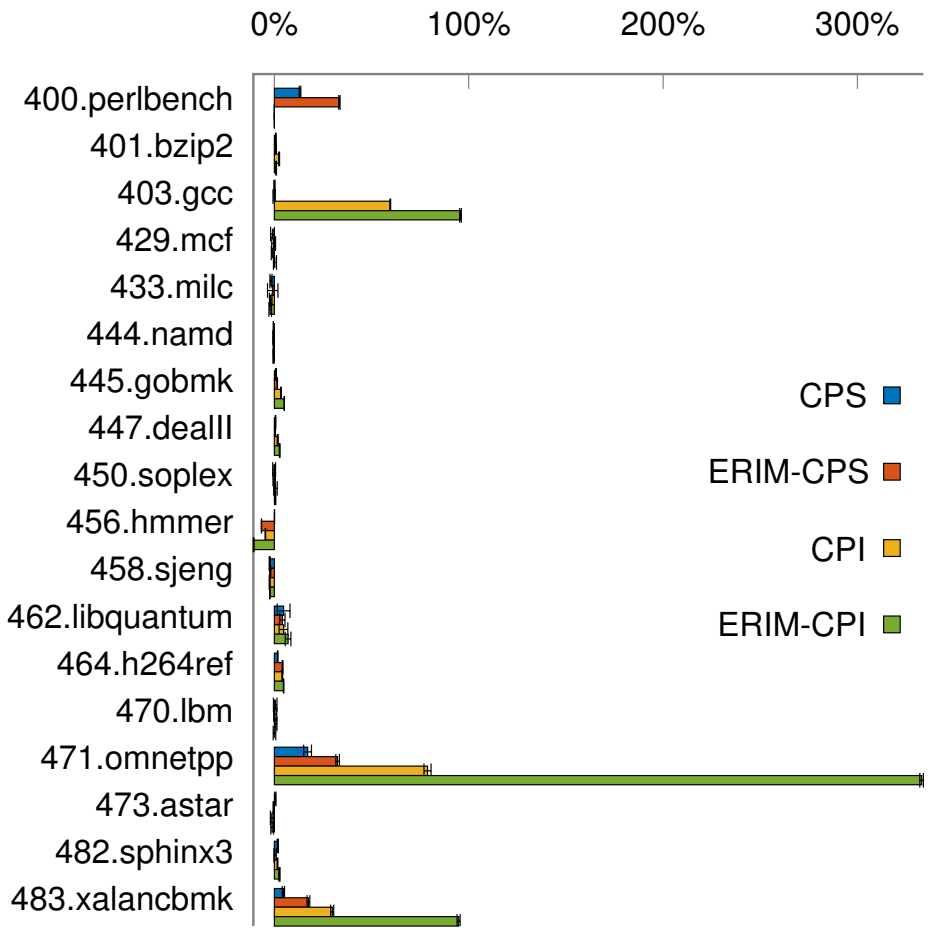


Figure 4.2: SPEC CPU overhead for CPI/CPS and ERIM-CPI/CPS, relative to no protection.

switches/sec. From the table, it is clear that the high overheads on gcc, omnetpp and xalancbmk are due to the extremely high switching rates on these three benchmarks (between 1.6×10^7 and 8.9×10^7 per second). Profiling the execution of these benchmarks indicated that the reason for the high rate of switches is tight loops with pointer updates (each pointer update incurs a switch). An additional optimization pass could lift the domain switches out of the loops safely by using only direct control flow instructions and enforcing store instructions to be bound to the application memory, but we have not yet implemented it.

Benchmark	Switches/sec	ERIM-CPI overhead relative to orig. CPI in %
403.gcc	16,454,595	22.30%
445.gobmk	1,074,716	1.77%
447.dealII	1,277,645	0.56%
450.soplex	410,649	0.60%
464.h264ref	1,705,131	1.22%
471.omnetpp	89,260,024	144.02%
482.sphinx3	1,158,495	0.84%
483.xalancbmk	32,650,497	52.22%

Table 4.5: Domain switch rates of selected SPEC CPU benchmarks and overheads for ERIM-CPI without binary inspection, *relative to the original CPI with ASLR*.

Table 4.5 also shows the overhead of ERIM-CPI *excluding* binary inspection, relative to the *original CPI* over ASLR (not relative to an unprotected baseline as in Figure 4.2). This relative overhead is exactly the cost of ERIM's switching. Depending on the benchmark, it varies from 0.03% to 0.16% for 100,000 switches per second or, equivalently, 7.8 to 41.6 cycles per switch. These results indicate that ERIM can support inlined reference monitors with switching rates of up to 10^6 times a second with low overhead. Beyond this rate, the overhead becomes noticeable.

4.5.2.2 CPS

The results for CPS are similar to those for CPI, but the overheads are generally lower. Relative to vanilla SPEC with no protection, the geometric means of the overheads of the original CPS and ERIM-CPS across all benchmarks are 1.1% and 2.4%, respectively. ERIM-CPS overhead relative to the original CPS is within 2.5% on all benchmarks, except except perlbench, omnetpp and xalancbmk, where it ranges up to 17.9%.

4.5.3 Protecting session keys in nginx

Next, we use ERIM to isolate SSL session keys in a high performance web server, nginx. We modified OpenSSL's libcrypto to isolate the keys and the functions for

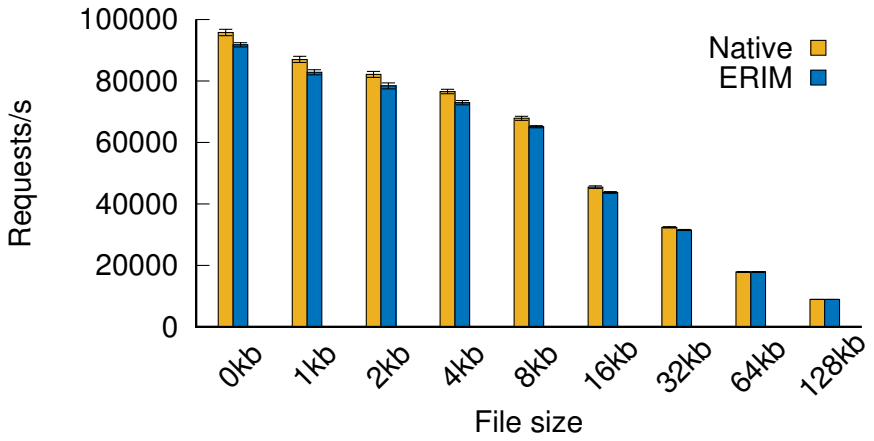
File size (KB)	Throughput		Switches/s	CPU load native (%)
	Native (req/s)	ERIM rel. (%)		
0	95,761	95.83	1,342,605	100.0
1	87,022	95.18	1,220,266	100.0
2	82,137	95.44	1,151,877	100.0
4	76,562	95.25	1,073,843	100.0
8	67,855	95.98	974,780	100.0
16	45,483	97.10	820,534	100.0
32	32,381	97.31	779,141	100.0
64	17,827	100.00	679,371	96.7
128	8,937	99.99	556,152	86.4

Table 4.6: Nginx throughput with a single worker. The standard deviation is below 1.1% in all cases.

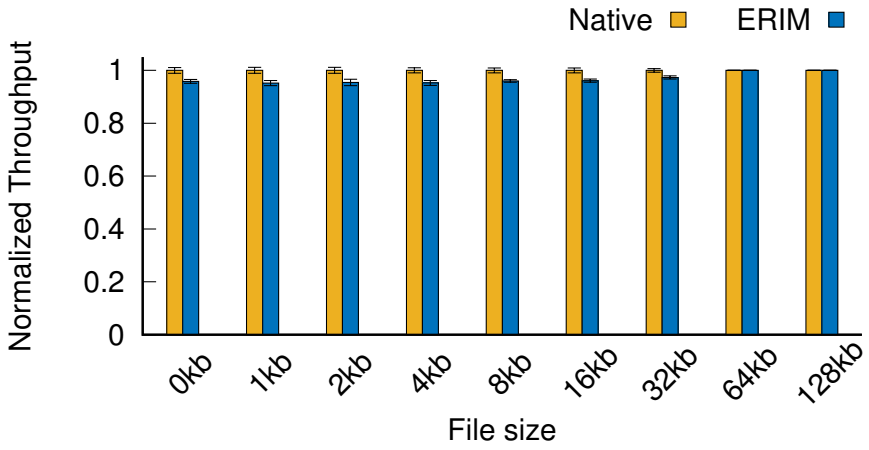
AES key allocation and encryption/decryption into ERIM's T and use ERIM call gates to invoke these functions.

Our goal is to measure ERIM's overhead on the peak throughput of nginx. To start, we configure nginx to run a single worker pinned to a CPU core, and connect to it remotely from 4 concurrent ApacheBench (`ab`) [8] instances over HTTPS with keep-alive. Each instance simulates 75 concurrent clients. The clients all request the same file, whose size we vary from 0 to 128KB across experiments. Figure 4.3b shows the throughput of ERIM-protected nginx relative to our baseline (native nginx without any protection) for different request sizes, measured after an initial warm-up period. Figure 4.3a shows the absolute throughputs in requests/s in the same experiment. All numbers are averages of 10 runs.

ERIM-protected nginx provides a throughput within 95.18% of the unprotected server for all request sizes. To explain the overhead further, we list the number of ERIM switches per second in the nginx worker and the worker's CPU utilization in Table 4.6 for request sizes up to 128KB. The overhead shows a general trend up to requests of size 32 KB: The worker's core remains saturated but as the request size increases, the number of ERIM switches per second decrease, and so does



(a) Average number of requests per second for native and ERIM.



(b) Normalized throughput to native (no protection).

Figure 4.3: Nginx throughput with one worker, with and without ERIM protection, with varying request sizes. Standard deviations were all below 1.1%.

ERIM's relative overhead. The observations are consistent with an overhead of about 0.31%–0.44% for 100,000 switches per second. For request sizes of 64KB and higher, the 10Gbps network card saturates and the worker does not utilize its CPU core completely in the baseline. The free CPU cycles absorb ERIM's CPU overhead, so ERIM's throughput matches that of the baseline.

Note that this is an extreme test case for a web server. Here, the web server does almost nothing and serves the same cached file repeatedly. To get a more realistic assessment, we set up nginx to serve from a 571 MB corpus of 15,520 static HTML Wikipedia pages snapshotted in 2006 [137]. File sizes vary from 417 bytes to 522 KB (average size 37.7 KB). 75 keep-alive clients request random pages (selected based on pageviews on Wikipedia [138]). The average throughput with a single nginx worker was 22,415 requests/s in the base line and 21,802 requests/s with ERIM (std. devs. below 0.6% in both cases). On average, there were 615,000 switches a second. This corresponds to a total overhead of 2.7%, or about 0.43% for 100,000 switches a second.

4.5.3.1 Scaling with multiple workers

To verify that ERIM scales with core parallelism, we re-ran the first experiment above with 3, 5 and 10 nginx workers pinned to separate cores, and sufficient numbers of concurrent clients to saturate all the workers. Table 4.7 shows the relative overheads with different number of workers. For requests larger than those shown in the table, the network card saturates, and the spare CPU cycles in the native base line absorb ERIM's overhead completely. For comparison, the second and third columns of the table repeat the numbers of the 1 worker configuration of Table 4.6.

In the baseline, nginx's throughput scales quite well with the number of workers. Importantly, the relative overhead of ERIM's protection does not increase with the number of cores. Thus, ERIM scales with multi-core parallelism indicating that ERIM adds no additional synchronization and scales perfectly with core parallelism.

File size (KB)	1 worker		3 workers		5 workers		10 workers	
	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)	Native (req/s)	ERIM rel. (%)
0	95,761	95.83	276,736	96.05	466,419	95.67	823,471	96.40
1	87,022	95.18	250,565	94.50	421,656	96.08	746,278	95.47
2	82,137	95.44	235,820	95.12	388,926	96.60	497,778	100.00
4	76,562	95.25	217,602	94.91	263,719	100.00		
8	67,855	95.98	142,680	100.00				

Table 4.7: Nginx throughput with multiple workers. The standard deviation is below 1.5% in all cases.

This is unsurprising given that updates to the PKRU of a core affect execution on that core only.

4.5.3.2 Comparison to kernel-based isolation

Using the single worker nginx experiment, we compare ERIM's overhead to that of lwCs [74], a state-of-the-art system for in-process isolation based on *standard* page-table protections. LwCs map each isolated component to a separate address space (in the same process). A switch between components requires kernel mediation to change page tables.

Since lwCs were implemented only for FreeBSD, whose current kernel supports neither MPKs nor our Xeon Gold machines, we run this comparison experiment on an older machine without MPK support and use an emulation of WRPKRU to account for ERIM's overhead as described in Section 4.5.1.2. All experiments described here were performed on Dell OptiPlex 7040 machines with 4-core Intel Skylake i5-6500 CPUs clocked at 3.2GHz, 16GB memory, 10 Gbps Ethernet cards, running FreeBSD 11.

We use the existing single worker nginx experiment (Section 4.5.2) to compare the performance of an ERIM-based isolation to that of lwC-based isolation. As opposed to ERIM switches, which operate entirely in userspace, lwC switches are syscalls. We create a second instance of nginx that uses an lwC (in place of an ERIM

component) to isolate session keys and basic cryptographic functions. We allocate data buffers in a memory region that is shared between the protected lwC and the web server lwC to facilitate efficient data sharing. The overhead of WRPKRU is emulated as previously explained.

Figure 4.4b depicts the overhead of the ERIM- and lwC-based variants relative to the native baseline of an unmodified nginx, averaged over 20 runs. Nginx is configured to run one worker and serves 4 ApacheBench instances each simulating 75 clients accessing a static file via HTTPS with keep-alive.

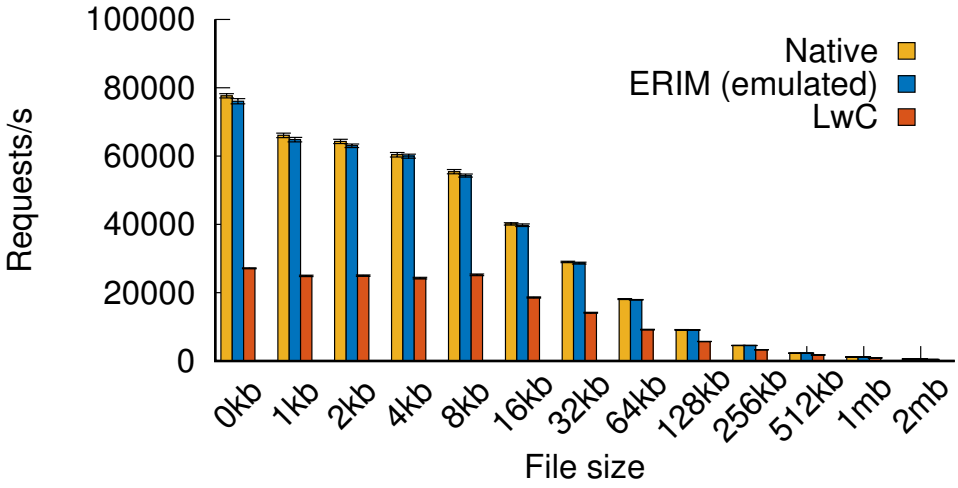
The ERIM-based emulation provides throughput within 97.88% (within 99% for files 64KB and larger) of the unprotected native server, whereas the lwC-based isolation is limited to 50% of the native server throughput for small files and up to 80% for large (2MB) files. The reason is the cost of lwC switch syscalls, which is too high given the rate of invocations of the encryption functions.

Figure 4.4a shows the absolute number of served requests per second for the same experiment. The lwC-based nginx cannot sustain more than 26,500 req/s, whereas ERIM performs close to the native implementation. At 64KB files we saturate the 10Gbit network link resulting in lower req/s for the native baseline and ERIM.

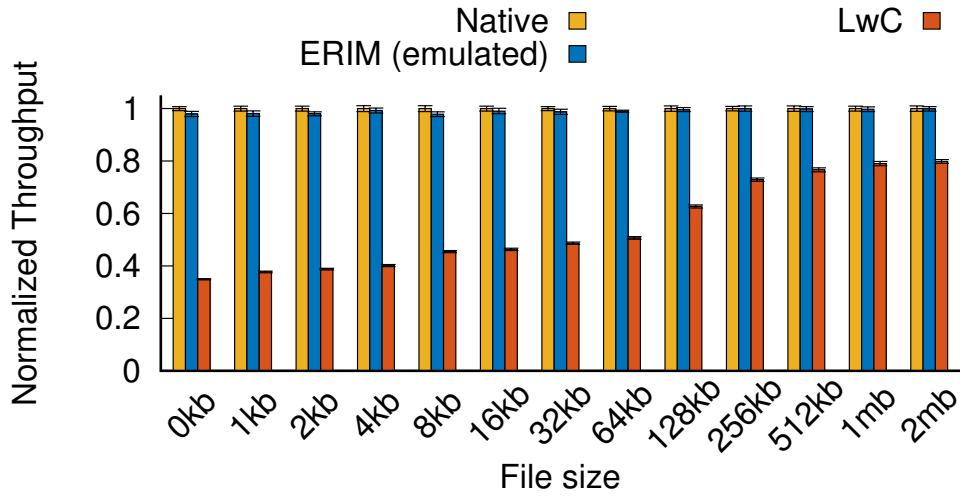
In summary, we find that lwCs perform significantly worse than ERIM in this experiment: The throughput of nginx with lwC-based isolation is never above 80% of native nginx and, for small requests, where the switch rate is higher, it is below 50% of native nginx. In contrast, with ERIM's isolation, the throughput is within 95% of native nginx in all configurations. Hence, ERIM performs significantly better than kernel-mediated isolation at high switch rates.

4.5.4 Isolating managed runtimes

Next, we test ERIM's use to isolate a managed language runtime from an untrusted native library. Specifically, we link the widely-used native database library, SQLite,



(a) Average number of requests per second, native, ERIM and lwC.



(b) ERIM and lwC throughput normalized to native.

Figure 4.4: Nginx throughput with one worker, with emulated ERIM protection and with lwCs, with varying request sizes. Standard deviations were all below 1.1%.

to Node.js, a state-of-the-art JavaScript runtime and use ERIM to isolate Node.js from SQLite by mapping Node.js's runtime to T and the native library to U. We manually instrumented SQLite's entrypoints to invoke call gates. Additionally, since we want to isolate Node.js's stack from SQLite, we run Node.js on a separate stack in M_T, and add code to switch to the standard stack (in M_U) prior to calling a SQLite function. Finally, SQLite uses the libc function `memmove`, which accesses libc constants that are in M_T, so we implemented a separate `memmove` for SQLite. In total, we added 437 LoC.

We measure run time using the speedtest1 benchmark that comes with SQLite and emulates a typical database workload [122]. This benchmark performs a total of 32 short tests that stress different database functions like selects, joins, inserts and deletes. We increased the iterations in each test by a factor of four to make the tests longer. Our base line for comparison is vanilla SQLite linked to Node.js without any protection. We configure the benchmark to store the database in-memory and report averages of 20 runs.

The geometric mean of ERIM's overhead on runtime across all tests is 4.3%. The overhead is below 6.7% on all tests except those with more than 10^6 switches per second. This suggests that ERIM can be used for isolating native libraries from managed language runtimes with low overheads up to a switching cost of the order of 10^6 per second. Beyond that the overhead is noticeable. Table 4.8, columns 1–3, show the relative overheads for tests with switching rates of at least 100,000/s. These are consistent with an average overhead between 0.07% and 0.41% for 100,000 switches/s. The actual switch cost measured from direct CPU cycle counts varies from 73 to 260 cycles across all tests. The switch cost exceeds 100 cycles only on benchmarks where the switch rate is very low (less than 2,000 times/s). We verified that these higher cycle counts are due to instruction cache misses—at very low switch rates, the call gate is flushed out of the instruction cache between switches.

Test #	Switches/s	Overhead (%)	
		ERIM	WebAssembly
100	11,183,281	12.73%	132.48%
110	8,329,914	12.18%	135.44%
400	8,161,584	15.42%	156.04%
120	7,190,766	13.81%	145.19%
142	7,074,553	9.41%	165.88%
500	6,419,008	12.13%	119.15%
510	5,868,395	5.60%	113.76%
410	5,091,212	3.64%	122.77%
240	2,358,524	3.74%	126.63%
280	2,303,516	3.22%	100.05%
170	1,264,366	4.22%	104.87%
310	1,133,364	2.92%	81.71%
161	1,019,138	2.81%	138.64%
160	1,014,829	2.73%	136.27%
230	670,196	2.04%	193.42%
270	560,257	2.28%	92.78%

Table 4.8: Overhead relative to native execution for SQLite speedtest1 tests with more than 100,000 switches/s. Standard deviations were below 5.6% for native, and ERIM and below 15.4% for WebAssembly.

4.5.4.1 Comparison to isolation with bounds checks (SFI)

We also use the above experiment to compare ERIM to isolation based on bounds checks. For this, we re-compile the SQLite library to native code indirectly through WebAssembly, a new memory-safe, low-level language designed specifically for writing safe native plugins for JavaScript environments [53]. The WebAssembly to native code translation inserts bounds checks prior to indirect memory accesses. Compilation via WebAssembly is the currently recommended method for safely adding native plugins to Google's Chrome web browser; most major web browsers are expected to recommend the same method in the near future.

Across all 32 tests, the geometric mean of the relative overhead of WebAssembly-based isolation on run time is 133.5%. The overheads range from 66.4% to 280.6%, which is significantly higher than ERIM's overheads. However, WebAssembly's overheads do not increase with the switching rate since it does not interpose on

switches. Instead, it imposes a continuous overhead while execution is in SQLite. Other work using bounds checks has found similarly high overheads on performance-intensive benchmarks [95, 53].

4.6 Related Work

Reference monitors [6] mediate privileged access by untrusted applications protecting data confidentiality and integrity, by sandboxing the application or checks inserted into the applications execution. All implementations share the important property of protecting the reference monitor's code and state from corruption and deploy isolation techniques shielding the reference monitor. Least-privilege and privilege separation [105] define the basis for today's reference monitor mechanisms in hardware ([82, 63, 12]), hypervisors ([14, 19]), operating systems ([132, 35]) or applications ([129, 80, 41, 143, 29]). Two recent surveys [126, 117] show viable attacks and possible counter measures to protect against data confidentiality and integrity violations. Furthermore their interception granularity varies from a separate guest OS ([14]), a single application ([19, 20, 132]) or application components ([129, 82, 74, 80, 41, 143, 29, 144]). Intercepting at fine granularity offers isolation across application components, whereas course-grain interception isolates independent applications or components. Due to its frequent invocations, fine-grained interception solutions require isolation techniques with low overhead. ERIM isolates application components and intercepts fine-grained security relevant events within the application, similarly to ARMlock [144] or SFI-based isolation [129, 80, 41, 143]. This is in contrast to techniques using OS process boundaries ([74]) or CPU privilege levels ([19, 14]).

Koning et al. [68] survey techniques for efficient data encapsulation within a process, including SFI, dynamic encryption of private data using the Intel AES-NI ISA extensions, approaches that use VT-x virtualization hardware, and those that rely on the Intel MPX and MPK ISA extensions. It then presents a general isolation

technique, called MemSentry, which instruments programs using an LLVM pass. The instrumentation can rely on any one of the above mentioned isolation techniques to ensure that only legitimate accesses to encapsulated data are allowed. One of the isolation techniques used by MemSentry is MPK, and experimental results show that this technique is the most efficient in situations where encapsulated data is located in a few contiguous regions and accesses are frequent.

MemSentry relies on (and assumes the existence of) a general defense against control flow hijacks to prevent untrusted code from exploiting the WRPKRU instruction to raise its privileges. General defenses against control flow hijacks, however, are either incomplete or inefficient. For instance, instrumenting every load and store operation is hard to circumvent but expensive. A defense such as protecting only return addresses by using a shadow stack, on the other hand, can be done efficiently, but does not prevent corruption of other code pointers or indirect branches. Moreover, MemSentry burdens each application that requires data encapsulation with the overhead of a general defense against control flow hijacks, even if it is not otherwise required.

ERIM also uses MPK as the underlying isolation mechanism, but does not rely on a generic defense against control flow hijacks. Instead, it provides a specific defense against abuse of the WRPKRU instruction, a much simpler problem that can be solved securely and efficiently. The performance gains of this change in approach are significant—ERIM imposes less than 16% overhead on the SQLite speedtest (see Section 4.5.4) while isolating via standard WebAssembly increases the runtime by a geometric mean of 133.5%. WebAssembly similarly relies on control-flow integrity for isolation.

In the following we survey techniques to isolate memory and reference monitor applications using hardware-based TEE, hypervisor-/OS-based, language, compiler and runtime techniques.

Hardware-based trusted execution environments. Recent additions to Intel's [63] and ARM's [12] ISA allow applications to execute a trusted part in a trusted execution environments (TEE). Isolated from the remaining hypervisor, operating system or applications, TEE's provide strong isolation and require targeted implementations. Research systems use TEEs to reduce the TCB (see Flicker [82]) needed to execute code securely. Due to their strong isolation, TEE's do not allow direct access to OS services like filesystem or network which limits the usage due to overheads while crossing the isolation boundary. ERIM isolates application components, similar to Flicker, while maintaining the programming model allowing access to OS services.

Haven [17] and SCONE [13] provide a systematic approach to run entire applications in TEEs while providing OS services. This shields entire applications from their environment instead of protecting a specific application part. Security vulnerabilities existing in the unshielded application, also exist in the shielded execution which attacks may use to violate memory safety. As a result research is already looking at using memory safety techniques [71] in TEEs.

In addition to TEEs Intel also added support for bound checks using Intel MPX and memory regions with MPK [64]. Both techniques have been studied for their effectiveness and adaptability in [68, 95]. Originally designed to enforce memory bounds on application's data structure accesses, MPX efficiently isolates memory, when dividing into few regions. Once the program uses more regions, the internal bound registers need to access shadow copies from main memory and stall executions. In addition MPX currently allows only single-threaded applications.

Hypervisor/OS-based reference monitors. Today's processors provide several protection rings to isolate hypervisors, OSes and application. Invented to provide efficient virtualization of the underlying hardware, e.g., Xen [14], Wedge [20] and Dune [19] isolate application components to run at different ring levels and monitor OS resources.

Nexen [115] decomposes the Xen hypervisor into isolated components and a security monitor, using page-based protection within the hypervisor's ring-0 privilege level. Control of the MMU is restricted to the monitor; compartments are de-privileged by scanning and removing exploitable MMU-modifying instructions. By relying on MPK, ERIM is able to use a similar approach within application processes at ring-3, by scanning and removing exploitable WRPKRU instructions.

SIM [114] relies on VT-x to isolate a security monitor within an untrusted guest VM, where it can access guest memory with native speed. In addition to the overhead of the VMFUNC calls during switching, these techniques incur overheads on TLB misses and syscalls due to the use of extended page tables and hypercalls, respectively. Overall, the overheads of virtualization-based encapsulation are comparable to OS-based techniques.

Novel kernel abstractions like light-weight contexts [74] or secure memory views [59] have reduced the cost of data encapsulation to the point where isolating OpenSSL *long-term* signing keys is feasible with little overhead [74]. ERIM achieves higher switch rates which allow isolating not only the *long-term* signing keys, but also the *short-term* session keys.

Systems to automatically divide applications into different privilege levels use OS process boundaries as protection [22, 67]. Their utility is limited by the switching overhead across processes to monitoring infrequent events like file descriptor creations (open, fopen). In ERIM the monitor similarly controls OS service access to untrusted applications, while reducing the switch overhead between monitor and application.

Language and runtime techniques Memory isolation may be implicit in a memory-safe programming language. The provided isolation depends on the correctness of the compiler and the language runtime. However, such isolation can be easily undermined by native libraries written in memory-unsafe languages.

Wahbe et al. [129] introduce software-fault isolation (SFI), a technique to isolate untrusted applications and control access to OS services. Subsequently, additional

effort [144, 80, 41, 143] improved the guarantees, threat model, and performance. SFI isolates by restricting the binary code during compilation, by binary rewriting, or by dynamically emulating instructions. Instead of relying on a compiler, binary rewriter, or emulator, ERIM trusts the Intel CPU to enforce memory isolation.

Recent systems [7] adapt SFI to just-in-time (jit) compiled languages like JavaScript and isolate the language runtime from native libraries by generating binary code restricting memory accesses. Similarly, ERIM isolates native libraries from the language runtime as described in Section 4.3.3. Instead of restricting all memory accesses, an ERIM-protected code generator is only required to generate WRPKRU free binary code which results in less overhead during code execution and code generation.

In contrast to SFI, which isolates untrusted applications, inlined reference monitors (IRM) [38] insert security checks into the untrusted application to isolate it from the trusted component. Using a compiler pass, checks are inserted before, e.g., memory accesses, jumps or function returns to protect the execution from accessing protected memory. Two well known examples are control-flow integrity (CFI) [1] or code-pointer integrity (CPI) [72]. To protect the confidentiality of its state (pointer tables), CPI relies on address space layout randomization (ASLR) as one possible isolation mechanism. However, ASLR is known to be easily breakable [60, 39, 94, 49]. ERIM offers an alternate, efficient technique for isolation in mechanisms like CPI.

Shreds [29] monitors and protects security relevant variables within applications. It requires annotations by the programmer and a compiler pass to insert control switches. At the beginning of the application a memory region is created to store the isolated variables. When the execution accesses a variable, the execution switches to trusted code and access the isolated variable. Compared to ERIM Shred's switches are very costly.

IMIX [43] introduces an additional page protection type called IMIX which can only be accessed using a new smov CPU instruction. In combination with a

compiler supporting automated partitioning of sensitive data, IMIX compiles source code which allocate sensitive data in an IMIX protected page. IMIX requires a complementary control-flow integrity technique to be present and disallows dynamic loading of code. ERIM, on the other hand, guarantees isolation despite control-flow hijacks without relying on a technique for control-flow integrity.

Aurasium [141] dynamically relinks Android framework invocations and user data accesses to an in-process reference monitor. In contrast to ERIM, Aurasium does not protect the reference monitor's memory from accesses. Hence, malicious code may call Android framework functions or access user data directly.

Memory hiding. Memory Hiding relies on ASLR to place secrets at random locations and restricts pointer computations [113, 58]. Although a very efficient memory isolation solution, several attacks [60, 39, 94, 49] demonstrate how to subvert the probabilistic guarantees and find hidden secrets. ERIM does not rely on randomization, but still has comparable performance.

In addition to hiding secrets at random locations, secrets can also be hidden in execute-only pages which include the secret as immediate values in CPU instructions such as load immediate. A computation using the secret first loads the secret into a register, computes the function and destroys the contents of the register before returning to the regular execution. Redactor [31] and NEAR [135] hide code pointers in execute-only memory to prevent code disclosure which is used by return-oriented programming attacks.

4.7 Conclusion

We conclude with a brief summary of ERIM and how it compares to other memory isolation techniques. Relying on the recent Intel MPK ISA extension and simple binary inspection, ERIM provides hardware-enforced isolation with an overhead of

less than 1% for every 100,000 switches/s between components on current CPUs. It imposes no overhead on execution within a component.

Existing hardware isolation techniques rely on either kernel- or hypervisor-mediation for switching protection domains and incur much higher switching costs—the state-of-the-art lwCs impose approximately 10% overhead for every 100,000 switches/s. Other techniques based on access bounds-checks such as SFI or memory-safe languages provide isolation without interposing on domain switches, but impose costs of several tens of percent on the normal execution of untrusted code, even with mainstream hardware support for bounds checking (e.g., MPX), and additionally require control-flow integrity to provide strong security. Yet other software techniques like ASLR impose negligible runtime overhead but offer very limited defense against strong user-space adversaries.

ERIM's comparative advantage prominently stands out on applications that switch very rapidly, and that spend a nontrivial fraction of time in untrusted code. We have demonstrated ERIM's efficacy on three such applications: isolating session keys in web servers, isolating an inlined reference monitor's private state and isolating managed runtimes from native libraries. In all cases, ERIM provides strong isolation with overheads significantly lower than those of existing techniques.

CHAPTER 5

Conclusion

Today computers store and analyze valuable and sensitive data such as personal multimedia or client records. An important goal is to protect the confidentiality and integrity of such data, minimizing the risk of illicit release, loss or modification. However, existing techniques to protect confidentiality and integrity are vulnerable to malicious attacks or are inefficient. This thesis contributes two new techniques, Guardat and ERIM, providing confidentiality and integrity for *persistent* and *in-memory* data securely and efficiently.

Guardat enforces, at the storage layer, rich per-file confidentiality and integrity policies with low overhead. The enforcement at the storage layer reduces the attack surface and the risk of circumvention due to software bugs, misconfigurations and operator errors in higher layers. Guardat overcomes the gap between storage layer enforcement and per-file policies by attesting the state of files and associated policies through cryptographically-signed certificates. To specify policies, we develop a domain-specific language which allows data accesses conditioned on authentication, trusted wall clock time, and a file's state including the content. We demonstrate an efficient implementation to enforce such policies in an iSCSI SAN server and apply Guardat to two use cases protecting the content, executable, and log files of a web server, as well as enforcing mandatory access logging.

ERIM provides data confidentiality and integrity for *in-memory* data by isolating sensitive data from accesses by untrusted components. It isolates sensitive data

into a separate, trusted memory component using Intel MPK and ensures that only the trusted component has access to sensitive data. To prevent malicious attacks from escalating privileges using the unprivileged WRPKRU CPU instruction, ERIM additionally protects the trusted component via secure control transfers and binary inspection. Secure control transfers ensure that the untrusted component cannot elevate access permissions without the involvement of the trusted component. Binary inspection guarantees that no executable binary code sequence elevates access permissions to the trusted component, while executing untrusted code. ERIM's isolation imposes no additional overhead on the execution and less than 1% runtime overhead per 100,000 switches/second. Unlike state-of-the-art isolation techniques, the low switch cost and no overhead on execution allows ERIM to efficiently isolate frequently-used session keys in web servers, an in-memory reference monitor's private state, and managed runtimes from native libraries as demonstrated in the evaluation.

5.1 Future Work

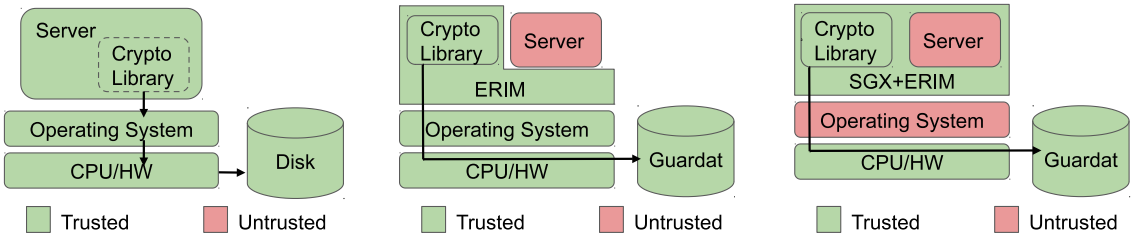
Guardat and ERIM independently protect the confidentiality and integrity of sensitive *persistent* and *in-memory* data. While Guardat restricts data accesses at the storage layer, it does not protect data released to an application. In contrast ERIM restricts access to application data, but does not protect persistent data from malicious attacks. Although beyond the goal of this thesis, in this section we discuss how to overcome the limitations of each technique and attain an end-to-end confidentiality and integrity guaranty of sensitive *persistent* data throughout its *in-memory* use in an application.

We can overcome the limitations of each individual system by connecting an ERIM-isolated monitor to a Guardat device which stores the secrets. These persistently stored secrets on a Guardat device could be protected from arbitrary data accesses by associating a policy which allows data access only by authorized connec-

tions. However, without any changes to ERIM, an ERIM-isolated monitor would not be able to access these secrets. To gain access, the monitor has to connect to the Guardat device and authenticate itself. The current design of ERIM does not provide a way to generate a unique authorization secret that is reliable and consistent across environments and reboots. In order to generate authorization secrets, existing techniques like trusted platform modules (TPM) or Intel SGX generate authorization secrets for code by measuring the code's in-memory footprint as a secure hash. This hash is then used to derive a unique authorization secret.

Similar to existing techniques, we suggest to change ERIM's initialization to generate an authorization secret by measuring the trusted monitor's footprint. Once measured, the secret is placed in the trusted monitor's memory (outside of the untrusted application's reach). Using this secret, the trusted monitor authenticates itself to the Guardat device. This approach guarantees that Guardat only releases secrets to an ERIM-isolated monitor, while ERIM protects the in-memory copy of the secret from accesses by an untrusted application.

By connecting an ERIM-isolated monitor to Guardat we provide an end-to-end confidentiality and integrity guarantee which would be particularly interesting for server applications that rely on a secure connection to clients using asymmetric cryptography. In today's server and cryptographic library implementations (see Figure 5.1a) the private and session keys are not isolated in memory or protected in persistent storage. Existing server applications read a private key from persistent storage into memory and use the key to establish a secure connection by negotiating a session key. The session key is stored in memory and used by both parties to encrypt and decrypt messages. Hence, security vulnerabilities and bugs in the untrusted server application, the operating system, or applications with access to the storage may result in a confidentiality or integrity violation. For example, unauthorized applications may read the persistent private key and release them, violating confidentiality. Similarly, the key could be modified on disk, violating



(a) Today's server applications

(b) Isolating a cryptographic library using ERIM and connecting it to a Guardat device

(c) Combining ERIMwith Software Guard Extensions (SGX) to isolate from the operating system

Figure 5.1: Steps towards an isolated cryptographic library in server applications

integrity. Once the keys reside in memory, malicious attacks like Heartbleed [90] can release or modify the keys, violating both confidentiality and integrity.

To protect the keys from these types of threats, Guardat in combination with ERIM can protect the persistent keys and the in-memory keys efficiently as shown in Figure 5.1b. The server needs to be split into a trusted monitor which only holds the cryptographic functions and an untrusted server which handles the communication and provides the service to the client. During initialization ERIM isolates the trusted monitor's memory from the untrusted server application. ERIM measures the trusted monitor and provides the authorization secret to the trusted monitor. It then allows the trusted monitor to initialize, connect to a Guardat device, authorize using the previously measured secret, and read the private key. Guardat checks that the authenticated client actually is the ERIM-isolated trusted monitor which is specified in the policy. After finishing the trusted monitor's initialization, ERIM starts the untrusted server, which begins its usual operation waiting for clients to connect. Once a client connects to establish a secure connection, the untrusted server accepts the connection, starts the SSL/TLS handshake protocol and switches to the trusted monitor whenever encrypting or decrypting messages using the in-memory private key and generating new session keys.

In contrast to the threat model of Guardat which assumes intermediate layers, such as the OS, to be vulnerable to malicious attacks or circumvention, the technique proposed above, however, assumes the OS to be trusted, since ERIM's guarantees depend on the OS. As a result, the proposed solution would only protect confidentiality and integrity against attacks from outside, e.g., malicious clients attacking the server like Heartbleed [90], and any threats on the network between the machine running the server and the Guardat device. Such a threat model is common for servers running in the cloud.

Further hardening ERIM against OS vulnerabilities: While the cloud threat model is commonly assumed, recent attacks [24] show how other cloud tenants can access in-memory secrets violating confidentiality and integrity. In addition, highly sensitive applications may not assume the cloud provider to be trusted and, hence, intermediate layers like the OS and VMM which provide isolation are no longer trusted. In order to strengthen the threat model of the presented technique, the memory isolation guarantees of ERIM have to be independent of intermediate software layers like the OS or VMM.

To defend against these threats, trusted execution environments (TEE), in particular Intel SGX, can be used to shield sensitive data from the cloud platform and other tenants. SGX provides in-memory enclaves to store sensitive data and execute code independent of the running OS or VMM. Several research systems [17, 13, 70] demonstrate the use of Intel SGX to shield an application against the cloud platform.

While SGX provides strong memory isolation guarantees, its high switch costs comparable to a context switch hinders its adoption, and prevents it from being used to isolate frequently-used secrets. Existing work overcomes these performance limitations by either isolating infrequently-used secrets like private keys or isolating an entire application. However, pushing entire applications (e.g., a server) into

an enclave without further memory isolation leaves the application vulnerable to malicious attacks, due to the size and complexity of these applications. Recent work [71] suggests further protecting applications in SGX enclaves by adding memory bound checks. However, such checks incur substantial runtime overhead, and it is not sufficient for ERIM to simply swap Intel MPK for Intel SGX. Instead, we need to combine both approaches to allow entire applications to run within SGX enclaves, shielding them from the remaining cloud software stack, while isolating secrets within the application using a mechanism similar to Intel MPK.

To this end, we suggest amending the SGX specification, since it has no provision for MPK-like memory isolation using per-page domains and an access permission register such as the PKRU register. The enclave memory descriptors reside in processor reserved memory which is inaccessible to system software (e.g., OS or VMM). An important descriptor is the enclave page cache map (EPCM), a table-like structure, which holds information about which memory pages belongs to an enclave and holds per-page access permission bits. Currently the EPCM allows pages to be accessible with read, write, and execute permission and does not allow pages to be tagged with a MPK domain.

To allow page-level memory isolation within SGX enclaves, we suggest adding memory domain identifiers to the EPCM and extending the CPU's memory access permission check to validate the current access permissions in the PKRU register against the memory access's EPCM domain identifier. This approach extends the reach of Intel MPK into SGX enclaves. Similar to the use of MPK in ERIM, this solution is vulnerable to malicious attacks and hence needs to be combined with ERIM's secure control transfers and binary inspection. For code in enclaves, we can simplify ERIM's binary inspection, since enclave memory is allocated once at the start of an enclave and can only be extended with a special protocol including a step in which the enclave approves the extension [83]. The binary inspection could scan once at the start for unsafe WRPKRUs in all executable memory and at runtime it

could only approve new pages which do not contain unsafe WRPKRUs. There is no need for kernel modifications or signal handlers. By amending the SGX specification and combining it with ERIM's secure control transfers and binary inspection, we can isolate frequently-used secrets within SGX enclave without trusting the system software like the OS (see Figure 5.1c).

In this section we have shown how to extend the protection of persistent files from Guardat to in-memory data using ERIM. We discussed the challenge in authenticating an ERIM-isolated trusted monitor to a Guardat device and describe a technique to generate an authentication secret using code measurements. We discuss its use in a commonly assumed threat model for cloud environments. For highly sensitive applications, we describe an extension of Intel's SGX to protect against rogue cloud providers and other cloud tenants.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2005.

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

[3] 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 USENIX Conference of File and Storage Technologies (FAST)*, 2003.

[4] Hussain M. J. Almohri and David Evans. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[5] Amazon. Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`, 2011.

[6] James P. Anderson. Computer Security Technology Planning Study (Volume II), 1972.

[7] Jason Ansel, Petr Marchenko, Ulfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2011.

[8] Apache HTTP Server Project. https://httpd.apache.org/docs/2.4/programs/ab.html.

[9] Apple Inc. Time Machine, 2007.

[10] Apple Inc. Apple Fusion Drive. https://www.apple.com/de/imac, 2017.

[11] ARM. Developer guide: ARM memory domains. http://infocenter.arm.com/help/, 2001.

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

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Mark L. Stillwell, David Goltzsche, David Eyers, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[15] Fritzgerald Barth, Victor Chin, Moshe Ferber, Sean Hittel, Laurie Jameson, Natheniel Mason, Hardeep Mehrotara, Ashish Mehta, Mihir Mohanty, Krishna Narayanswamy, and Michael Roza. Top threats to cloud computing plus industry insights. https://www.couldsecurityaliance.org/download/top-threats-to-cloud-computing-plus-industry-insights/, 2017.

[16] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[17] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Transactions on Computer Systems*, 33(3), 2015.

[18] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and Semantics of a Decentralized Authorization Language. In *Proceedings of IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[19] Adam Belay, Andrea Bittau, and Ali Mashtizadeh. Dune: safe user-level access to privileged cpu features. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[20] Andrea Bittau and Petr Marchenko. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of Networked System Design and Implementation (NSDI)*, 2008.

[21] Matt Blaze, J. Fiegenbaum, John Ioannidis, and oAngelos Keromytis. The KeyNote Trust-Management System Version 2. `http://www.ietf.org/rfc/rfc2704.txt`, 1999.

[22] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of USENIX Security Symposium*, 2004.

[23] B-tree FS (Btrfs). `https://btrfs.wiki.kernel.org/index.php/Main_Page`, 2014.

[24] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of USENIX Security Symposium*, 2018.

[25] Kevin R. B. Butler, Stephen McLaughlin, and Patrick Drew McDaniel. Rootkit-resistant disks. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2008.

[26] Kevin R. B. Butler, Stephen E. McLaughlin, and Patrick D. McDaniel. Kells: A Protection Framework for Portable Data. In *Proceedings of Annual Computer Security Applications Conference*, 2010.

[27] Kevin R. B. Butler, Steve McLaughlin, Thomas Moyer, and Patrick McDaniel. New security architectures based on emerging disk functionality. In *IEEE Security and Privacy*, volume 8, 2010.

[28] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Port. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[29] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[30] Dave Cooper. Internet X. 509 Public Key Infrastructure Certificate and Certificate Revocation List Profile. `http://www.ietf.org/rfc/rfc5280.txt`, 2008.

[31] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[32] crysetup. dm-crypt. https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt.

[33] John DeTreville. Binder, a Logic-Based Security Language. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2002.

[34] Dyinst. Dyninst: An application program interface (api) for runtime code generation. http://www.dyninst.org.

[35] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-oberwagner, Deepak Garg, and Peter Druschel. Thoth : Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of USENIX Security Symposium*, 2016.

[36] emscripten. https://github.com/kripken/emscripten.

[37] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, 2003.

[38] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2000.

[39] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[40] Michael Factor and Eran Rom. Capability based Secure Access Control to Networked Storage Devices. In *Proceedings of IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2007.

[41] Bryan Ford and Russ Cox. Vx32 : Lightweight User-level Sandboxing on the x86. In *Proceedings of USENIX Anual Technical Conference (ATC)*, 2008.

[42] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Jitguard: hardening just-in-time compilers with sgx. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[43] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *Proceedings of USENIX Security Symposium*, 2018.

[44] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. *ACM Transactions on Storage*, 8(4), 2012.

[45] Deepak Garg and Frank Pfenning. A proof-carrying file system. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2010.

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

[47] Robert Gawlik and Thorsten Holz. Sok: Make jit-spray great again. In *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[48] 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 ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[49] Enes Göktas, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (and What to Do about It). In *Proceedings of USENIX Security Symposium*, 2016.

[50] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[51] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SQCK : A Declarative File System Checker. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[52] Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of IEEE Computer Security Foundations Symposium (CSF)*, 2008.

[53] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

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

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

[56] Dave Hitz, James Lau, and Michael Malcom. File System Design for an NFS File Server Appliance. In *Proceedings of USENIX Winter Technical Conference*, 1994.

[57] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[58] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. librando: Transparent Code Randomization for Just-in-Time Compilers. *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[59] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[60] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2013.

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

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

[63] Intel Corporation. Software Guard Extensions Programming Reference. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`, 2014.

[64] Intel Corporation. Intel(R) 64 and IA-32 Architectures Software Developer's Manual. *Architecture*, 2016.

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

[66] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy*, 2001.

[67] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Anual Technical Conference (ATC)*, 2003.

[68] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2017.

[69] Ramakrishna Kotla, Tom Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. Pasture: Secure Offline Data Access using Commodity Trusted Hardware. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[70] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy Enhanced Secure Object store. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2018.

[71] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2017.

[72] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. Code-pointer integrity. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[73] Ninghui Li and John C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of ACM Symposium on Practical Aspects of Declarative Languages (PADL)*, 2003.

[74] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[75] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[76] Boon Thau Loo. *The Design and Implementation of Declarative Networks*. PhD thesis, University of California, Berkeley, 2006.

[77] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, Ion Stoica, Thau Loo, Petros Maniatis, Tyson Condie, Timothy Roscoe, and Joseph M. Hellerstein. Implementing declarative overlays. In *ACM SIGOPS Operating Systems Review*, volume 39, 2005.

[78] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[79] Michelle L. Mazurek, Yuan Liang, William Melicher, Manya Sleeper, Lujo Bauer, Gregory R. Ganger, Nitin Gupta, and Michael K. Reiter. Toward strong, usable access control for shared distributed data. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[80] Stephen Mccamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of USENIX Security Symposium*, 2006.

[81] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[82] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, 2008.

[83] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel software guard extensions support for dynamic memory management inside an enclave. In *Proceedings of Hardware and Architectural Support for Security and Privacy*, 2016.

[84] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *Proceedings of USENIX Security Symposium*, 2017.

[85] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[86] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8), 2003.

[87] Microsoft Corp. Bitlocker . https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview.

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

[89] Microsoft Corp. What Is Volume Shadow Copy Service?: Data Recovery. https://technet.microsoft.com/en-us/library/cc757854(v=ws.10).aspx, 2003.

[90] MITRE. CVE-2014-0160. https://nvd.nist.gov/vuln/detail/CVE-2014-0160, 2014.

[91] Node.js Foundation. https://nodejs.org.

[92] Oasis. eXtensible Access Control Markup Language, 2005.

[93] OCZ Technology Inc. Deneva 2 Data Sheet. https://drive.google.com/file/d/0B4hWjkpwenosS0VMOWZkZ1BtR1E/view?usp=sharing, 2011.

[94] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *Proceedings of USENIX Security Symposium*, 2016.

[95] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. In *Proceedings of ACM on Measurement and Analysis of Computing Systems*, 2018.

[96] OpenSSL. Crypto (OpenSSL cryptographic library). http://www.openssl.org/, 2012.

[97] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping Trust in Modern Computers. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2011.

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

[99] Adam G. Pennington, John Linwood Griffin, John S. Bucy, John D. Strunk, and Gregory R. Ganger. Storage-Based Intrusion Detection. *ACM Transactions on Information and System Security*, 13(4), 2010.

[100] Andrew Pimlott and Oleg Kiselyov. Soutei, a Logic-Based Trust-Management System. In *Proceedings of International Symposium on Functional and Logic Programming (FLOPS)*, 2006.

[101] Sean Quinlan and Sean Dorward. Venti: a new approach to archival data storage. In *Proceedings of USENIX Conference of File and Storage Technologies (FAST)*, 2002.

[102] Erik Riedel, Christos Faloutos, Garth a Gibson, and David Nagle. Active Disks for Large Scale Data Processing. *Tc*, 34(6), 2001.

[103] Rust language. `https://www.rust-lang.org/`.

[104] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.

[105] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, 1975.

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

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

[108] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Transactions on Information and System Security*, 14(1), 2011.

[109] Seagate Technology LLC. Kinetic Open Storage Platform. `http://www.seagate.com/solutions/cloud/data-center-cloud/platforms`.

[110] Seagate Technology LLC. Self-Encrypting Hard Disk Drives in the Data Center. Technical Report TP583, 2007.

[111] Seagate Technology LLC. Barracuda Data Sheet. `http://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barracuda-xt-ds1696.3-1102us.pdf`, 2012.

[112] Seagate Technology LLC. Momentus XT Data Sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_momentus_xt.pdf`, 2012.

[113] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2004.

[114] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2009.

[115] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, Haibing Guan, and Jiñming Li. Deconstructing xen. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.

[116] Jiwu Shu, Zhirong Shen, and Wei Xue. Shield: A stackable secure storage system for file sharing in public storage. *J. Parallel Distrib. Comput.*, 74(9), September 2014.

[117] Rui Shu, Peipei Wang, Sigmund A. Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. A Study of Security Isolation Techniques. *ACM Computing Surveys*, 49(3), 2016.

[118] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[119] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[120] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andre Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of USENIX Conference of File and Storage Technologies (FAST)*, 2003.

[121] SQLite. `https://www.sqlite.org`.

[122] SQLite. Speedtest1. `https://www.sqlite.org/testing.html`.

[123] Storage Work Group of the Trusted Computing Group. Self-Encrypting Drives Take off for Strong Data Protection. `https://trustedcomputinggroup.org/self-encrypting-drives-take-off-strong-data-protection/`, 2010.

[124] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[125] Sun Microsystems. Solaris ZFS, 2009.

[126] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[127] TCG. TCG Storage Architecture Core Specification. `https://trustedcomputinggroup.org/tcg-storage-architecture-core-specification/`, 2007.

[128] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Ant Rowstron, Tom Talepy, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[129] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[130] Kevin Walsh and Fred B. Schneider. Costs of Security in the PFS File System. Technical report, Computing and Information Science, Cornell University, 2012.

[131] David H.D. Warren. an Abstract Prolog Instruction Set. Technical Report Technical Note 309, SRI International, 1983.

[132] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. A taste of Capsicum. *Communications of the ACM*, 55(3), 2012.

[133] Carsten Weinhold and Hermann Härtig. VPFS: Building a virtual private file system with a small trusted computing base. In *ACM SIGOPS Operating Systems Review*, 2008.

[134] Carsten Weinhold and Hermann Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *Proceedings of USENIX Anual Technical Conference (ATC)*, 2011.

[135] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proceedings of ACM SIGSAC Asia Conference on Computer and Communications Security (Asia CCS)*, 2016.

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

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

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

[139] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1), 1994.

[140] Ted Wobber, Aydan Yumerefendi, Martín Abadi, Andrew Birrell, and Daniel R. Simon. Authorizing applications in singularity. In *ACM SIGOPS Operating Systems Review*, volume 41, 2007.

[141] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of USENIX Security Symposium*, 2012.

[142] Yuanzhong Xu, Alan M. Dunn, Owen S. Hofmann, Michael Z. Lee, Syed Akbar Mehdi, and Emmett Witchel. Application-defined decentralized access control. In *Proceedings of USENIX Anual Technical Conference (ATC)*, 2014.

[143] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[144] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM Yajin. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.