Secure Storage with a Small, Verified TCB

Afonso Tinoco de Faria Cecílio dos Santos
Instituto Superior Técnico, Lisboa, Portugal

December 9, 2019

Abstract

In today's storage systems, policies related to stored data are maintained by complex relations of multiple levels of non-bug-free, unverified software. For instance, on most operating systems, filesystem policies are maintained and checked by the operating system connected to the storage device(s) where the filesystem and data is stored. This leads to the policies not being maintained in case there are failures in the operating system or other unverified components in this chain of trust.

In this work we propose *Secure Verified Storage* (SVS), a formally verified SSD firmware and storage architecture that enforces data policies at the storage layer, fully independent from the operating system and filesystems at upper levels. This ensures that in case of a fault of an upper component, properties related to confidentiality and integrity of data stored on SVS are maintained. We implemented the whole firmware in C that compiles to ARM in around 8000 LoC, making use of a custom version of the `mbedtls` library to provide SSL over ATA, and supporting atomic multi-operations through a batch API.

Besides the SVS architecture and fully functional implementation, we propose systematic techniques to verify rich security properties of embedded devices firmware under custom memory models, and we used them to do a partial formal verification of confidentiality and integrity properties of SVS . By using `frama-c`, we proved runtime security of the firmware code, functional correctness of a few data structures, and proved integrity and confidentiality of our firmware with respect to the ATA read/write API.

Keywords: Secure Storage, Firmware, Verification and Policy System.

1. Introduction

1.1. Motivation

Information storage is an essential part of today's computing systems. From the requirements on information privacy enforced by the GDPR[16] to the requirements on information integrity and confidentiality imposed by the fact that almost all the software that runs on nowadays personal computers uses a combination of local and cloud storage, we can see that information must be stored securely and that enforcing certain properties, such as integrity, availability, and confidentiality of stored data, is an important part of those systems. Implicitly or explicitly, software developers fulfill these requirements by defining policies that the systems will have to enforce on the stored data.

The storage systems that we deal with nowadays have these policy enforcing mechanisms split trough very complex subsystems, making security hard to manage. For instance, on a Linux machine with a storage device with ext4, data that can be stored there is split trough files that have access control policies to read, write, or execute the files. These policies can be about users, groups of users, or the owners of the files, however certain users can change the groups of other users, and some other users might be able to use the storage device as a block device, ignoring the filesystem policies entirely. This is an example for a very simple and common system. Furthermore, even for a simple website, we probably will have a database on top of the filesystem, which will make security depend both on the filesystem and on the database. As we move towards more complex systems, the components present on this chain of trust become harder and harder to keep track of - the operating system connected to the storage devices, all the applications on top of the operating system that deal with confidential data, and web servers that send this confidential data to third parties who in many cases should be the only ones with access to the trusted data. Even if we move toward simpler examples, such as a personal computer, all of the policies related to the file storage are managed by the filesystem that relies on the operating system to verify them. In case there is any vulnerability or bug on this operating system, the enforcement of all the data policies is at risk.

Besides this complexity that makes security hard to track if the program's logic is correct, most of these components of complex software still have other bugs and vulnerabilities inherent from programming, which are hard to eliminate, and whose quantity only increases with the number of lines of code and components present.

In this document we describe, implement, formally verify and analyse a solution that enforces storage policies directly on an SSD firmware, thus providing security properties even in cases where the operating system or other mid layer components are adversarial (in which cases we either provide full policy compliance, or stronger guarantees than any existing solution, ie, if the OS is compromised and user keys were leaked policies that do not depend on the leaked keys will still be maintained), and providing performance comparable to regular SSD read and write operations (either to a local disk or to a network attached storage device, depending on the physical location of the storage device). By targeting the storage device firmware directly we provide a small and verified TCB.

1.2. Existing Solutions

Existing solution for storage systems policies come in different approaches to solve different problems. The core set of related approaches, which are summarized

next, only cover a subset of the properties we are aiming to guarantee (A more detailed explanation of the most relevant related work appears in Chapter 2).
Policy languages exist with the purpose of expressing rich policies about data, and are an important part of security mechanisms. However, their policy enforcer is a component that relies on a large TCB and there is no verified implementation that we are aware of that targets directly storage devices firmware. Examples of policy languages are Guardat[15], Binder[7] and SecPAL[2].
Enclave based computing (such as SGX[4], TrustZone[1], Komodo[9]) works as a way to make sure that components dealing with confidential data are untampered by creating a small isolated piece of software with a small Trusted Computing Base (TCB) to deal with confidential data, and usually enclaves are small enough to be easily verified. There is some work done on the verification of enclave software, such as slashConfidential[14]. The problem with enclaves is that data storage is maintained by external components which become part of the TCB. In particular, SGX has the possibility to detect rollbacks of stored data between restarts by relying on an internal monotonic counter. However, detection relies on the SGX seal/unseal interface which has keys and counter bind to that specific enclave on the exact same machine as the original process that stored the data was running, which is not an ideal condition for distributed systems. Some solutions exist to counter this problem (ROTE[11]), but none of them has performance that is comparable to regular data storage devices and themselves have a very big TCB.
Protected Storage [3] systems are solutions that control access to storage data via the presence of a hardware token, a successful attestation of the host computer, or a physical password. The problem with these devices is that the available solutions are either self encrypting disks[13], which are disks that work in the presence of a password presented to the operating system connected to them, only protect against storage medium stealing, not providing any other kind of guarantees; or web storage services[8] like Amazon S3 provide the same guarantees based on Client identities, but trusting the whole (undisclosed) Amazon architecture, which becomes part of the TCB; or other types of network attached disks provide the same guarantees of Amazon, having a smaller TCB, but still containing an unverified operating system and drives on the TCB, and none actually having a verified implementation.
Protected filesystems exist in order to reduce the TCB of storage. For instance jVPFS[17] implements a filesystem monitor inside a microkernel in order to keep the rest of the kernel drivers outside the TCB. Other verified filesystem implementations exist, such as DiscSec[10], which provides a crash safe filesystem with confidentiality and integrity guarantees. The problem is that this filesystem runs as a fusefs filesystem, thus relying on the whole Operating System implementation again. Nevertheless the techniques used to prove integrity and confidentiality are very relevant to our work.

1.3. Objectives

In this document, we describe how we implemented and verified a solution that enforces storage policies at the storage layer, with a very small TCB, and with performance comparable to that of regular reads and writes to the storage device. Given the lack of verified solutions that provide actual policy enforcement and the large trusted computing base of those policy enforcing solutions, we believe our work is relevant as it gives very strong guarantees that stored data will not be tampered by an attacker, or by software bugs, thus setting ground for trustable policy enforcing systems even in presence of middleware component malfunction.

In the implementation part of our work, we intend to provide a firmware in C that compiles to ARM and runs on openSSD and that is able to enforce policies directly on the SSD, removing the OS from the TCB.

In the verification part of our work, we intend to provide proof of absence of runtime errors, of functional correctness and of confidentiality and integrity of stored data with respect to the policies. To do so, we will use frama-C with it WP plugin. With our work, we provide the following contributions:

- An architecture for trusted storage that is easily implementable in any project.
- The implementation of such architecture both in terms of storage firmware as well as client libraries.
- The verification of our implementation both in terms of safety as well as more complex properties, such as confidentiality and integrity properties.
- The presentation of a systematic verification methodology to deal with embedded systems security properties.
- The validation of the overhead of our solution against regular storage SSDs firmware when used directly connected to a computer, and against NAS storage units when used as a remote storage device.

2. Background

In this chapter we will briefly summarize some main concepts related to our project. For an in depth description of related work, please check our thesis.

2.1. Policy Based Security

In complex systems we need a way to describe security properties of stored and flowing data. The way it is done is by describing properties that the stored data and its handling must maintain – policies. After that, it is either responsibility of each component to follow these policies (less common, as we have to trust all the components that deal with data for the policies to be maintained), or there is a global monitor that enforces

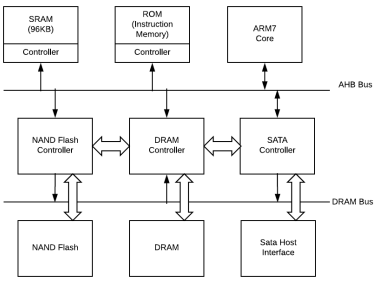


Figure 1: SSD Hardware diagram

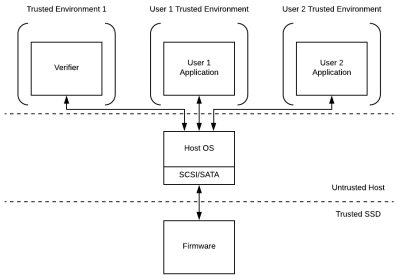


Figure 2: System Architecture

the policies.

2.2. Trusted Storage, Firmware

Several hardware based solutions exist to provide some sort of security guarantees on stored data. However, all of them are not extensible and focus on a specific concern. We will describe some protected storage devices. Embedded firmware has the property of usually having a tiny codebase with few software dependencies that runs on specific hardware, often with very tight computational and memory constraints. On the one hand, this allows to specify very thoroughly the security requirements of the firmware, on the other hand, the developer needs to have a deep understanding of all the behaviors that might occur at the hardware level, which is different from the model than the one usually used on more abstract software. The embedded system we intend to base our work on is OpenSSD [12], which is an open source hardware project that implements different types of SSDs and is intended to be easy to deploy firmware to.

Developing code for this kind of systems usually leads to some common bugs, on which there is extensive literature about [5, 6].

2.3. OpenSSD

OpenSSD is a project that brings an open source solid state drive hardware to be used on research and education. We will not focus our related work on their publications, instead we will try to do a brief description of how an SSD works, and what are the characteristics of the SSD we will use (which is OpenSSD Jasmine). The Hardware Architecture for the Jasmine SSD, is close to what we represented on Figure 1.

3. Design and Implementation

In this chapter, we will describe the solution we implemented. We will begin by describing the overall ideas used on the solution (section 3.1). Then we will present an in-depth description of the architecture (section 3.2).

3.1. Overall architecture

Given the lack of storage solutions that are trustable at its core, we implemented a system with a small TCB that enforces storage policies at the storage layer, more specifically we implemented an SSD firmware that enforces policies and also works as a normal SSD block device, and a client library to be run by the end clients.

The system has several components that can be seen on Figure 2. Overall, we have SSDs that may or may not run our firmware, and clients that want to access authenticated SSDs. There is an extra optional component, which is are external trusted verifiers, which the SSD may use to verify policies that itself cannot check (iex either due to performance or organizational boundary issues).

Our SSD communication protocol is implemented on top of the already existing ATA protocol and allows users to choose to do authenticated operation requests using new ATA commands, or to do regular unauthenticated requests trough regular ATA block read/write commands.

Our whole implementation was done in C that compiles to ARMv7, followed by formal verification of the implementation's C code with frama-C.

The main components of the system are the SSD firmware, the clients, and optional external verifiers.

On the SSD side, we have a notion of object, which is a sequence of extents on the SSD. These objects can have policies on a policy language, which will be checked every time an operation that affects those objects is performed. Operations on objects can be done trough the regular ATA read/write commands, if the policies do not require authentication. If the policies require authentication, then the client must first authenticate itself to the SSD. This authentication is done trough the client library, which creates a bidirectional secure channel (mutually authenticated and optionally[1] encrypted) from the client to the SSD on top of custom ATA commands.

In order for clients to authenticate themselves, they must first register with a PSK on the SSD and can in the future created a mutually authenticated secure channel based on that key.

The notions of object and policy on our system are very close to the ones defined on Guardat. However, we are not bind to any specific policy language, and instead we allow for the system to be extended through verifiers for different policy languages that our system can represent. These verifiers need to have their code

[1] To increase performance in case encryption is not needed

formally verified in order to be trusted and can be included directly on the firmware. However including them solely on the firmware might not always be possible so we have an extension to our system that supports external verifiers and authorization components. These external verifiers should run on an authenticated trusted execution environment or else they will weaken the existing trust model.

Objects can be created at any time on the SSD through a batch API, that allows to do several operations atomically. The data about object extents and policies is stored on the SSD as metadata on NAND sectors that are only acessible by the SSD firmware. While the SSD is powered on, all the metadata is cached on the SSD RAM for performance (so that when a read or write happens we don't have to access NAND flash storage).

3.1.1 Concept List

• Device - The SSD controller
• Host - The CPU and operating system connected to the device
• Verifier - A verifier in an external trusted third party that the device uses to verify policies that cannot be verified internally, and to establish sessions with the users.
• Sector - A sector is the basic R/W unit on the NAND flash, composed of 512 bytes. It is also the basic R/W unit exposed on the regular R/W ATA calls. On a regular R/W ATA call, we have a 6 byte lba used to specify the starting sector number and a 2 byte sector count used to specify the number of transferred sectors
• Extent - An extent is a range of bytes. (addressed at byte level). A given extent can belong to multiple files or to multiple positions on the same file.
• Object - An object is a sequence of extents. It can have policies associated with it. An object has an unique id (ObjectId) associated with it.
• Policy - A policy is a set of rules that must be maintained in order for a given R/W or Batch Operation to happen. Each policy has an unique id (PolicyId) associated with it. Policies are associated with objects and describe rules on the extents of those objects. In our implementation, we are not bound to any specific policy language, this means that we support any policy language that can be described using transactions on the objects described here, and we provided some examples of simple policies described later.
• User - A user is an external agent that communicates with the controller using an insecure channel. He can identify himself using a key, establishing a session with the device. Each user hash an unique id (UserId).
• Batch - A batch is a set of operations, that is processed in an atomic way. A batch is signed by a User in order to provide an identity. Each Batch has an unique id (BatchId).
• ATA RW Operation - An ATA operation is a regular ATA call to read or write to a specific sector on the SSD. It is done without any kind of authentication.
• Signed RW Operation - A signed RW Operation is a RW Operation to a given sector, bounded to a specific user, performed via our custom ATA commands.

3.2. Architecture

3.2.1 Extents to policy mapping, efficient policy lookup

In order to assign policies to files, we have objects, which represent a sequence extents on the SSD that can have policies associated to them. An object can have the same extend repeating multiple times, and multiple objects can have repeating extents (we allow so to easily support deduplicating filesystems).

In order to efficiently assign policies to files and to efficiently lookup if some sector on the SSD has a policy that needs to be checked when a read/write is in progress, we use an interval tree – that stores a mapping between extents and objects, and allows for very efficient interval lookup.

In order to safely test policies, when an extent is added to an object, we put it on the interval tree. When an extent is deleted from an an object, we remove it from the interval tree. When a read/write happens, we query the interval tree for all the objects that intercept with the extent of the read/write. Then for each policy associated with those object, we evaluate them according to the operation we are doing and if the operation complies with all of them we allow it, otherwise we don't.

3.2.2 Authentication, Confidentiality and Batch API

When an operation happens, we want to guarantee authentication and confidentiality both from the clients to the device – so that policy about owners of keys can be effectively verified – as well as from the clients to the device – so that results of operations can be actually attested. In order to do that, each client has a shared PSK with the device which they can use to mutually authenticate each other. Keys should not be the same between different clients. Key distribution can be done trough a trusted third party, present in the external verifier (see 3.2.3).

In order for the clients to sign messages, we use the Batch API. When a clients wants to do authenticated operations, he must always create a batch, which will generate a nonce that will be present on all messages from the client to the device for that batch. After that, the client writes to the device the operations he wants to do, followed by a signature of the operations and the batch id and the nonce. If the device checks that the signature matches the, then the batch is authenticated, otherwise it fails. When the device performs the operations on the batch, it can sign then sign a response with a value for either failure or success for that batch, which then knows can only come from the device. If we want confidentiality of messages, we have also sup-

port for the batch contents to be encrypted using SSL, instead of plain operations.

Another use for this Batch API is that is provides atomicity for operations. As we block all reads and writes on the device when we are closing a batch, and either perform all the operations on the batch or none of them. This also means, that when we are closing a batch, we must check the policies for all the operations on the batch and if any fails the whole batch will fail. Specifically in our implementation we achieve this by starting to replay the batch, and during and in the end of the batch checking if all the policies are still valid, if they aren't, we roll back the operations.

3.2.3 External Verifier

There are certain kind of policies that cannot be verified inside the device. For instance due to having high computational costs, or due to requiring data not stored on the device. However, we want our architecture to be expressive enough to still support those policies, and for that we implemented an external verifier. In short, the external verifier is a trusted third party that has a session with the device, and that is used to verify policies like the ones aforementioned.

In our implementation, when a user does an operation that would require an external verifier, the device will then communicate with an external verifier and request for the verifier to verify the operation. If the verifier verifies the operation, then it is successful, otherwise, it fails. The verifier might do extra calls to the device to request extra data required to evaluate the policy compliance.

In our implementation, we also allow for the external verifier to issue PSK's to authenticate users and devices. This way, we can support a much broader set of authentication protocols without having to include them on such a critical part of the TCB such as the device firmware.

3.2.4 Policies implemented and tested

Our implementation allows to have policies expressed in any language that our primitives allow to evaluate – any language that supports extent addressable file policies, with users/key base authentication, and with per operation policy checking, can easily be included in our firmware. Particularly, we tried to be as compatible with Guardat as possible, as it is a very expressive language.

For this thesis, we have implemented a few kind of policies as proof of concept, namely an access control list for each type of file operation, an append only file policy, and an external verifier only policy.

4. Verification

As we want to provide strong security guarantees about our code, not only we keep the TCB small, but we also formally verify some properties. In our project we formally verified some functional properties, and integrity and confidentiality (with respect to the policy language) of simple read and write operations using frama-C. In this chapter we will begin by giving an overall idea of the security properties we wanted to prove (4.1), followed by a description what kind of proofs we achieved and how (4.2).

4.1. Verification Methodology and Guarantees

In order to assure that our system will have secure behavior, we want to verify three kinds of properties:

- General code safety properties
- Code functionality properties
- Integrity and Confidentiality of stored data based on the policies

A great part of the firmware code was written in C that compiles to ARNv7, so in order to prove these properties, we used frama-C with the weakest precondition plugin. This means that we defined functional contracts about our code (separation logic pre-conditions and post-conditions), and frama-C generates first order logic formulas that are equivalent their correctness and code termination. This means that the correctness of our code, is not only dependent on the proof, but also on the specifications we defined in ACSL. We tried to keep the specifications as simple as possible, in order to reason about them in a simple and straightforward way.

In frama-C, proving properties about the code translates to adding assertions and invariants throughout the code, as well as adding and proving predicates that the SMT solvers can use to prove the formulas. Therefore, most of the verification work done was about refactoring the code and rewriting it in a way that was simple enough such that writing provable predicates was doable.

4.1.1 Runtime safety properties

For the general code safety properties, we began by proving that common bugs such as buffer overflows, integer overflows, etc. do not occur. This makes other subsequent properties of the code easier to prove (in the case of frama-C it is a necessary condition for actually proving anything about the code). In order to do so, we just used the runtime errors plugin of frama-C, which automatically generates conditions for the safety of all memory accesses and signed arithmetic operations.

However, given the way CPU SDRAM and regular DRAM were separated in our processor, the memory model in frama-C was not enough, as it was only able to deal with the SDRAM. We had to create predicates to extend this memory model to make it aware of the existence of two types of memory that need to be accessed differently, and therefore all our pointers are typed additionally to the type of memory they belong to. Additionally, our memory functions that deal with SDRAM, have contracts that could not be proved as they change the type of pointers, but we just assumed their implementation correctness with respect

to typing as this functions are impossible to prove, as they basically define a memory model for this CPU and they are just 3 or 4 lines (iex, DRAM_READ_WORD, DRAM_WRITE_WORD, DRAM_TEST_WORD, ...).

4.1.2 Functional properties

In order to make the code safety properties provable, we actually have to define contracts for most functions, as whenever a function receives a pointer, it needs some precondition about the validity of that pointer, and such property is not automatically inferred by frama-C. So our initial approach at functional correctness was just defining memory and integer range conditions for all functions. As we kept doing it, we managed to prove a lot of properties about memory safety, but some of them were not proved yet, because they required a complete functional definition of th data structures used, and that's when we reached the second part of our proofs. In parallel with the initial pass, we defined and proved functionally properties of the several components of the SSD firmware. To do so, we defied predicates about abstract data structures and we proved that the functions that deal with those data structures have those predicates as pre and post conditions.

4.1.3 Integrity and Confidentiality

The other remaining properties we want to prove are the more interesting security properties – Integrity and Confidentiality of stored data. In order to do this, we used the same reasoning as on SFSCQ, as there are a lot of similarities between a filesystem and an SSD. Our definition for integrity and confidentiality is the same as on SFSCQ – data non-interference, but adapted to a more generic definition in terms of policies, and more appropriate for the SSD.

For integrity – If a user writes to a sector or puts a policy on it, a user that doesn't have write/change policy access to that sector according to the policies on that sector cannot do valid operations on the SSD in order to alter the stored content/policies on that sector. (this translates to: a valid write followed by a read, is indistinguishable from a the same valid write followed by many invalid writes followed by a read)

For confidentiality – If a user writes to a sector or puts a policy on it, a user that doesn't have reads policy access to that sector according to the policies on that sector cannot do valid operations on the SSD in order to gain any information about the stored data on that sector, other than what the policies of that sector allows to infer. (this translates to: two invalid reads on a sector, are indistinguishable if their policy is not dependent on their content).

In order to prove both of these properties, we used the same methodology as SFSCQ – we reduced all our reads and writes from the SSD to two functions, and proved that all codepaths that lead to those functions mandatorily call the policy verifier on the policies of the files stored on those sectors. By doing this, we don't need to reason about temporal logic, we just need to prove correctness of storing and retrieving the policies associated with some sector. As for the policy verifier, given we never implemented any policy language, we just require that it's correctness is proved separately, and is an assumption for our proofs (we didn't even included it in our proofs, our only condition is that the verifier is called).

We were able to partially prove this properties with respect to the single read and write API. We also wanted to prove the same properties with respect to the batch API, but such was not possible given the time we had for the thesis, and the atomicity and quantity of operations of the batch API making the predicates and proofs much more complex.

4.2. Verification Framework

4.2.1 Memory model

Our program has two types of memories that need to be accessed differently. The SDRAM follows the traditional memory model that frama-C is aware of, in which memory accesses are readily available by reading and writting to addresses, on the other hand, the DRAM needs to be accessed trough a Memory Management Unit, and it's values might not be ready on the next instruction making us have to test the state of the MMU ready bit. (The NAND memory can be accesssed and programmed trought the same MMU, but we will not describe it so deeply here, as it is basically just another type of DRAM except it is persistent on poweroff). These two memories also share different address spaces, therefore all pointers and pointer arithmetics in both memories should respect those spaces. As frama-C is not aware of this memory model, we had to extend the pointer logic by adding a new type to pointers in frama-C, trough adding predicates that need to affect all the pointers, and asserting information about all the addresses we use initially.

After this, we also had to define the memory interfaces to accessing DRAM from SDRAM, we kept them simple enough such that their implementations does not need proving (their code was actually generated by xillinx and then manually optimized). Regarding defining the ranges of DRAM that are valid, we actually abstracted that from the code functionality, and made all memory accesses to DRAM go trough the memory allocator, except on cases where we use memcpy, which we just require that both addresses are on DRAM.

4.2.2 Memory allocator

Our memory allocator manages how we allocate and use memory inside DRAM. Basically, we have different DRAM regions, used by different memory allocator instances, and each one of them stores objects of the same type. Based on this, we can express invariants about the DRAM objects, and say that once we read them to SDRAM, they will be objects with those in-

variant, and that when we store them on DRAM, we require that the invariants are still maintained. These fact that all memory accesses go trough the memory allocator allows us to easily type DRAM this way.

In order to prove properties about our memory allocator, we used ghost code in order to specify that the allocator is similar to a simple program that returns a free block if there is any.

Our specifications for xmalloc and xfree are basically build around the bitset of free and non free_blocks. The expected behavior for malloc is that it will return a valid free slot if there is any and mark it as used, or 0 otherwise. The expected behavior for free is that it will free the slot it was called with as argument, and have a precondition that the slot was not marked as free. It is worth noting that for our specification the 0 pointer has a special meaning, as the nullpointer, and therefore is always marked as used, even after a free.

In order to prove actual functionality of the memory allocator we had to add the ghost code to correctly update the ghost properties, and to add some predicates to help smt solvers reason with the binary heaps in our code. We also had to simplify the design of our binary heaps.

5. Results

We envision that our solution should evaluated according to 4 metrics:

- Efficiency - is the code efficient when compared to regular (policy free) storage?
- Verification - was the code verified successfully and what was the effort required to verify it?
- TCB size - is the final system TCB small?
- Metadata size - is the memory usage of our implementations on metadata acceptable?

In order to evaluate efficiency, microbenchmarks and synthetic read/write benchmarks can be run on the system. Evaluating operation latency and throughput is something possible by both of these solutions. (section 5.1)

In order to evaluate the metadata size and efficiency, we can use application level benchmarks on the system (section 5.2). A solution that mixes interesting use cases and with this kind of evaluation would be ideal.

In regards to Verification (section (section 5.3)) and TCB size (section 5.4), no experiments are needed or possible, and the best we can do is compare them to other solutions. We can try to fuzz our system after verification, but we expect the results to be blank if the verification was complete. We evaluate the effort required to verify the code by measuring the time spent in the verification process as well as the percentage of the code that was verified.

5.1. Microbenchmarks

In order to evaluate read/write performance we did some synthetic benchmarks on reads and writes from the operating system on 3 versions of the code:

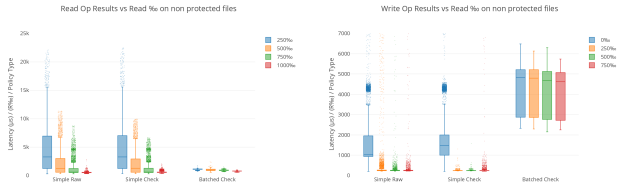
- Baseline - Original firmware accessed trough the regular read/write ATA API.
- Regular Version - Our firmware accessed trough the regular read/write ATA API (which wil do policy checking on every read and write).
- Batched Version - Our firmware accessed trough the batch read/write API, doing a single operation per batch (this will always be worse than the performance of multiple operations per batch).

We also varied the number of protected files on each test from 0 to 16300, and for each test we qualitatively selected if we wanted to do operations on protected files or on non-protected files. The policy used was a simple always accept policy.

We ran the tests under different read/write workloads for 5 minutes on each configuration.

All of the tests were run on an windows machine with an i7-7700k processor with 32GB of 2133Mhz RAM, connected to jasmine openSSD via SATA. Both the motherboard and the processor communication throughput greatly exceed the maximum throughput of the jasmine openSSD hardware, therefore this hardware setup is adequate.

We plotted all the benchmark points related to non-protected files on figure 3.



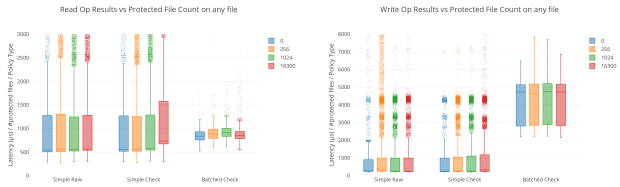
(a) Reads (b) Writes

Figure 3: Benchmark on non-protected files.

Regarding the non-protected files results, we can see that as expected average protected firmware overhead is small when compared to the baseline raw firmware for all workloads. Regarding the writes, we can see that only for the highest write intensive workload does the performance greatly decrease when compared to the raw version (we see an increase of 350μs of the average latency, or 20%). For the batched operations the latency does not vary with the workload, which is expected as the batch multiple requests overhead covers the pending writes latency.

We also used the results of the previous tests to measure the impact of the number of protected objects on the average latency of operations (figure 4). As we can see from the graphs, the latency of the stock firmware is not influenced by the number of protected files, while the simple check version increases logarithmically with the number of protected files, having on the worst case an increase in latency of 200μs, which agrees with the performance tests on the interval tree.

At last, we also measured what would be the impact of the external verifier on write operations. We ran the



(a) Reads (b) Writes

Figure 4: Benchmark on the impact of file count.

same tests under different workloads, with an extra requirement that on the batch version, writes needed to be approved by an external verifier in a remote computer (simulated via a 50ms approve latency) which would always approve operations (figure 5). We can see that the performance increases linearly with the performance of writes, as expected. The latency of the verification (60ms) is a little bit higher than expected, but we can attribute the overhead to time spent on scheduling the external policy verifier on Windows.

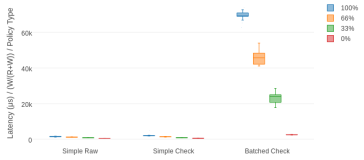


Figure 5: Microbenchmarks for external verifier writes.

5.2. Usecases (Macrobenchmarks)

We will describe some use cases where our solution might be effective and bring stronger security guarantees. These cases might be interesting to explore as future work, but we only evaluated a single usecase, which was as a webserver user data policy enforcer.

5.2.1 As local backup protector in case of exploited OS

When virus attack a computer it's common for them to modify existing files, either for ransom or persistence. With our solution, a user can both force OS related files to be only updatable by the OS distributor, as well as protect backup files from being modified (such as personal pictures, for instance).

5.2.2 As a support for SGX persistent storage

SVS is an ideal solution for SGX storage persistence problem, as it provides an CPU independent solution for confidential, integrity and rollback protected data storage.

5.2.3 As a medium for a trusted unverified filesystem

SVS can be used in conjunction with some unverified filesystem in order to provide read/write/execute permissions not managed by the operating system. This is a topic we haven't explored and don't think of exploring this during our thesis, but is interesting future work.

5.2.4 As a webserver user data policy enforcer

With the regulations required by the GDPR, any solution that can help enforce policies on user data is interesting. Therefore, using SVS to define policies about data on a webserver is highly relevant and we ran some macrobenchmarks on this use case.

Using the same hardware setup as for the microbenchmarks, we implemented a simulation of a webserver that uses SVS in order to serve wikipedia pages. We used a dataset from the page view logs of an hour of wikipedia [18]. Given the size of the SSD and number of files restrictions, compared to the size and number of pages of wikipedia, we had to pick a sample of the pages, so we picked the ones that were more accessed during that week until the number of files reached the maximum we could store on our SSD. Then, we randomly assigned two simple policies to each file: pass or fail_randomly(p), and ran the test. As a baseline for the test, we picked the top 16000 most used pages, and ran the requests on the log for those pages for one hour on the original unmodified firmware. After that we ran the same logged requests on the SVS firmware, using the secure ATA read and write interface, and measured both the total running time, as well as the average overhead per request over time. We plotted the results on figure 6.

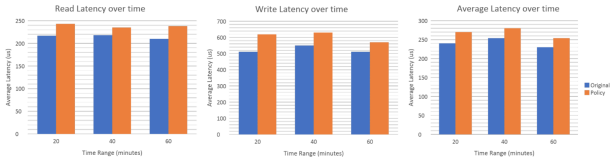


Figure 6: Benchmark on the wikipedia log.

We used 16000 files with 15000000 read or write operations over all the requests wikipedia served for those pages under one hour, on the same single box. The overall time to run the tests on the verified version was 67 minutes, which is a 12 % throughput loss. From the graphs we can see the maximum latency increase versus the original version was 20% for the write latency on the first twenty minutes of the test.

From these results, we can see that overhead of the solution is still low enough not to be a bottleneck to run a wikipedia node on a single machine (of course in terms of network bandwidth and webserver memory this solution would not be adequate and more servers

would be necessary, as well as serve more pages than the number of files we supported).

5.3. Verification effort results

On this thesis, we verified properties about 8000 LoC (30% of which are comments). Given the huge proof effort necessary, a few less relevant things were left unspecified or unproved. Namely, we skipped all of the SSL code specification and verification, as it would be very difficult to do correctly, and very time consuming. We also did not completely prove a lot of functional properties about some of the data structures we used, as it seemed less relevant and could be achieved with just more hours of work. We took around 300h to specify and verify the 8000LoC, showing that each line of code takes on average 3 minutes to verify. We wrote around 4000LoC of verification statements, mostly on ACSL function pre and post conditions, and datatype invariants. Our number of lines of code falls very short of the expected 1:2 lines of actual code vs line of verification expected ratio, this is the consequence of both not fully specifying some of the data structures used (given lack of relevance for the scope of the work), as well as the fact we also did not prove every property, given missing properties to verify on each module were taking more and more time to verify, as they often needed major code rewriting. Overall, frama-C generated 2000 formulas that needed proving with the help of SMT solvers, and with all the code properties we added, alt-ergo and Z3 were able to 90% of them. We proved all the safety properties for most of our code files. Leaving a big amount of unproved code on the ftl, the NAND operation wrapper, on the memory functions (all the memory code was abstracted by calling always calling a few memory functions which we asserted as following their contract), and a big part of the interval tree was abstracted to be a set of files. We also did not completely specify or prove the batch API.

Overall, the verification effort done on the thesis allowed us to fix a relevant number of bugs on the code, and to give stronger guarantees of the safety properties our system achieves. Most of the bugs we were able to find fall on two categories described on the related work: out of bounds array access and integer overflows/wrong typing. Our memset could overflow given we were used integers as iterators and we were able to find it when frama-c was not able to prove loop invariants nor loop termination (on our actual hardware this could never happen, because we never had pointers, but it was an interesting bug to find on such a simple piece of code).

5.4. TCB and Metadata sizes

Our TCB and metadata size are easily measurable and both are small as efficient.

5.4.1 TCB size

Regarding the TCB size, the easiest measure is the number of lines of code. The lines of code of our project can be separated in 4 categories:

- Core Logic - 4000 lines of code
- SATA and FTL code - 3900 lines of code
- SSL code (mbedtls) - 10000 lines of code
- Windows/Linux API - 500 lines of code

Given the fact that our core logic is only 4000 LoC, we can say the core of the TCB is reasonably small, and therefore very interesting. This core logic code is also most of it verified.

Regarding the SATA and FTL code, the number of LoC was a bit out of our control, as most of that code was already developed and we needed it.

Regarding the SSL code, we used an industry certified ssl implementation, which gives us very strong guarantees, as most of its code is showed not only resistant to the kind of threat models our verification protects against, but also against side channel attacks, whose proofs are much more complicate.

Our Linux API, is very straightforward, and even though we didn't verify it, it is not a potential source of security bugs.

In total, our potential attack surface is 8000 LoC, which although big, is not that relevant for a project this big. If we remove white lines and comments, this number drop to 3050 LoC, which we consider very positive.

5.4.2 Metadata size

In our implementation, we limited the number of object, extents, and policies to a fixed number in order to maximize them based on the available DRAM. More specifically we have 64mb of DRAM to share with SATA structures, and we allow to specify limits on the number of objects, extents and policies based on the usecase. In this subsection we will try to do a brief overview of the limits we have in terms of memory, but overall, the metadata overhead is in general minimal and close to optimal uncompressed, without any high memory to performance tradeoff.

The metadata overhead of our system depends on three parameters: O (the number of distinct Objects), D (the number of distinct Datablocks (extents)) and K (the number of distinct Policies (the same policy can be assigned to several objects)) Based on the sizes of each structure on our code and the maximum amount of DRAM available, we plotted the metadata maximum DRAM usages for the some limits on these parameters on table 5.4.2.

From table 5.4.2, we can see that our metadata usage to provide per page protection on 64GB disk would not be that high, and although we can't give per page protection on Jasmine given the small amount of DRAM it has, commercial SSDs typically have much more RAM (on the order of a few GB), showing this is a viable solution even for larger SSDs.

$\overline{O}$	$\overline{D}$	$\overline{K}$	memory required (MB)
2^{20}	2^{22}	2^{16}	94 [per page protection]
2^{16}	2^{20}	2^{8}	18.7
2^{17}	2^{17}	2^{8}	4.44
2^{14}	2^{16}	2^{10}	1.29
2^{16}	2^{21}	2^{8}	36.94
2^{19}	2^{20}	2^{8}	27.34
2^{19}	2^{20}	2^{16}	29.34

Table 1: Maximum metadata overhead for some parameters

6. Conclusions

With all the work developed on this thesis we were able to develop and verify a solution for trusted storage and policy enforcement that relies on a small TCB and has very little performance overhead.

We consider the system and the threat model to be relevant both for the usecases described as for any system that needs to apply policies on data.

Furthermore, we were able to validate the performance of the solution against both microbenchmarks as well as a sample usecase.

During this thesis, we were able to learn and apply techniques from several fields of computer science to converge on the overall system. We learned a lot on secure and critical systems design while planning the secure firmware. While planning and implementing it, we learned about the process of software engineering. While planning the secure channels between the SSD, the clients and the verifier, we were able to develop our understanding of cryptography and SSL channels. Then while verifying, we developed a lot of knowledge on formal verification and program analysis.

Overall this thesis also shows a methodology for developing a secure system that is both efficient and verified with very strong functionality and security guarantees. This work is a proof of concept that developing an efficient and verified secure system is possible with state of the art technology and the right methodology.

6.1. Achievements

During this thesis we were able to develop a full solution that enforces file policies at the storage layer – thus having a very small TCB –, verify some interesting properties about this solution, and validate that this system is very efficient both in terms in performance and metadata overhead.

For each objective proposed we quantify our results:

- Efficiency - the code has a worst case 20% latency overhead on the worst performing microbenchmarks
- Verification - we were able to verify 90% of the properties we specified about the code, and prove relevant functionality and security properties. Most of the things we left out are data datastructure that don't influence the evaluation
- TCB size - our TCB is basically the 8000 LoC of the firmware plus the SSL implementation, thus being small
- Metadata size - the metadata overhead is really low, being able to fit on the small DRAM of openSSD

Given the importance of enforcing policies on stored data in order to achieve more secure systems and the gaps in current solutions – both in terms of verifiable code base, as well in terms of efficient policy enforcement – this thesis shows that the development of a fully verified storage solution is possible and important.

We developed a firmware for an SSD that provides security properties, a library for clients to communicate with the firmware, and verified the security properties of this system.

We believe that our solution is be a step forward in trusted computing, as it is be the first formally verified SSD firmware to provide expressive policy enforcing mechanisms. It is also be a solution with a very small and therefore trustworthy TCB. In addition to the overall contribution of building the storage system itself, our verification effort and techniques on embedded firmware are also an interesting contribution.

6.2. Future Work

First of all, possible interesting future work could start by finishing the specification and verification of the security properties on the Batch API mode. Doing so would provide a fully verified firmware (minus the openSSL).

In this thesis, we also did not consider very deeply protection against side channel attacks, taking as prerequisite to our security properties that the information we leak is no more than that of the policy language. Therefore, further work implementing in this firmware some specific policy language that could leak information could shed led on the possibility of giving stronger guarantees against side channel attacks trough the firmware than that initially supported by the language

It would also be very interesting to implement the use cases we describe in this project and evaluate the performance of this system against it.

Furthermore, we implemented our work in a single threaded, single memory channel and very simplified SSD. Optimizing this project for multithreaded SSDs (while probably losing all the verification work, as verifying multithreaded applications is much more complex), or for SSDs with more processing power, would also be a very interesting challenge.

At last, extending this work to work seamlessly with intel SGX would also be very interesting, as we would provide a much stronger model than current SGX thrusted storage model supports.

References

[1] ARM. *ARM Security Technology Building a Secure System using TrustZone® Technology*, 2009.

[2] M. Y. Becker, C. Fournet, and A. D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4):619–665, Dec. 2010.

[3] K. Butler, S. Mclaughlin, T. Moyer, and P. McDaniel. New security architectures based on emerging disk functionality. *IEEE Micro*, 12(6):34–45, 1 2010.

[4] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[5] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association.

[6] A. Cui, M. Costello, and S. Stolfo. When firmware modifications attack: A case study of embedded exploitation. 2013.

[7] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 105–, Washington, DC, USA, 2002. IEEE Computer Society.

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

[9] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo. In *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP 17*. ACM Press, 2017.

[10] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich. Proving confidentiality in a file system using disksec. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 323–338, Carlsbad, CA, 2018. USENIX Association.

[11] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback protection for trusted execution. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1289–1306, Vancouver, BC, 2017. USENIX Association.

[12] S. Yong Ho, J. Sanghyuk, L. Sang-Won, and K. Jin-Soo. Cosmos openssd: A pcie-based open source ssd platform. In *Proc. of Flash Memory Summit*, 2014.

[13] SEAGATE TECHNOLOGY LLC. Self-encrypting hard disk drives in the data center. Technical report, TP583, 2007.

[14] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*. ACM Press, 2016.

[15] A. Vahldiek-Oberwagner, E. Elnikety, A. Mehta, D. Garg, P. Druschel, R. Rodrigues, J. Gehrke, and A. Post. Guardat. In *Proceedings of the Tenth European Conference on Computer Systems - EuroSys15*. ACM Press, 2015.

[16] P. Voigt and A. v. d. Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[17] C. Weinhold and H. Härtig. jvpfs: Adding robustness to a secure stacked file system with untrusted local storage components. *USENIX Annual Technical Conference '11*, 01 2011.

[18] Wikipedia contributors. Wikipedia traffic dumps — Wikipedia, the free encyclopedia, 2016. [Online; accessed 30-October-2019].