Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services

Youngjin Kwon[1]   Alan M. Dunn[2*]   Michael Z. Lee[3*]
Owen S. Hofmann[2*]   Yuanzhong Xu[1]   Emmett Witchel[1]

[1]The University of Texas at Austin   [2]Google   [3]Facebook

{yjkwon, yxu, witchel}@cs.utexas.edu   {amdunn, osh}@google.com   mzlee@fb.com

Abstract

Sego is a hypervisor-based system that gives strong privacy and integrity guarantees to trusted applications, even when the guest operating system is compromised or hostile. Sego verifies operating system services, like the file system, instead of replacing them. By associating trusted metadata with user data across all system devices, Sego verifies system services more efficiently than previous systems, especially services that depend on data contents. We extensively evaluate Sego's performance on real workloads and implement a kernel fault injector to validate Sego's file system-agnostic crash consistency and recovery protocol.

General Terms   Security, Verification

Keywords   Application protection, Virtualization-based security, Paraverification, Crash consistency

1. Introduction

A seemingly endless parade of privileged software compromises (e.g., OS-level zero-day exploits commanding half-million dollar ransoms as revealed by the Hacking Team emails [39]) and a desire to migrate sensitive computations to the cloud have energized research into removing trust from privileged software like the host operating system or hypervisor.

Once privileged software is untrusted, there are two approaches to system services: replace or verify (in this paper by verify we mean check at runtime that the behavior matches a specification). Many systems (e.g., Haven [11]

and MiniBox [25]) force an application to link into its address space much of its own trustworthy software, e.g., to protect persistent storage. Haven includes an entire library operating system. Finer-grained systems like Overshadow [14], Appsec [30], parts of Virtual Ghost [16, 17], and Inktag [23] verify system services at runtime, for example, using the guest file system, but checking that the guest OS is operating correctly (e.g., by verifying the cryptographic hash value of a secure file).

The advantage of replacement is that the application has complete control over its software trusted computing base (TCB). The disadvantage is that complex libraries, including library operating systems, have large attack surfaces that are likely vulnerable to the types of runtime compromise that plague existing complex libraries and operating systems. Systems that do not trust privileged software, but verify its behavior at runtime, incur time and computational expense to encrypt and hash data as it passes through the trust boundaries of the system. For example, SGX-enabled processors encrypt and decrypt data as it passes from the processor cache (trusted) to RAM (untrusted). Inktag and Overshadow encrypt and hash data as it passes from trusted RAM (protected by the hypervisor) to untrusted RAM (protected by the untrusted guest OS).

Exacerbating the performance and complexity problems of these systems is guest OS services that examine the contents of memory pages actively used by user-level code. For example, kernel memory sharing (KSM) reduces physical memory use by looking for pages with identical contents (as often occurs when running multiple virtual machines). Memory compaction copies memory to physically contiguous regions to enable those regions to be mapped efficiently with a single TLB entry (e.g., a 2MB page for Intel's x86 architecture). Whenever an untrusted guest OS touches a memory page of a secure process in Inktag and Overshadow, it triggers a hypervisor page fault and the hypervisor must encrypt and hash the page's data.

Finally, file system recovery is difficult to do efficiently when the OS is not trusted. Many file system operations that appear atomic (e.g., creating a file) consist of a sequence of updates to persistent storage. Journaling file systems make

---

file system operations efficient to perform and recover, but at the cost of complex data structures that introduce more intermediate states. The hypervisor has two problems managing persistent metadata: it cannot trust the OS's existing file system metadata and it generates new intermediate states during secure file operations (e.g., the secure process must first tell the hypervisor it wants to create a file before it tells the OS). Crashes that occur during the new state transitions must be cleaned up properly by the hypervisor. Fast recovery of persistent data for guest OS crashes is important because the guest can be malicious and crashing is a malicious behavior that can erase or make unavailable persistent data for secure processes and harms availability if it requires costly recovery.

This paper introduces Sego, a system that efficiently verifies system services at runtime; Sego makes protected, trusted metadata pervasive across all system devices, eliminating the trust boundary that necessitates encryption and hashing. In the Sego model, data stays in plain text, but it is protected by trusted components that have access to the trusted metadata. For example, Sego protects RAM using the extended page tables and secure page metadata, which is managed by the trusted hypervisor. Persistent storage is protected by persistent metadata that is accessed only by the trusted virtual block device.

Sego is faster than InkTag or Overshadow for workloads where the untrusted OS touches secure memory, because those systems must lock pages touched by the OS, and encrypt and hash their contents. Sego is 13%–15% faster for IO-bound workloads, up to $2\times$ faster for memory-intensive workloads with KSM active, and up to 19% faster when memory compaction is active. For the file system, Sego efficiently recovers files purposefully or mistakenly deleted by the guest OS. Previous systems, e.g., InkTag and Haven, provide secure file recovery after a guest OS crash; however, they require a full disk scan to find and recover all pieces of secure files. Sego's recovery time is proportional to the size of the recovered data, not the size of the storage device. For example, Sego is $11\times$ faster than Inktag recovering a 100MB secure file from a 24GB virtual disk, even if the virtual disk is completely cached in host memory.

The cost of pervasive metadata is a modified interface. In Sego, the untrusted OS manipulates pages by reference, making a hypercall so the hypervisor will perform the operation after verifying that the OS's request is legal (e.g., the OS is not trying to overwrite one trusted application with another's data). Sego changes the interface for untrusted software to make verification of memory page operations more efficient (designing interfaces that are efficient to verify is called paraverification [23]).

Our Sego prototype is implemented as part of the KVM hypervisor, and the QEMU hardware emulator. While Sego's model of pervasive metadata is amenable to pushing the TCB into hardware, our goal was to build a prototype that executes on currently available hardware. Therefore, Sego

modifies the hypervisor (protecting the CPU), uses nested paging (to protect RAM), and modifies the hypervisor's virtualized block device (to protect the file system). We modify the guest Linux operating system to conform to Sego's security model. It no longer manipulates memory directly, but uses hypercalls to initialize, copy, and test pages. We extensively evaluate Sego's performance on various real workloads, implement a kernel fault injector to validate Sego's crash consistency and recovery protocol, and demonstrate improved performance.

The contributions of this paper are the following.

1. Develop a model for secure computing where trusted metadata and untrusted data remain coupled, across all devices on which they reside (§3).
2. Design an efficient, secure data recovery protocol for applications that rely on a journaling file system provided by an untrustworthy operating system (§4).
3. Prototype and evaluate a trusted hypervisor that efficiently verifies untrusted system services that depend on data contents (§5, §6).

2. Overview and Background

This section explains the parts of Sego[1] that derive from other untrusted privileged code systems (keeping the paper self-contained). Because it is implemented in the hypervisor, Sego's lineage traces back to InkTag [23] and Overshadow [14], but much of Sego's assumptions are shared with other designs such as Haven [11], SP[3] [37], SecPod [35], and Virtual Ghost [17], though we emphasize the portions that are most relevant to Sego's novel contributions.

2.1 Threat model

Sego completely removes trust of the guest operating system. Therefore, Sego assumes the guest OS can (try to) read or modify any area of a user application's memory, and intercept or manipulate data en route to an IO device. It can modify control flow when a user application returns from a system call or interrupt. An attacker can intentionally crash the OS at any point to subvert an application. For example, when a user application updates its security settings (stored in a file), the attacker can crash the OS before the changes are made persistent and pretend as if the updates were persisted. The OS can try to revert file contents. For example, when a user application deletes a file and writes a new version of the file with the same name, the attacker can modify OS file metadata to point to the old version of file, rolling back the file's state.

The benefit of removing trust from the operating system is that it prevents the operating system from being a shared vulnerability across the entire system. Currently, if an OS is dynamically compromised using a vulnerability in one application, then all applications on the platform are untrustworthy. Systems like Sego allow trusted applications to continue

---

[1] Sego is from super-ego. It restrains the untrusted OS, ensuring well-adjusted behavior, regardless of purity of intent.
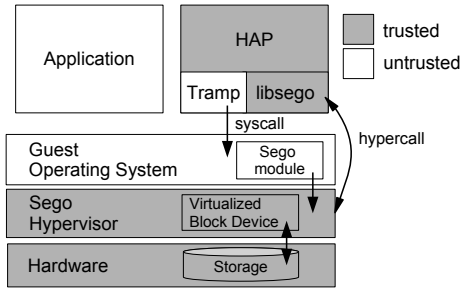
Figure 1: Sego design overview. In Sego, high-assurance processes (HAPs) make hypercalls to the virtual machine hypervisor to verify the runtime behavior of the operating system. The hypervisor and the virtualized block device are trusted. The section labeled Tramp is untrusted trampoline code and data used by the HAP to make system calls.

to function with a compromised and even overly hostile operating system.

To protect the application from OS, Sego relies on a trusted hypervisor and assumes hardware faithfully executes hypervisor code. Like most work in untrusted OSes, Sego does not prevent application bugs or application collusion with the OS. Applications can establish trusted channels through the untrusted operating system just as they can across a network, with Sego providing trusted identifiers. However, once an application trusts another application, Sego provides no protection.

Sego (and related systems) do not guarantee OS availability. A compromised OS can simply shut down or refuse to schedule processes. While such activity would be disruptive, it is also easily detected.

2.2   Sego security guarantees

Figure 1 shows an overview of the Sego architecture. Sego's trusted application code executes in a **high-assurance process, or HAP**. After booting the OS, the Sego hypervisor can start a HAP in such a way that the HAP can verify its own initial code and data, bootstrapping trustworthy execution in a way similar to a trusted program module (TPM) or software guard extensions (SGX). The details are identical to Inktag [23]. Once running, the Sego hypervisor ensures that a HAP's process context (registers) and trusted address space are isolated from the operating system. Context switches are handled by the hypervisor which hides information about the HAP's program counter and register contents from the guest OS. The Sego hypervisor uses hardware nested page tables to ensure privacy and integrity for the HAP's address space including code and allocated memory. The Sego hypervisor provides full address space privacy and integrity for the HAP's data and control flow. While we explain the high-level ideas, details are available from previous systems [14, 23]. Sego views the HAP's address space as a set of secure 4KB pages (called S-pages) that contain the HAP's code and data. Sego protects S-pages using nested paging (a set of page tables accessible to the hypervisor and not to the guest operating system). If the guest OS accesses S-page data, the

nested page table hardware faults and transfers control to the Sego hypervisor, which considers the OS action malicious and denies the operation (possibly terminating the OS).

Rather than extensively tracking guest OS state, the Sego hypervisor uses paraverification, where the hypervisor protects important guest data structures (like the page tables) and the guest clearly communicates its actions and intent to the hypervisor. We assume the Sego hypervisor can start as part of a trusted booting process [3], though the details are out of scope for this work. In order to operate on S-page data, a guest OS must make a hypercall and the hypervisor will perform the action after verifying that the OS action comports with the HAP's intent, as expressed to the hypervisor via a set of hypercalls. While the Sego hypervisor contains new hypercalls to verify guest OS operations, these functions are generally simple and only minimally increase the hypervisor attack surface. Section 3 provides more detail.

Sego supports secure persistent file storage (called S-files). HAPs allocate S-pages to load S-files into memory so that the OS cannot read or modify contents of S-files. To protect S-files resident in persistent storage, the Sego hypervisor maintains some per-file and per-file-data-block metadata and uses it to verify accesses to secure files (§ 5.1).

The Sego hypervisor refers to each object that it guarantees privacy and integrity by an identifier known as an **OID**. For example, OIDs name secure files and anonymous memory regions. They are 64-bits long in the prototype.

Sego does not manage network I/O, however applications can safely communicate over the network via mechanisms such as transport layer security (TLS [18]), that enable secure communication over an untrusted channel.

2.3   libsego

HAPs must express their intent to the hypervisor, so the hypervisor can verify OS activity. For example, if the HAP wants to read data from a file, it must inform the hypervisor of its intent and then perform the system calls to open and read the data. Forcing the HAP programmer to make many additional hypercalls would be burdensome, but fortunately it is unnecessary because system calls always require the same hypercalls to precede them.

HAPs link a small library (less than 5,000 lines) called libsego to get Sego services without having to change their code. libsego mostly handles system calls and can be viewed as a small modification to the C runtime library. For example, when a HAP tries to access an S-file, it transparently invokes recovery operations (§4) if needed.

Each HAP also contains a small amount of *untrusted trampoline* code that interacts with the operating system. The Sego hypervisor switches control between secure HAP code and the untrusted trampoline, and all system calls are issued from the trampoline, protecting HAP control flow. All system call arguments are copied into trampoline memory, and results are copied out by the HAP.

For example, when a HAP calls open, code in libsego translates the path name into an identifier (OID) and then

makes a hypercall to allow the hypervisor to check permissions. If the hypervisor approves the open, the libsego code calls the operating system by copying arguments into the trampoline, then transferring control to it via the hypervisor. When the system call returns, libsego verifies the results. For example, if the hypervisor approved the open, but the operating system claimed that no such file exists, libsego must take action to reconcile the discrepancy, if possible (see Section 4). libsego verifies system call parameters and the return value to prevent Iago attacks (§ 5.5).

3. Secure pages

Sego provides a uniform view of protected and trusted metadata with its associated data across all system devices (libsego, hypervisor, and virtualized block device): secure metadata and the data to which it refers are always logically coupled, no matter where the data resides. Previous systems that eliminate trust in privileged software provide an abstraction for secure data, but they all require complex management of secure data as it transitions between being protected by the system's TCB and residing in untrusted storage.

Sego (and many similar systems) manage secure data in fixed-sized units we call secure pages (S-pages). S-page data and metadata can reside in a processor's cache, in RAM, or in persistent storage. An S-page in Sego is a block of data with secure metadata specifying the persistent object (OID) and offset, uniquely representing the memory. Sego maintains the same secure metadata for S-pages across all system devices pervasively.

To contrast Sego S-pages with previous systems, SGX [24] only considers on-chip cache secure storage, so SGX-secured data is encrypted and MACed (message-authentication code) by hardware when it resides in RAM (or on persistent storage). The InkTag hypervisor also encrypts and MACs secure data as it moves to untrusted storage.

The Sego model assumes S-page metadata is bound to its data across all devices. This binding can be provided directly by the hardware, or emulated with some combination of hardware and trusted software. In our prototype, Sego is implemented in a hypervisor, and it protects S-pages in RAM using hardware memory protection (nested page tables) to ensure the untrusted guest OS cannot access them. The Sego hypervisor reads and writes S-page metadata to storage devices, emulating device support. Our Sego prototype does not encrypt or MAC data in order to protect it; it uses alternate techniques, such as nested paging and trusting the hypervisor's handling of block I/O.

Sego will terminate any untrusted privileged software that directly manipulates S-page data. Therefore, untrusted privileged software must be rewritten to access S-pages by reference. Fortunately, privileged software's access to S-page data is generally limited and stylized (e.g., it mostly needs to copy the data, and occasionally needs more complex operations like checking if two pages have equal contents). As we quantify in Section 6, modifying Linux to operate on user pages by reference is non-invasive. Sego modifies 1,572 lines of guest kernel code.

3.1 Pervasive metadata for S-page

An S-page abstraction that spans all system devices simplifies the model and implementation of the TCB, making security easier to achieve. It also allows efficient, protected S-page access for untrusted privileged software. For example, Sego requires less metadata per page than InkTag. InkTag requires 64 bytes of metadata for each 4KB disk sector[2], while Sego requires only 20 bytes[3]. Much of InkTag's overhead is for maintaining cryptographic hashes and initialization vectors for persistent data. While InkTag uses nested paging, Sego simplifies its use. In Sego, an S-page always resides in a trusted nested page table, while InkTag must move S-pages between trusted and untrusted nested page tables. Sego's approach is simpler because it knows when S-files are being read from IO devices and where the S-files are being placed in memory, so it can allocate the proper type of memory page (§ 3.3). InkTag does not trust IO devices, so it first reads encrypted S-file blocks into an untrusted page table and decrypts and moves them to a trusted page table when the HAP accesses the S-file data. Therefore, Sego's memory page management code is more than $10\times$ smaller than InkTag's (which is less than a thousand lines of code).

3.2 Pages in Sego

Sego views persistent and in-memory storage as objects that contain a sequence of 4KB blocks. Each block can be in one of three states:

Untrusted. An untrusted block may be freely used by untrusted software, privileged and unprivileged. Sego does not maintain metadata for untrusted blocks. All other blocks are **trusted**.

Reserved. A reserved block is protected by Sego and cannot be accessed directly by untrusted privileged software, but it contains incomplete metadata. Reserved blocks are used during non-atomic transitions, like when a memory frame is allocated to receive protected data from disk.

S-page. An S-page is protected by Sego and cannot be directly accessed by untrusted privileged software. Sego maintains the OID and the offset of each S-page. It sometimes tracks additional metadata, for example it adds a sequence number to anonymous memory S-pages.

Table 1 describes the interface the untrusted operating system uses to create and manipulate S-pages. The untrusted operating system uses its data structures (e.g., a `struct page` in Linux) to distinguish the states of physical frames. For example, if it needs a new empty page for a HAP, rather than initialize the page itself, it invokes the `0PAGE` hypercall, which zeroes the page, and protects it from the OS.

---

[2] OID and offset: 16B, hash: 16B, previous hash: 16B, IV: 16B.

[3] OID and offset: 16B, version number: 4B.

Hypercall	Description
`RESV(fr)`	Reserve a physical frame (fr), e.g., in preparation for a disk read. untrusted→reserved
`POPL(fr)`	Populate a reserved physical frame (fr) as S-page, e.g., in completion of a disk read. reserved→S-page
`SET_PT(fr)`	Guest OS requests Sego hypervisor to map and verify an S-page. S-page previously `POPL`ed, `0PAGE`ed or `COW`ed. S-page →verified S-page
`0PAGE(fr)`	Zero-initialize a new new frame (fr). The Sego hypervisor protects the page and fills it with zeros. untrusted→S-page
`COW(dst, src)`	The Sego hypervisor copies the S-page data and metadata, thereby marking *dst* a copy of *src*. S-page →two S-pages
`FREE(fr)`	Release an S-page. If the S-page contains sensitive data (i.e., is not already a zero page), it is zeroed by the Sego hypervisor. S-page →untrusted

Table 1: Sego interface for the untrusted operating system to request services from the hypervisor to manage physical memory frames. The description of the call concludes with the state change induced on the frame.

3.3 Reading and writing the buffer cache

Untrusted disk blocks can be read into untrusted memory frames, but the Sego **virtualized block device** will refuse to read untrusted data into trusted physical memory. The virtualized block device is code that executes in the hypervisor that implements the block device interface used by the OS. It is a necessary part of any hypervisor that exposes a physical or virtual block device to a guest. The untrusted OS must first reserve a frame before passing the frame to the virtualized block device as the destination of a read. When the virtualized block device receives a request for an S-page disk block, it ensures that the destination frame is reserved, then reads the S-page data into the frame and the S-page metadata from storage into hypervisor memory. Once the reserved frame is filled with data and its associated metadata is in place, the reserved frame becomes an S-page. Sego maintains S-page metadata in trusted storage as described in Section 5.1.

3.4 OS-generated S-pages

In addition to regular files, applications (including HAPs) have memory pages that do not come from a file on disk, but are instead generated by the operating system. For example, `malloc` will call `mmap` to generate anonymous memory to hold dynamically allocated data. When an application calls `fork()`, it expects its anonymous memory to become copy-on-write: if, after the fork, either the parent or child attempts to write to anonymous memory, the operating system creates a private copy of the page for that process. Maintaining copy-on-write semantics and verifying its correctness for these anonymous regions is challenging for Sego.

The Sego hypervisor tracks each HAP's anonymous memory regions as S-pages, analogous to the handling of files. Each anonymous memory S-page has an OID that refers to the owning HAP, and an offset that specifies the location of

the page in the HAP's virtual address space. When the parent process calls `fork()`, the Sego hypervisor clones all anonymous S-page metadata into the new process (via the `COW` hypercall). Although the hypervisor clones S-page metadata, the data segments for cloned S-pages may be shared, as long as the data remains read-only. Copy-on-write faults are processed by the guest OS and validated by the hypervisor.

To track S-pages generated by copying or zero-initialization, Sego relies on an S-tag to represent the contents of the page. An S-tag is a vector with several fields updated by the Sego hypervisor that tracks the state of the S-page as it is copied and modified.

The S-tag for the S-page has four fields:

⟨*OID, offset, sequence number, dirty bit*⟩

The S-tag is updated as follows: every time the S-page becomes writable, the S-page's OID and offset are copied into the corresponding fields in the S-tag. The sequence number is incremented, and the dirty bit set to indicate that the page may have been written. If the S-page becomes read-only, the dirty bit is cleared. When the untrusted OS tries to map the copy-on-write pages (using the `SET_PT` hypercall), Sego looks up the corresponding S-tag by OID and offset and verifies the mapping.

3.5 Discussion

There are advantages and disadvantages to the Sego model for secure data. Security-conscious users want their cloud data encrypted while "at rest." In the Sego model secure data is stored in plaintext. Sego's security model would be strengthened by hardware enforcement of trusted metadata, which would protect persistent data from malicious administrators, if not physical attacks. A security-conscious user can always use encryption and MACing with Sego, though at some performance cost.

An important question is whether hardware support for encryption and hashing will erase Sego's performance gains over an approach like InkTag. Currently, IO bandwidth is growing faster than encryption/hashing performance, keeping encryption/hashing a bottleneck for InkTag- and Overshadow-like approaches. CPU vendors are improving the performance of cryptographic operations. For AES-GCM (128 bit) the Intel Xeon E5 processor achieves 1GB/s throughput per core [6] and a blog reports up to 1.5GB/s on an Intel i7 4570HQ [7]. Sego shows a 13%–15% speedup over InkTag for an IO-bound workload using a single SSD, which has 250MB/s of write bandwidth (§6.4). Assuming the faster AES speed, Sego's performance win is reduced to 8%. However, recent SSDs can sustain 520MB/s IO bandwidth (e.g., Samsung 850 SSD Pro [9]) which would bring Sego's win back up to 34%. We obtained the 850 and measured Sego's sequential read performance as 37% faster than Inktag's.

Sego's elimination of encryption and hashing is important as the bandwidth of high-end IO devices are increasing faster than processor speeds (e.g., PureStorage announced 5GB/s–9GB/s of IO bandwidth in the FlashArray//m product [8]).

Device bandwidth can also be increased using RAID striping. Figures for how memory bandwidth differs between system and enclave memory for SGX are not yet available.

3.6 Future work

We hope Sego is attractive to device manufacturers because it gives device manufacturers a path to integrate trusted metadata into their devices and improve performance. Hardware support for trusted metadata would close the security gap of data not being encrypted at rest because even at rest, data is protected by hardware.

Another advantage of hardware support is that emulating secure devices can be expensive. The situation is similar to device support for virtualization [28]. As software virtualization gained popularity, device manufacturers directly supported virtualization, improving performance. If Intel's SGX [24] forms the basis for a new generation of trusted software, Intel and other device manufacturers might embrace the Sego model of pervasive trusted metadata because then they could support it in hardware, limiting the TCB and improving performance. For example, the Sego's virtualized block device could be implemented in an intelligent SSD [19]. Ultimately, technology trends and marketplace demands will determine the fate of trusted computing.

4. Secure files

A Sego block storage device is an array of 4 kB blocks, each of which can be untrusted, reserved, or a persisted S-page. Blocks by default are untrusted, but they are reserved by the virtualized block device when identified as the eventual location of in-memory S-page data and become persisted S-pages when data is committed. Users access secure data on disk via S-files, which are just like normal files, but privacy and integrity is guaranteed by the virtualized block device and hypervisor (the Sego TCB). In addition to per-block trusted metadata, Sego maintains the following per-file metadata: OID, file length, and access control information. Per-file metadata is persisted and managed by the hypervisor, in storage that appears as a file in the host system (though the file cannot be read or written by the guest operating system or any guest applications).

Sego provides fast recovery of secure persistent storage for guest OS crashes. One contribution of InkTag was to ensure the integrity of persistent storage in the presence of guest OS and hypervisor crashes. While InkTag's mechanism has been adopted by other untrusted OS projects [11], recovery for crashes (secure fsck) requires a complete scan of any attached disks. Sego must be able to recover from hypervisor crashes, but recovery continues to require a full disk scan, just as it does in InkTag. Sego focuses on the more common case of guest OS crashes.

Recovery of persistent data should be fast. The guest OS can be malicious and crashing is a malicious behavior harms availability if it requires costly recovery.

Fast recovery is difficult. The Sego hypervisor cannot trust the indexing structures of the guest OS (e.g., the journal) and must not harm I/O scheduling with many additional sync operations or reordering barriers. Sego's pervasive metadata model allows libsego, the Sego hypervisor and the virtualized block device to work cooperatively by sharing uniform view of metadata, detecting any inconsistency between S-file and S-file metadata and recover the S-file.

4.1 Sego's S-file model

To facilitate fast recovery and to preserve the security guarantees of S-files, the model for S-files is somewhat limited relative to normal files:

- **No sparsity.** S-files are not sparse; they must contain on-device disk blocks for all file data.
- **No shrinking.** S-files may be ftruncated only to zero length, or a length larger than the file's current length. If a file's length is extended via ftruncate, it is filled with zeroes to preserve the "No sparsity" property.
- **sync required.** A HAP cannot be sure that a file change is resident on secondary storage until it returns from a sync operation (e.g., sync, fsync, fdatasync). POSIX programs must also sync to be sure file updates are committed to storage, but the Sego hypervisor does more aggressive caching than most file systems.

Sego's recovery guarantee for secure files is that any data present on a storage device or in a device queue at the time of a guest OS crash is recoverable after the crash. Sego further provides privacy, integrity and prevention of rollback attacks for S-files. Recovery is initiated by HAPs (optionally, transparently) when they first access an S-file after a guest crash.

The Sego hypervisor and the virtualized block device collaborate to verify access to S-files, and to efficiently recover them. Sego storage maintains the invariant that each block of a secure file is stored at **a unique location**. For example, there can be only one block whose per-block metadata identifies it as OID X, offset 0. If there is more than one block corresponding to an offset of a secure file, then the OS can direct writes to only one of those blocks and direct reads to the other block, thereby achieving a **roll back attack**. The unique location allows the virtualized block device to locate an S-file's S-pages without scanning all of persistent storage, making recovery efficient.

The Sego hypervisor must track S-file length because it cannot rely on the OS. For instance, consider a S-file persisted up to length Y, and the OS records length $Y - x$ due to a legitimate journal rollback or a malicious action. If the Sego hypervisor believes the $Y - x$ is correct and allows a HAP to write data to that offset, the S-file will have multiple data blocks for the same offset.

Challenge of journaling file systems. Modern file systems like ext4, NTFS and XFS [33] support journaling for crash consistency, but these guest OS mechanisms only create challenges for Sego recovery: Sego cannot trust OS journals,

Recovery target	Inconsistency	Detected
S-file (`RSF`)	Sego creates an S-file but OS does not	When HAP opens the S-file
S-file length (`RSL`)	S-file length is different from persistent storage	When guest OS boots
Committed block (`RCB`)	Sego loses persistent data from an S-file	When HAP opens a S-file
Reserved block (`RRB`)	S-file data block corrupted (not correctly committed)	When Sego hypervisor runs secure fsck

Table 2: Recovery actions for S-files in Sego.

and must not harm a well-behaved OS performing proper recovery from a journal. The default behavior of journaling file systems, when a HAP writes a block of data, is for the OS to write a modified `inode` into its journal and to asynchronously write the data. When the data write reaches persistent storage, the Sego hypervisor increases the length of the S-file. A crash at this point would cause a (well-behaved) OS to believe the data block had not made it safely to persistent storage (because the modified inode has only been written to the journal, not to its on-storage location), and the file's length would remain unchanged. In recovery, the OS would disregard the uncommitted journal transaction, causing the OS and hypervisor to believe the S-file has different lengths (with the hypervisor believing the file is longer). To resolve this type of inconsistency, Sego introduces a novel recovery mechanism.

4.2 Recovery from OS crashes

Table 2 shows Sego recovery actions for different types of S-file inconsistencies between the hypervisor and OS. For simplicity, we discuss each cases individually, but it is possible that a single file needs multiple different recovery actions.

To prevent data loss, Sego must accurately track S-file existence (see §5.1 for hypervisor on-storage data structures). S-file creation is not atomic, so there is a window of vulnerability between when the Sego hypervisor believes an S-file exists and the OS believes the corresponding S-file exists. `libsego` checks on open for such inconsistencies and recovers by recreating the missing S-file in the OS file system (case `RSF`).

Figure 2 shows the timeline of events on the left, and the consequences of an OS or hypervisor (HV) crash on the right. A HAP creates an S-file, causing libsego to make a hypercall. The hypervisor creates the S-file and before it writes the metadata, if the hypervisor were to crash, both it and the OS would agree that the S-file does not yet exist. Once the hypervisor syncs its per-file metadata, then on a OS crash it would believe the file exists, while the OS would not. This is the most common disagreement between the hypervisor and the OS in our experiments (§6). Once the OS has (asynchronously) written the new inode to its location on storage, the hypervisor and OS would again agree on the state of the file system were either to crash. The OS region
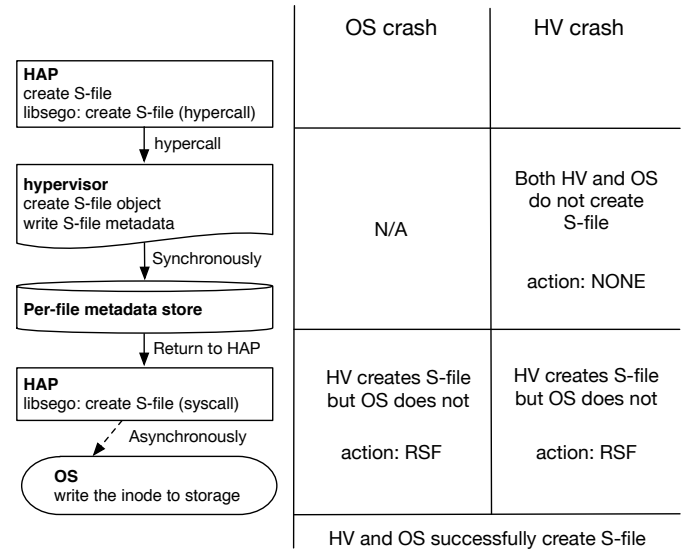

Figure 2: Windows of vulnerability while creating an S-file.

is marked N/A during hypervisor execution, because the OS cannot crash while it is not executing.

For writes, `libsego` extends the length of an S-file before writing data. If the OS crashes after the S-file length is extended, but before the OS persists the new S-page, Sego must recover the correct length of the written data (case `RSL`). The virtualized block device tracks the length of the S-file, shares the S-file state with the Sego hypervisor, and persists the metadata even when the guest crashes. The Sego hypervisor can determine the true (persisted) length of the S-file by examining the shared state from the virtualized block device without exhaustively searching storage. Figure 3 illustrates this complex case, specifying the possible disagreements between OS and hypervisor and the actions necessary to reconcile state after a crash.

The most challenging recovery case is recovering committed data blocks (case `RCB`). RCB arises when the guest OS rolls back an uncommitted journal transaction and the OS's notion of the length of the S-file is shorter than what the hypervisor's metadata says. When a HAP opens an S-file, `libsego` compares the OS length and the hypervisor length, and if the OS length is shorter, `libsego` initiates data recovery. A HAP indicates target S-pages in memory to hold recovered data with a recovery hypercall, and the hypervisor and the virtualized block device find and return the missing data. Importantly, this does not require searching through all persistent blocks. RCB can ignore all non-secure data block because the trusted block bitmap (§5.2) allows the hypervisor to distinguish trusted from untrusted blocks. `libsego` then makes system calls to write the recovered data into the S-file to allow the OS to come to agreement with the hypervisor on the contents of the S-file. Note that the guest OS is isolated from all recovery actions, and cannot access any recovery information.
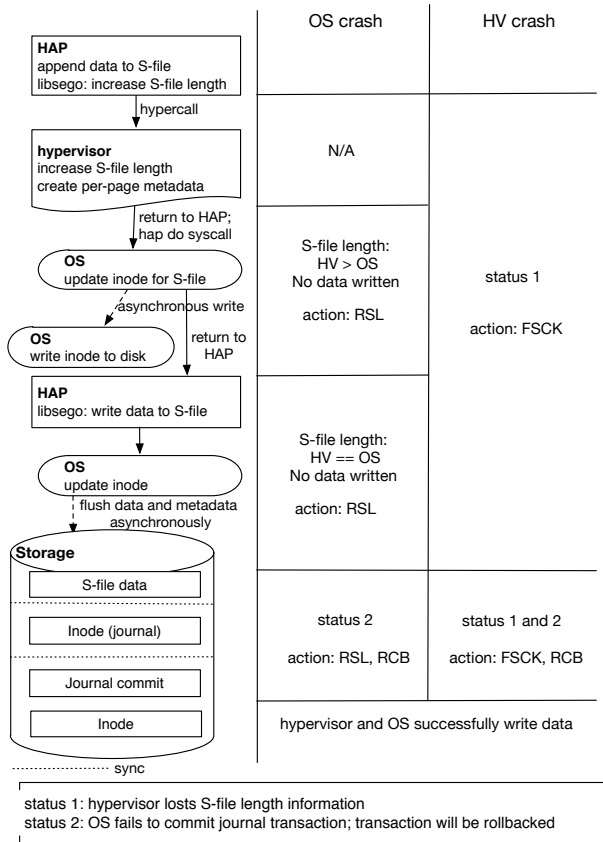
OS crash | HV crash

HAP
append data to S-file
libsego: increase S-file length

↓ hypercall

hypervisor
increase S-file length
create per-page metadata

N/A

↓ return to HAP;
hap do syscall

OS
update inode for S-file

S-file length:
HV > OS
No data written

action: RSL

status 1

action: FSCK

OS
write inode to disk

return to HAP

HAP
libsego: write data to S-file

OS
update inode

flush data and metadata asynchronously

S-file length:
HV == OS
No data written

action: RSL

Storage

S-file data

Inode (journal)

Journal commit

Inode

status 2

action: RSL, RCB

status 1 and 2

action: FSCK, RCB

hypervisor and OS successfully write data

.... sync

status 1: hypervisor losts S-file length information
status 2: OS fails to commit journal transaction; transaction will be rollbacked

Figure 3: Appending data to S-file. Guest OS uses journaling file system (Ext3 ordered mode in this figure). The dotted line "sync" implies the successive two blocks are stored in order, guaranteeing that the first is persisted before the second.

Hypervisor crashes will not lose secure file data, but recovering from a hypervisor crash requires a complete disk scan (InkTag's secure `fsck`). RRB recovery can only occur during a secure `fsck`. We leave optimizations for hypervisor crash recovery as future work.

4.3 Secure deletion: preventing file rollback attack

Deleting S-files creates a challenge for maintaining unique block locations, because the data and metadata for a single S-file can be scattered across a storage device. Consider a user who deletes a file, and then creates a file with new contents but the same name. The delete must somehow invalidate all blocks on the storage device before it can safely return, yet synchronously updating the metadata for every block in a file will be slow, especially for disks.

Sego provides a fast delete operation by adding a version number which is incremented on delete (as other file systems have done, e.g., LFS [31]). Sego defines the **effective OID** for a file as $hash(OID|verion)$. The hypervisor increments the version number when the HAP deletes a secure file by calling `ftruncate(0)` or `unlink`. The hypervisor and the virtualized block device maintain a mapping from original OID to effective OID. For crash consistency, the mapping
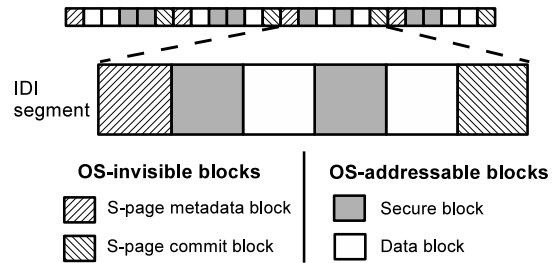

Figure 4: Sego storage layout. The untrusted OS reads and writes OS-addressable blocks, which are arranged in segments on the backing store and interleaved with OS-invisible blocks, that store S-page metadata.

information is synchronously persisted to per-file metadata store. The hypervisor stores the effective OID in per-block metadata, so the virtualized block device can separate active blocks and deleted block for secure files. Effective OIDs allow for fast, secure deletion of S-files.

5. Implementation

This section describes our prototype implementation of Sego for efficient, pervasive metadata and crash recovery.

5.1 Storage layout

The Sego hypervisor views persistent storage as a sequence of 4KB blocks. The Sego's virtualized block device executes on behalf of the hypervisor and helps track S-page and S-file metadata.

Figure 4 illustrates the persistent storage layout of *OS-addressable* blocks, which contain untrusted data or S-page data and may be read directly (for untrusted data) or indirectly (for S-pages) by the guest OS. The storage image also contains *OS-invisible* blocks, which contain metadata for S-pages. OS-addressable blocks are arranged in *segments* on the storage device, with OS-invisible blocks interleaved at a fixed ratio. The Sego hypervisor presents the OS-addressable blocks to the untrusted guest as a contiguous virtual block device. Sego interleaves its metadata, defining a virtual storage image format that we call interleaved data image or **IDI**.

For each segment of OS-addressable blocks, metadata for S-pages contained within those blocks is stored in two OS-invisible blocks. The *leading* invisible block stores S-page metadata such as OID and offset, and the *trailing* invisible block stores a commit block, that atomically indicates the completion of an update to both S-page metadata and data.

5.2 Implementing pervasive metadata

The Sego hypervisor (running as part of the Linux `kvm` module), its virtualized block device (running as part of the userspace `qemu` process), and the guest kernel share information about the state of physical frames in memory and on disk.

Trusted block bitmap The untrusted guest kernel and the virtualized block device share a bitmap of the trusted blocks, indicating which blocks contain S-file data (the trusted block

bitmap). Before submitting a disk request, the guest kernel (specifically, the `VirtIO` driver) checks the bitmap, and invokes a hypercall for a secure disk operation if the blocks contains S-file data.

Sharing S-page metadata with the virtualized block device
The Sego hypervisor and the virtualized block device share a map of physical memory to exchange S-page metadata. The map is indexed by the guest physical address of an S-page and contains S-page metadata for each entry. When the virtualized block device receives a write request with data in guest physical memory and a virtual disk address as a destination, it checks the shared map to see whether the request comes from an S-page, updates the secure block bitmap if necessary, and writes the S-page metadata, along with the data to virtual disk. When the virtualized block device receives a read request, it verifies the block's status by looking it up in shared memory: the destination physical frame must be marked as an S-page if the block is trusted or marked as untrusted if the block is untrusted. If the destination physical frame is an S-page, the virtualized block device populates the S-page metadata in the shared map by reading the S-page metadata block. The size of S-page metadata is 30 bytes[4] for each 4KB S-page, which is at most 0.7% of total memory size.

5.3 Recovering S-files

To aid secure recovery, the virtualized block device and the Sego hypervisor share a memory region. Once `qemu` boots, it allocates a fixed amount of memory and sends the address to the hypervisor via `ioctl` interface.

When `libsego` requests `RCB` recovery (see Table 2 in §4.2) to the hypervisor, the hypervisor maps the shared memory, signals the virtualized block device to fetch recovery data and send the recovery data to `libsego`. Most importantly, the Sego hypervisor never trusts any information from OS and the OS cannot access or modify the shared memory during the `RCB` recovery.

The first 4K page of the shared memory contains information for recovery such as OID and range of offset. Once the virtualized block device gets signal for a recovery request, it searches corresponding blocks for the S-file with the block bitmap information (without full scan of disk such as what InkTag does) and copies found data to the shared memory for recovery.

Sego restores S-page and S-file metadata consistency following a hypervisor crash (not guest OS crash) via an `fsck`-like process that reads the entire disk. Blocks having both associated S-page metadata and a valid commit block are considered secure blocks. If a block has S-page metadata, but no commit block, then it is considered reserved. Because the hypervisor cannot be sure whether or not private data has been written to the OS-addressable block, the virtual-

ized block device can only zero the block on disk to recover the reserved block. (`RRB` recovery)

5.4 Optimizing storage access

The InkTag storage layout is designed for crash consistency. When the OS writes a block for a secure file, the write is transformed into three writes. First, S-page metadata is written to the leading metadata block, which is invisible to guest OS. Then, the data is written to the OS-addressable block. Last, the commit block is written to indicate the completion of the previous two writes. To reduce seeks on disk, the InkTag's virtualized block device[5] writes metadata, data and commit block sequentially within each segment. The host (hypervisor) I/O scheduler can merge and reorder I/O request from the InkTag's virtualized block device so InkTag must `sync` after each write to guarantee that the three writes are persisted in order.

Sego batches writes to a single segment, writing leading metadata, all data blocks, and then the commit block in order. This process eliminates a sync for every data block written to a segment, beside the first. Segments are 64KB in our prototype.

5.5 Preventing Iago attacks

Sego prevents known Iago attacks [12]. It prevents attacks on `mmap` as InkTag does, using a user-provided token returned by the OS from `mmap`, which allows the user to verify that newly memory mapped regions do not overlap old ones. Sego saves and restores the `fs` and `gs` registers on context switches, because the `gcc` toolchain uses them for thread-local storage and other purposes.

Sego only supports unnamed POSIX semaphores, which reside in the processes' address space, disabling the OS from modifying their values. Sego uses uClibc for `HAP`s and the resume path in `sem_wait` checks whether the thread is being woken up by another thread/process doing a `sem_post`. This check ensures that a waiting process will not proceed when the OS wakes it up for any reason except a corresponding `sem_post`.

6. Evaluation

In this section, we evaluate the performance and crash recovery of the Sego system. For IO benchmarks we use a 3.4GHz quad-core Intel i7-3770, with a 500GB HDD and a 256GB SSD and for the server benchmarks we use a 2.8GHz quad-core Intel i7-860 with a 160GB HDD and a 256GB SSD. Sego is originally implemented at Linux 2.6.36 and QEMU 0.12.5[6] and ported to Linux 3.19.3 and QEMU 2.3.2. Our test system uses Ubuntu 14.04.4. Our guest VMs use a single virtual CPU, 2GB memory and a 12GB disk image. For the InkTag experiments, we turn off MAC functionality to

---

[4] OID: 8B, offset: 8B, version number: 4B, sector number: 8B, state variable: 2B.

[5] This is the similar component to Sego's virtualized block device and used for InkTag's storage layout

[6] The server benchmark, crash test and IO benchmarks are done using the older code base.

Service	Application	Non-HAP	InkTag	Sego
KSM	429.mcf	521	1095 (110%)	548 (5%)
	471.omnetpp	319	478 (49%)	333 (4%)
	470.lbm	218	351 (61%)	230 (5%)
	Graph analysis	200	278 (39%)	202 (1%)
Comp.	Memory read	237	298.6 (25%)	251.4 (6%)

Table 3: Execution time with KSM (in second) and memory compaction (in millisecond). Application slowdown is normalized by Non-HAP.

Domain	Masked	Short	Recovery			Total
			RSF	RSL	RCB	
File system	51	9	40	2	1	103
Kernel	85	7	10	0	1	103

Table 4: Random fault injection experiments, with faults injected to file system code and into the entire kernel. Either the benchmark ran correctly (faults Masked), the OS and Sego hypervisor both observed the same amount of data in affected files (Short), or Sego recovered a secure file, secure length, and possibly secure data (RSF, RSL, and RCB).

Domain	Recovery			Success	Total
	RSF	RSL	RCB		
File system	29	7	0	150	150
Kernel	26	0	0	150	150

Table 5: Random fault injection experiment for git

emulate GCM [20] (or upcoming hardware support for SHA hashing [5]). Sego modifies 1,572 lines of Linux kernel code for guest OS and 5,411 lines of KVM hypervisor code. InkTag modifies 962 lines of guest kernel code and 4,068 lines of hypervisor code.

6.1 Performance comparison with modern OS services

This section compares the efficiency of Sego and Ink-Tag when a HAP runs with modern kernel services: kernel samepage merging (KSM) and memory compaction. Table 3 shows application slowdown when the applications execute with KSM and/or compaction enabled. To quantify the overhead from KSM, we run three memory-intensive benchmarks from SPEC CPU 2006[7] and a graph analysis application [2]. KSM uses its default configuration, where a dedicated kernel thread scans 100 pages on every invocation and sleeps 20 milliseconds between invocations. The KSM thread constantly scans and test secure memory, causing severe performance interference (up to $2\times$) by InkTag's cryptographic operations. To analyze compaction overhead and thereafter memory access delay, we use synthetic micro-benchmark: this benchmark maps 500MB of secure memory, requests compaction and read the memory after compaction finishes. Our investigate reveals 64MB secure memory is moved by compaction (InkTag hypervisor encrypts the pages). When HAP reads the pages, InkTag decrypts the pages, which causes 19% slowdown of memory read performance.

6.2 Crash Consistency

We evaluate Sego's crash consistency via directed and randomized fault injection. We analyze file system code to determine four windows of vulnerability for different types of data loss and use System Tap [21] to inject faults at the beginning, middle and end points of each window (directed crash test). These experiments stress worst case behaviors that would be difficult to generate with user-level workloads. For example, when extending a file, Linux often merges the length and data updates into a single inode write, but we test a crash between these updates. Sego recovers from all faults in the directed crash tests for the `ext3`, `ext4`, `vFAT`, and `btrfs` filesystems.

We further validate that Sego can withstand and recover from faults by randomly injecting faults into a guest (that has an `ext3` file system). We use a kernel fault injection model [13, 34, 38], updating a copy of the injection framework from Swift et. al. [34] for modern kernels. We use two fault distributions, one for the whole kernel and one that targets file system code. These distributions are based on recent real-world fault studies [15, 26, 29, 38]. Our results are in Table 4. Our software fault injection framework is available online.[8]

A fault injection trial starts with the execution of four HAPs concurrently writing 20MB S-files, while 20 randomly selected faults are injected. Then four more HAPs are created to write additional 20MB S-files, all data is synced to storage and the system reboots. After the reboot, all of the S-files are read by the HAPs.

Table 4 shows the results of injecting random kernel and file system faults. Sego can detect and recover from every case where the OS and Sego hypervisor disagree on the state of the affected S-files. In our experiment, the RCB recovery happens when the guest OS believes the S-file does not exist (though the S-file actually exists and data are persisted). Thus, the recovery procedure is combined with two recovery mechanisms: re-creating the S-file (RSF) and recovering blocks of the S-file (RCB).

We do the fault injection evaluation to version control system, git. We make git (v1.2.3) run as HAP and configure the git to store all meta-information and object files as secure files. The evaluation starts with initializing repository, adding 20MB file and commiting the file, followed by sync. Git can persist meta-information (index, header for initial commit) and objects for current snapshot. Faults are injected with the same configuration used in table 4. 30MB file is added and commited to the repository and system shuts down. After reboot, we run git's consistency checking tool (`git fsck`). If git successfully finishes the consistency check, we count it as success. Table 5 shows the result. Sego recovers every trial enough for git to successfully run `git fsck`.

---

[7] We choose these three benchmarks because they have enough memory accesses to show a difference between InkTag and Sego.

[8] `https://github.com/ut-osa/fault-injection`

Recovery time for a secure file in Sego.

File Size	4KB	1MB	10MB	50MB	100MB
Disk	0.02	0.03	0.36	1.63	3.49
SSD	0.01	0.02	0.16	0.78	1.51

Recovery time for InkTag.

	Fully cached	Half cached	Not cached
SSD	17	41	65

Table 6: Recovery time (seconds). A 24GB virtual disk is stored on an SSD and the host caches different amounts of the virtual disk during recovery.

Server		Linux VM	Sego
Apache	throughput (req/s)	1526	1411 (7.5%)
	latency (ms)	65.5	70.9 (8.2%)
OpenLDAP	insert (ms)	1307.7	1515.7 (15.9%)
	query (ms)	1079.8	1118.2 (3.6%)
	delete (ms)	1269.4	1459.5 (15.0%)
DokuWiki	1 client (s)	7.9	11.1 (40%)
	16 client (s)	11.3	16.9 (49%)

Table 7: Server application performance.

Sego recovery provides an optimized path for recovering secure data that is maliciously erased by the operating system. If the malicious OS executes `rm -rf /`, Sego has trusted metadata distinguishing all secure disk blocks so it preserves their contents and rejects any subsequent OS writes to them. As haps try to access the deleted S-files, Sego recovers their contents by searching just the secure data on the disk, not the entire disk contents. Table 6 shows the amount of time it takes to recover S-files. To measure the recovery time of Sego, we delete (`rm -f`) an S-file and have a HAP reopen the S-file to recover the data. Most of the time is spent in the disk controller ($\sim$90%) reading secure data, and the HAP successfully recovers the entire contents of the S-file. The second table shows recovery time of Ink-Tag. InkTag has to scan entire disk to recover the length of S-files and inconsistent blocks since it cannot trust the on-disk metadata of OS. We measure the time by running `fsck -c` (the option that scans for bad blocks) on the virtual disk to estimate InkTag fsck time. Sego's recovery time is $19\times$ faster than InkTag when the entire virtual disk is already in the host buffer cache. Because Sego maintains fine-grained metadata, it does not need to recover the entire disk after a crash, it can recover only missing secure data.

6.3 Server applications

We build several server applications as HAPs, and evaluate their performance. The clients run in a different machine connected over 1Gbps Ethernet.

For the Apache webserver, we use the `ab` benchmarking tool to measure the performance overhead in Sego. The benchmark executes 10,000 requests, at a concurrency level of 100. Table 7 shows the results; throughput and latency overheads are below 9%.
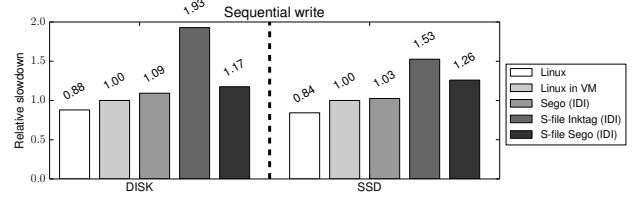

Figure 5: Sequential IO performance of disk and SSD. We normalize I/O throughput by the number of Linux VM; Slowdown of Linux is less than 1.

SSD	Linux in a VM	S-file Sego	
grep-src	0.20	0.22	(10.1%)
Fortune 50	1.80	1.95	(8.3%)

Table 8: GNU grep execution time in seconds. The standard deviation over 20 trials is less than 4% of the mean.

We build the OpenLDAP [4] server as a HAP, using Berkeley DB as the backend and secure all database and log files as S-files. We measure the overhead of inserting, querying, and deleting 200 entries, and the results are shown in Table 7. The overhead is 3.6% for queries, and under 16% for insert and delete. The server calls an `fdatasync` for inserts and deletes, in which the Sego hypervisor synchronizes the per-file metadata for crash consistency.

We run DokuWiki [1] as a HAP in Sego and follow the methodology from the InkTag paper [23]. DokuWiki is a PHP-based wiki that stores wiki data as S-files. We use the same corpus of wiki edits used in the InkTag benchmarks. We run a sequence of operations against DokuWiki via its XML-RPC interface, with a 90% read workload, where each write advances a wiki page to its next version. Sego runs DokuWiki's CGI binary as a HAP. We measure the slowdown for a sequence of 20 operations, averaged across 5 trials. Our results are shown in Table 7.

6.4 IO benchmarks

Figure 5 shows the sequential write performance of Sego on both disk and SSD. The micro-benchmark performs 200MB of sequential write with a 64K request size. The interleaved disk image (labeled IDI) has a segment size of 64KB and a 4KB block each for the metadata and commit blocks.

The write penalty for IDI is reasonably low ($< 10\%$), Sego's S-file write performance is 17% and 26% slower on disk and SSD respectively. We measure between 8 to 10 MB of additional metadata writes when writing a 200 MB S-file on SSDs. Sego outperforms InkTag by 76% and 27% in disk and SSD respectively because Sego reduces the number of `sync` by our I/O optimization and eliminates encryption and MAC for IO data. Our investigation identifies 13%–15% IO performance is improved by removing the cryptographic protection.

To measure IO performance with a real application, we compile GNU grep (v2.18) as a HAP and search for a string

(that is not found) in the grep source code and through a directory of the most recent form 10-K filing from the top 50 of the Fortune 500 (termed Fortune 50). The grep source code contains 918 files with an average file size of 10 KB and the Fortune 50 files contains 49 files (one of the companies is not publicly traded) with an average file size of 4.6 MB. Table 8 shows the wall-clock time for each setting. Grepping large and small files on Sego has about a 10% overhead though it is slightly worse when search through many small files. Opening an S-file induces an additional hypercall to check for file recovery, which shows up when working with many small files.

7. Related work

There has been significant work on providing verifiable system services built on untrustworthy operating systems. We summarize the contribution of many systems in a taxonomy to determine what has been done in the state of the art and what work remains.

Table 9 summarizes several recent systems, explains what properties they achieve and how they achieve them. Most abbreviations used in the table are explained in the caption. The table is divided into regions. The first region is about privacy and integrity of physical memory and address space mappings. The next region is about storage. Some systems distinguish storage for user's cryptographic keys; some systems support protected files; some assume applications will use their keys to encrypt and MAC their data. Sego and InkTag have focused on problems that arise with verifying the untrusted file system, while Haven replaces the untrusted file system with a library OS and most other systems assume a simple library for encrypted and MACed files. OS page indicates if a system allows the OS to temporarily swap a protected application's page to persistent storage (which it does to manage a limited amount of physical memory). Then follows access control, with HV acct indicating if the trusted system supports a user account distinct from, but parallel to OS accounts. In the final section, Split app indicates if the system requires an application to do privilege separation to isolate its security-sensitive components (which represents a significant barrier to entry).

Techniques for protecting the privacy and integrity of RAM is the area of widest innovation. RAM protection is the performance-sensitive basis for supporting applications on a hostile operating system, so it is logical that it is the focus of innovation. Most existing systems have not addressed address space privacy, so a hostile operating system can effectively get a page-granularity trace of any application's execution (and an effective attack has been published [36]).

A fundamental distinction is between systems that try to provide a protected file system and those that rely on applications (perhaps augmented with libraries) to protect their own data via encryption. Systems that support an untrusted file system (e.g., Sego and Overshadow) can `mmap` secure files, which provides compatibility with current systems and

a variety of services and optimizations. For example, `mmap` is the way Linux loads applications. It provides demand paging for executables and provides a basis for metadata-efficient integrity verification. Supporting a file system allows systems like Overshadow and Sego to support OS paging. Finally, verifiable file system support can include flexible access control.

Many recent systems have avoided providing file system support, relying on cryptography and do-it-yourself application libraries (e.g., virtual ghost and MiniBox). The simplicity of leaving the file system out of the list of system services that need to be verified is appealing. Without file system support, it is straightforward to support simple data storage for a single application (much like the app-centric storage model of iOS). If a user has privacy and integrity for a set of encryption keys, they can use those keys to encrypt and HMAC files. Encryption on files works well for a HAP to protect data that a future instance of the same HAP will access.

All features provided by an untrusted file system can probably be reimplemented by trusted applications, though these features include complex semantics and important performance optimizations. For example, application-directed paging would require new interfaces to allow applications to make swapping decisions. Paging requires the application and hypervisor to dynamically protect the privacy and integrity of memory pages and the integrity of their mapping. Virtual ghost has a design for OS paging, but does not implement it in their prototype, hence the Y/N. As another example of how systems compensate for lack of file system support, TLR relies on special hardware to prevent memory rollback attacks.

It is difficult to provide access control without a file system. Several systems feature user accounts in the hypervisor (though virtual ghost has a VM instead). Such accounts might be sufficient for the hypervisor to maintain per-user encryption keys. However, providing user and group-based access control common in file systems using only cryptography is still an open problem, especially for access revocation [10, 22].

minimizing the code in a HAP is desirable. It uses a shared wimpy kernel service that verifies only low-level I/O with the untrusted OS, not higher level services like file system operations (e.g., demand paging or copy-on-write).

8. Conclusion

Sego guarantees trustworthy services for user applications even when the guest operation system is malicious. We propose pervasive metadata for efficient verification and demonstrate substantial performance improvements for modern system services as well as IO-bound workloads. Sego provides crash consistency and recovery for secure data, which is validated by experiment. We believe Sego contributes a step forward for "secure" cloud computing.

System	RAM privacy	RAM integrity	AS privacy	AS integrity	User keys	FS privacy	FS integrity	FS crash	OS Paging	HV acct	Split app
Sego	EPT	EPT	None	PV & HV	Secure file	HV & Virtualized block device		HV recovery	Yes	No	No
InkTag [23]	EPT & Encrypt	EPT & HMAC	None	PV & HV	Secure file	Encrypt & Namespaces	Per-sector HMAC	Disk scan	Yes	No	No
Overshadow [14]	Shadow page table	HMAC	None	HV	Secure file	Encrypt	File HMAC	None	Yes	No	No
Haven [11]	HW Enclave	HW Enclave & HMAC	None	HW Enclave	LibOS file system	Shield module	Shield module	Disk scan	Yes	NA	No
Virtual Ghost [17]	Compiler & ISA	Compiler	None	ISA	TPM & VM	None	None	None	Y/N	Yes	No
MiniBox [25] TrustVisor [27]	EPT	EPT & vTPM	Pinned text	HV	μTPM & TPM	App	App	None	No	Yes	Yes
TLR [32]	ARM TrustZone memory		ARM TrustZone translation tables		Encrypt & memory HW	None	None	None	No	No	Yes

Table 9: A comparison of techniques that support applications running on untrusted operating systems. AS–address space, FS–file system, HV–hypervisor, EPT–extended page tables, PV–paravirtualization, attrs–attributes, PCR–platform configuration register. NA means the category is not applicable.

Acknowledgement

References

[1] DokuWiki. https://www.dokuwiki.org/.

[2] HPC Graph Analysis. http://www.graphanalysis.org/benchmark/.

[3] Intel Trusted Execution Technology. http://www.intel.com/technology/security/.

[4] OpenLDAP. http://www.openldap.org/.

[5] Intel SHA Extensions, 2013. https://software.intel.com/en-us/articles/intel-sha-extensions.

[6] AES-GCM Encryption Performance on Intel Xeon E5 v3 Processors, 2015. https://software.intel.com/en-us/articles/aes-gcm-encryption-performance-on-intel-xeon-e5-v3-processors.

[7] AES-NI SSL performance, 2015. https://calomel.org/aesni_ssl_performance.html.

[8] FlashArray//m Technical Specs, 2015. http://www.purestorage.com/products/technical-specifications/.

[9] Samsung V-NAND SSD, 2015. http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/ssd850pro/specifications.html.

[10] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: an online social network with user-defined privacy. In *SIGCOMM*, 2009.

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *OSDI*, 2014.

[12] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS*, March 2013.

[13] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, and David Rajamani, Gurushankarand Lowell. The rio file cache: Surviving operating system crashes. In *ASPLOS*, pages 74–83, 1996.

[14] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffery Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, May 2008.

[15] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP*, pages 73–88, 2001.

[16] John Criswell. *Secure virtual architecture: security for commodity software systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

[17] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *ASPLOS*, 2014.

[18] Tim Dierks and Eric Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2. http://tools.ietf.org/html/rfc5246, 2008.

[19] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *ACM International Conference on Management of Data (SIGMOD)*, 2013.

[20] Morris Dworkin. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf, 2007.

[21] Frank Ch. Eigler. Problem solving with systemtap. In *The Ottawa Linux Symposium*, pages 261–268, 2006.

[22] Ariel J Feldman, Aaron Blankstein, Michael J Freedman, and Edward W Felten. Social Networking with Frientegrity: Privacy and Integrity with an Untrusted Provider. In *USENIX Security*, 2012.

[23] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *ASPLOS*, 2013.

[24] Intel Corporation. *Software Guard Extensions Programming Reference*, 2015. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`.

[25] Yanlin Li, Adrian Perrig, Jonathan McCune, James Newsome, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. Technical Report CMU-CyLab-14-001, CyLab, Carnegia Mellon University, 2014.

[26] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *FAST*, pages 31–44, 2013.

[27] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*, May 2010.

[28] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. In *Intel Technology Journal*, volume 10, 2006.

[29] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *ASPLOS*, pages 305–318, 2011.

[30] Jianbao Ren, Yong Qi, Yuehua Dai, Xiaoguang Wang, and Yi Shi. Appsec: A safe execution environment for security sensitive applications. In *VEE*, 2015.

[31] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

[32] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In *ASPLOS*, 2014.

[33] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX ATC*, 1996.

[34] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, pages 207–222, 2003.

[35] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. Secpod: a framework for virtualization-based security systems. In *USENIX ATC*, 2015.

[36] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[37] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, pages 71–80, 2008.

[38] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. Is linux kernel oops useful or not. In *HotDep*, 2012.

[39] Kim Zetter. Hacking team leak shows how secretive zero-day exploit sales work. *Wired*, 2015.