BesFS: Mechanized Proof of an Iago-Safe Filesystem for Enclaves

Shweta Shinde*
National University of Singapore
shweta24@comp.nus.edu.sg

Shengyi Wang*
National University of Singapore
shengyi@comp.nus.edu.sg

Pinghai Yuan
National University of Singapore
yuanping@comp.nus.edu.sg

Aquinas Hobor
National University of Singapore &
Yale-NUS College
hobor@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Prateek Saxena
National University of Singapore
prateeks@comp.nus.edu.sg

ABSTRACT

New trusted computing primitives such as Intel SGX have shown the feasibility of running user-level applications in enclaves on a commodity trusted processor without trusting a large OS. However, the OS can compromise the integrity of the applications via the system call interface by tampering the return values. This class of attacks (commonly referred to as Iago attacks) have been shown to be powerful enough to execute arbitrary logic in enclave programs. To this end, we present BesFS — a formal and provably Iago-safe API specification for the filesystem subset of the POSIX interface. We prove 118 lemmas and 2 key theorems in 3676 lines of CoQ proof scripts, which directly proves safety properties of BesFS implementation. BesFS API is expressive enough to support 17 real applications we test, and this principled approach eliminates several bugs. BesFS integrates into existing SGX-enabled applications with minimal impact to TCB (less than 750 LOC), and it can serve as concrete test oracle for other hand-coded Iago-safety checks.

1 INTRODUCTION

Existing computer systems encompass millions of lines of complex operating system (OS) code, which is highly susceptible to vulnerabilities, trusted by all user-level applications. In the last decade, a line of research has established that trusting an OS implementation is *not* necessary. Specifically, new trusted computing primitives (e.g. Intel SGX [59], Sanctum [29], PodArch [74], Bastion [19]) have shown the feasibility of running user-level applications on a commodity trusted processor without trusting a large OS. These are called *enclaved execution* primitives, using the parlance introduced by Intel SGX — a widely shipping feature in commodity Intel processors today. Applications on such systems run isolated from the OS in a region of CPU-protected memory called an enclave; the adversary model defeated by individual designs vary (see [28, 58]).

The promise of enclaving systems is to minimize the trusted code base (TCB) of a security-critical application. Ideally, the TCB can be made boiler-plate and small enough to be *formally verified* to be free of vulnerabilities. Towards this vision, recent works have formally specified and checked the interfaces between the enclave and the CPU [30, 80], as well as verified confidentiality properties of an application [77, 78]. One critical gap remains unaddressed: verifying the integrity of the application from a *hostile* OS. Applications are increasingly becoming easier to port to enclaves [13, 21, 73]; however, these legacy applications optimistically assume that the OS is benign. A hostile OS, however, can behave arbitrarily violating assumptions inherent in the basic abstractions of a process or files, and exchange malicious data with the application. This threat is well-known, originally identified by Ports and Garfinkel as *system call tampering* [65], and more recently discussed as Iago attacks [22].

A number of enclave execution platforms have recognized this channel of attack, but left specifying the necessary checks out of scope. For instance, systems such as Haven [13], Panoply [73], Graphene-SGX [21], and Scone [12] built on Intel SGX have alluded to syscall tampering defense as an important challenge; however, none of these systems claim a guaranteed defense. One of the reasons is that a hostile OS can deviate from the intended behavior in so many ways, and reasoning about a *complete* set of checks that suffices to capture all attacks is difficult.

In this work, we take a step towards a formally verified TCB to protect integrity of enclaves against a hostile OS. To maximize the eliminated attack surface and compatibility with existing OSes, we propose to safeguard at the POSIX system call interface. We scope this work to the filesystem subset of the POSIX API. Our main contribution is BesFS— a POSIX-compliant filesystem specification with formal guarantees of integrity, and a machine-checked proof of its implementation. Client applications running in SGX enclaves interact with a commodity (e.g., Linux) OS via our BesFS implementation, running as a library (see Figure 1). Applications use the POSIX filesystem API transparently (see Table 1), requiring minimal integration changes. Being formally verified, BesFS specifications and implementation can further be used to test implementations of existing platforms based on SGX and similar primitives.

Challenges & Approach. The main set of challenges in developing BesFS are two-fold. The first challenge is in establishing the "right" specification of the filesystem interface, such that it is both safe (captures well-known attacks) and admits common benign functionality. To show safety, we outline several known syscall tampering attacks and prove that BesFS interface specification defeats at least these attacks by its very design. The attacks defeated are not limited to identified list here — in fact, any deviations from the defined behavior of the BesFS interface is treated as a violation, aborting the client program safely. To address compatibility, we empirically test a number of real-world applications and benchmarks with a BesFS-enhanced system for running SGX applications. These tests show no impact on compatibility, which bolsters our claim that the BesFS specification is rich enough to run practical applications on commodity OS implementations. The BesFS API has only 13 core operations. However, it is accompanied crucially by a

* These joint first authors contributed equally to this work.

composition theorem that safeguards chaining all combinations of operations, making extensions to high-level APIs (e.g., `libc`) easy.

The second challenge is in the execution of the proof of the BesFS implementation itself. Our proof turns out to be challenging because the properties require higher-order logic (hence the need for Coq) and reasoning about *arbitrary* behavior at points at which the OS is invoked. Specifically, the filesystem is modeled as a state-transition system where each filesystem operation transitions from one state to another. A number of design challenges arise (Section 4) in handling a stateful implementation in the stateless proof system of Coq, and uncovering inductive proof strategies for recursive data structures used in the BesFS implementation. These proof strategies are more involved than those applied automatically by Coq.

Results. Our Coq proof comprises of 118 theorems and 3676 LOC while our implementation of BesFS is 1449 LOC in size. We add 724 LOC for application stubs and compatibility with enclave code. We demonstrate the expressiveness of BesFS by supporting 17 applications. We show that BesFS is compatible with state-of-the-art filesystems. It is fully compatible with a large array of benchmarks we tested. It also aids in finding implementation mistakes. We hope BesFS serves as a specification for future optimizations and hand-coded implementations to be tested against.

Contributions. We make the following contributions:

- We formally model the class of attacks that the OS can launch against SGX enclaves via the filesystem API; and develop a complete set of specifications to disable them.
- We present BesFS — a formally verified set of API implementations which are machine-check for their soundness w.r.t. API specifications. Our auto-generated run-time monitoring mechanism ensures that the runs of the concrete filesystem stay within the envelope of our specification.
- We prove 118 lemmas and 2 key theorems in 3676 lines of CoQ proof scripts and evaluate correctness, compatibility and expressiveness of BesFS over a set of 17 applications from real-world programs from SPEC CINT 2006 and filesystem benchmarks to eliminate several bugs.

2 PROBLEM

There has been long-standing research on protecting the OS from user-level applications [45]. In this work, the threat model is reversed; the applications demand protection from the OS kernel. We briefly review the specifics of Intel SGX, on which our system is built, and highlight the need for a formal approach to safety.

2.1 Background & Setup

Intel SGX provides a set of CPU instructions which can protect selected parts of user-level application logic from an untrusted operating system. Specifically, the developer can encapsulate sensitive logic inside an *enclave*. When the hardware starts to execute the enclave, it creates a protected virtual address space for the enclave. The CPU allocates protected physical memory from *Enclave Page Cache (EPC)* that backs the enclave main memory; and its content is encrypted in the main memory (RAM). Only the owner enclave can access its EPC pages at any point during execution. The hardware does not allow any other process or the OS to access or modify

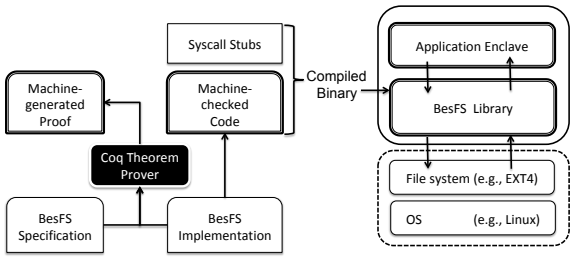

Figure 1: BesFS **Overview. Thick black and dotted outline represents trusted and untrusted components respectively.**

code and data inside the enclave's boundary. Interested readers can refer to [28] for full details.

Due to the strict memory protection, unprotected instructions such as `syscall` are illegal inside the enclave. However, the application can use *out calls (*`OCALLs`*)* to executes system calls outside the enclave. The enclave code copies the `OCALL` parameters to the untrusted partition of the application, which in turn calls the OS system call, collects the return values and passes it back to the enclave. When the control returns to the enclave, the enclave wrapper code copies the `syscall` return values such as buffers from the untrusted memory to the protected enclave memory. This mechanism facilitates interactions between the enclave and non-enclave logic of an application. Almost all enclave applications need to dispatch `OCALLs` either for standard APIs such as syscalls or for application-specific operations. To save developer time, the Intel's Software Development Kit for SGX (SGX SDK [3]) provides a boilerplate code and tools to generate the `OCALL` wrapper code. The developer can provide the type signature for the function call, and the SDK generates the wrapper code and switch-case from the type-based templates (using `Edger8r` tool [2]).

Enclaves ensure that the control from untrusted execution returning inside the enclave enters at the right entry points via the `ENCLU[EENTER]` and `ENCLU[ERESUME]` instructions [59]. This safety mechanism prevents the OS from resuming the enclave at arbitrary points and execute arbitrary sequences of logic inside the enclave. All the `OCALLs` have to use `ENCLU[ERESUME]` instruction to re-enter the enclave. For simplicity, SGX SDK consolidates all entry and exit points of the enclave into just a few selected locations. The SGX SDK internally creates a large switch-case statement for handling of different `OCALLs` and `ECALL` exits and entries respectively. To this end, the SGX SDK provides a helper function `sgx_ocall` which wraps the usage of `ENCLU [ERESUME]` instruction. It takes in the number of the `OCALL` as a parameter and uses it to select the right switch case. When the `OCALL` returns, the same number is used to un-marshal the return values.

Syscall Parameter Tampering. This is a broad class of attacks and has been inspected in various aspects by Ports and Garfinkel [65]; a specific subclass of it is called as Iago attacks [22]. Ports-Garfinkel first showed system call tampering attacks for various subsystems such as filesystem, IPC, process management, time, randomness and I/O. For file content and metadata tampering attacks, their paper suggested defenses such as maintaining protection metadata such as a secure hash for pages in the file along protected by MAC and

freshness counter stored in the untrusted guest filesystem. For file namespace management they proposed using a trusted, protected daemon to maintain a secure namespace which maps a file's pathname to the associated protection metadata. This way, verifying if OS return values are correctly computed would be easier than undertaking to compute them. An added benefit is that the TCB of such a trusted monitoring mechanism for the untrusted kernel is smaller. The recent work on Iago attacks shows a subclass of concrete attacks on these interfaces thus highlighting that verification of return values is non-trivial for complex kernel tasks such as managing virtual memory. Iago attacks demonstrate that verifying return values may require the supervisor to have a complete understanding of a kernel's memory management algorithms and data structures. In this paper, our focus is on the filesystem subset. Further, we concentrate mainly on enclave-like systems, but our work applies equally well to other systems [27, 41].

Threat to Existing Systems. Note that all systems such as Haven [13], Scone [12], Panoply [73], Graphene-SGX [21] which use either SDK or hand-code OCALL wrappers must address syscall parameter tampering attacks. All the systems are upfront in acknowledging this gap and employ ad-hoc checks for each API to address a subset of attacks. See Appendix A.1 for the informal claims made by prior works. Integrity preserving filesystems [9] and formally testing if a filesystem abides by POSIX semantics [66] are a stepping stone towards our goal, but their designs do not reason about intentional deviations by the untrusted OS.

2.2 Attacks

We demonstrate two representative attack capabilities on state-of-the-art enclave systems to motivate why a provable implementation (down to the details) is important: (a) executing arbitrary code inside the enclave using low-level memory exploits [22, 52] (b) subverting the integrity of the enclave operation by violating the high-level syscall semantics [65].

```
1 //enclave.c
2 char* buf = malloc(sizeof(char) * 100);
3 int status = ocall_fread(buf, 100, 1, fd);
4 //update buf
5 status = ocall_fwrite(buf, 100, 1, fd);
6 }
7 //ocall-helper.c
8 sgx_status_t SGX_CDECL ocall_fread(size_t* retval, void* ptr, size_t size,
        size_t nmemb, FILE ..) {
9   ...
10  ms->ms_size = size;
11  ms->ms_nmemb = nmemb;
12  ms->ms_FILESTREAM = FILESTREAM;
13  status = sgx_ocall(FREAD, ms); //FREAD is pragma for fread in switch case
14  if (retval) *retval = ms->ms_retval;
15  if (ptr) `memcpy((void*)ptr, ms->ms_ptr, _len_ptr);`
16  sgx_ocfree();
17  return status;
```

Listing 1: Intel SGX SDK's enclave OCALL mechanism.

Low-level Attacks. Listing 1 shows an example of OCALL mechanism for `fread` call generated by the Intel SGX SDK [3]. At line 13 the enclave wrapper code calls the untrusted `fread` function which executes outside the enclave. The results generated by the `fread` call are copied into the enclave on line 15. The onus of checking the buffer sizes of such untrusted OCALL return values lies on the enclave wrapper code. In our example, line 15 has a buffer overflow in the read system call because the SDK leaves such checks to be implemented by the client application by definition. As recently demonstrated, this buffer overflow can be used to corrupt the enclave stack and launch expressive attacks such as ROP on the enclave logic [52]. The OS can also leverage more sophisticated attacks such as data-corruption [42] to overwrite the OCALL number inside the enclave memory during un-marshaling. With such a corruption, the OS can fake a return from of a different system call. Once the OS jumps to the right OCALL return inside the enclave, the enclave starts executing the logic following the wrong OCALL return. In fact, the OS can chain enough of such OCALL return gadgets to program arbitrary logic, depending on client logic [43, 52].

High-level Attacks. Consider an application where the enclave is executing an anti-virus scan which white-lists user files. Listing 2 shows a code snippet of such an enclave function. It reads in the names of the suspicious files (line 4) and opens each file (line 8). The function `inspect` on line 9 then checks the signature of the file against a white-list and returns a value 0 or 1. The enclave then creates a new report file for logging results for each scanned file (line 12). If the file is marked benign the enclave writes `safe` to the `.log` file (line 13), else it writes `malicious` (line 14). The enclave is supposed to protect the signature files, ensure complete inspection of the suspicious files and safe logging of the scan results. However, a malware-infected OS might deviate from the expected filesystem semantics and cause the malware file to be falsely white-listed in the following ways:

```
1 char list[MAXBUFSIZ], logname[BUFSIZ];
2 FILE* l, fd, logname;
3 l = `fopen` ("list_of_suspicious_files", r);
4 int err = `fread` (l, list, ... );
5 if (!err) {
6   //process each new line entry in the list
7   for (f = getline(list)) {
8     fd = `fopen` (f);
9     result = inspect(fd);
10    strcpy (logname, f);
11    strcat (logname, ".log");
12    log = `fopen` (logname);
13    if (result == 0) `fwrite` (log, "safe");
14    else `fwrite` (log, "malicious");
15 }
16 else
17   //report that scan was successful
```

Listing 2: Code snippet of client enclave logic for anti-virus scanner to whitelist user files.

(A1) Paths & File Descriptor Mismatch. The OS can use the wrong file paths, wrong permissions or wrong file names. In our example, the OS can trick the enclave into believing that it is scanning a different file on line 8 by opening say a safe file `file4956` instead of a malicious `file4444`. This tricks the enclave to write the scan results of the wrong file in the log and the OS succeeds in marking the malicious `file4444` as safe. Alternatively, the OS can swap file descriptors on line 9 in order to redirect all file operations to a file of its choice. So, on line 13 – 14 instead of writing `"safe"` and `"malicious"` to `file4956` and `file4444` respectively, it can swap the descriptors. Thus the enclave ends up marking the file `file4444` as safe.

(A2) Size Mismatch. The OS can violate the size requested in the operations by increasing or decreasing the size of the buffers. For example, on line 4, instead of returning `["file4956", "file4345", "file1538"]`, the `fread` call returns `["file4956", "file4345"]` to bypass the checks for `"file1538"`.

(A3) Iago Attacks on File Content. Apart from simple parameter tampering, the OS can do subtle attacks at the memory mapping layer for file content. This includes (a) mapping multiple file blocks of the same or different files to single physical block (b) read/write content from/to the wrong offset or block (c) misalign the sequence of file blocks in a file. In our example, the OS can mark any file with any tag it wants by manipulating the file to block mapping in the above ways. If the last file to be scanned is `file4956` and it is safe, then the enclave is about to write the tag `safe` in the file `file4956.log` on line 13. At this point, the OS can map the blocks of all the `.log` files to a single physical block. Thus, when the enclave writes to `file4956.log`, the `safe` tag is written to all the `.log` files. The OS can do similarly file-to-block manipulation attacks as and when it desires to achieve arbitrary effects.

(A4) Error Code Manipulation. The OS can change the error codes returned by the filesystem and force the enclave to take a different control-flow path in its execution. In our example on line 4, the enclave checks if there was an error while reading the list of suspicious files that it wants to scan. If the enclave encounters an error, it simply reports that the scan succeeded (line 17), with zero malicious file warnings. The OS can intentionally send the error code indicating the file does not exist and thus bypass the checks from line 6 – 15. Note that this attack is more than just denial of service because the OS does return back an error value (so it does not deny the service), but it misrepresents filesystem state.

We do not claim to be the first to showcase these attacks. Further, our list of attacks is not exhaustive. They are merely a representative of the intractably large number of ways the OS can cheat, depending on the logic of the client application. This motivates a strong defense which not only strictly defines an acceptable behavior but also flags all violations as potentially dangerous.

3 BESFS **DESIGN**

All the classes of filesystem API attacks covered in Section 2.2 stem from the fact that the OS can deviate from its expected semantics. This, in turn, leads to exploitable behavior inside the enclave.

3.1 Approach

Attacks on an enclaved application can arise at multiple layers of the filesystem stack (Appendix A.2). Our choice of BesFS API to formally proof-check is guided by the observation that the higher the layer we safeguard, the *larger* the attack surface we can eliminate, and the more implementation-agnostic the BesFS API becomes. One could include all the layers including the disk kernel driver, where content is finally mapped to persistent storage, in the enclave. Enforcing safety at this interface will require simply encrypting/decrypting disk blocks with correct handling for block positions [51]. Alternatively, one could include a virtual filesystem management layer, which maps file abstractions to disk blocks and physical page allocations, in the enclave — as done in several LibraryOS systems like Graphene-SGX [13, 21]. To ensure safety at this layer, the model needs to reason about simple operations (reads, writes, sync, and metadata management). Further up, one could design to protect at the system call layer, leaving all of the logic for a filesystem (e.g., journaling, physical page management, user management, and so on) outside the enclave. However, this still includes the entire library code (e.g. the `libc` logic) which manages virtual memory of the user-level process (heap management, allocation of user-level pages to buffers and file-backed pages). This is about 1.29 MLOC in `glibc` and 88 KLOC in `musl-libc`, for instance. Once we include such a TCB inside the enclave, we either need to prove its implementation safety or trust it with blind faith. We decide to model our API above all of these layers, excluding them from the TCB.

BesFS models the POSIX standard for file sub-system. POSIX is a documented standard, with small variations across implementations on various OSes [66]; in contrast, many of the other layers do not have such defined and stable interfaces. At the POSIX layer, BesFS models the file/directory path structures, file content layouts, access rights, state metadata (file handles, position cursors, and so on). Specifically, BesFS ensures safety without the need to model virtual-to-physical memory management, storage, specifics of kernel data structures for namespace management (e.g., Linux inode, user groups), and so on. BesFS is thus generic and compatible with different underlying filesystem implementations (`NFS`, `ext4`, and so on). Further, this choice of API reduces the complexity of the proofs as they are dispatched for simpler data structures.

Solution Overview. BesFS is an abstract filesystem which ensure that the OS follows the semantics of a benign filesystem i.e., the OS is exhibiting a behavior which is observationally equivalent to a good OS. This way, instead of enlisting potentially an infinite set of attacks, we define a good OS and deviation from it is categorized as an attack from a compromised or a potentially malicious OS. Specifically, our definition of a good OS includes POSIX-compliance and a set of safety properties expected from the underlying filesystem implementation. We design a set of 13 core filesystem APIs along with a safety specification. Table 1 shows this BesFS POSIX-compliant interface, which can be invoked by an external client program running in the SGX enclave. It has a set of *methods*, *states*, and *safety properties* (SP1-SP5 and TP1-TP13) defined in Section 3.2. Each method operates on a starting state (implicitly) and client program inputs. The safety properties capture our definition of a benign OS behavior. Empirically, we show in Section 6 that the real implementations of existing OS, when benign, satisfy the BesFS safety properties — the application executes with the BesFS interface as it does with direct calls to the OS. Further, the safety properties reject any deviations from a benign behavior, which at least includes all the attacks outlined in Section 2.2.

The safety provided is proven to be compositional. First, the state safety properties (SP1-SP5) ensure that if we invoke a BesFS core API operation in a good (safe) state, we are guaranteed to resume control in the application in a good state. Second, we show that calls are chainable, i.e., the good state after a call can be used as an input to any of the BesFS calls, through a set of safe transition properties (TP1-TP13). This compositionality is crucial to allow executions of benign applications which make a potentially infinite set of calls; further, one can model higher level API (e.g. the `fprintf` interface in `libc`) by composing two or more BesFS API operations.

Scope. BesFS aims strictly at integrity property; it does not claim any guarantees about the privacy and the confidentiality of the file operations. A number of side-channels and hardware mistakes are known which impact the confidentiality guarantees of SGX [82, 84].

Out of 118 lemmas in BesFS, only one lemma assumes the correctness of the cryptographic operations. Specifically, BesFS assumes the secrecy of its AES-GCM key used to ensure the integrity of the filesystem content. Our lemma assumes that the underlying cryptography does not allow the adversary to bypass the integrity checks by generating valid tags for arbitrary messages. Further, we assume that the adversary does not know the AES-GCM key used by the enclave to generate the integrity tags.

3.2 BesFS Interface

BesFS interface is a state transition system. Specifically, it defines a set of valid filesystem states and methods to move from one state to another. While doing so, BesFS also dictates which transitions are valid by a set of transition properties.

State. BesFS has a set of type variables (denoted in sans-serif font type) which together define a state. Specifically, BesFS state comprises valid paths in the filesystem ($\mathcal{P}$), mapping from paths to file and directory identifiers and metadata ($\mathcal{N}$), a set of open files ($\mathcal{O}$) and the memory map of file content ($\mathcal{M}$).

All file and directory paths that exist in the filesystem are captured by path set $\mathcal{P}$, where `Path` represents the data type path.

$$\mathcal{P} \coloneqq \{\mathsf{p} \mid \mathsf{p} : \mathtt{Path}\}$$

A directory path type can be specifically denoted by $\mathcal{P}_{\text{DIR}}$, whereas a file path type is denoted by $\mathcal{P}_{\text{FILE}}$. We also define the Parent operator which takes in a path and returns the parent path. For example, if the path p is `/foo/bar/file.txt`, then Parent(p) gives the parent path `/foo/bar`.

BesFS captures the information about the files and directories via the node map $\mathcal{N}$. Thus, BesFS associates an identifier to each file and directory for simplifying the operations which operate on file handles instead of paths. We represent the user read, write and execute permissions by `Permission`. The size field for a file signifies the number of bytes of file content. For directories, the size is supposed to signify the number of files and directories in it. For simplicity, BesFS currently does not track the number of elements in the directory and all the size field for all the directories is always set to 0. For a path p, we use the subscript notations $\mathcal{N}(p)_{\mathtt{Id}}$, $\mathcal{N}(p)_{\mathtt{perm}}$, and $\mathcal{N}(p)_{\mathtt{Size}}$ to denote the id, permissions, and size respectively.

$$\mathcal{N} \coloneqq \mathtt{Path} \twoheadrightarrow \mathtt{Id} \times \mathtt{Permission} \times \mathtt{Size}$$

Each open file is tracked using $\mathcal{O}$ via its file id. $\mathcal{O}$ also tracks the current cursor position for the open file to facilitate operations on the file content. Given a tuple o in $\mathcal{O}$, for simplicity, we use subscript notations $o_{\mathtt{Id}}$ and $o_{\mathtt{Cur}}$ to denote the id and the cursor position of that file.

$$\mathcal{O} \coloneqq \{(\mathtt{fileId}, \mathtt{cursor}) \mid \mathtt{fileId} : \mathtt{Id}, \mathtt{cursor} : \mathbb{N}\}$$

The file content is stored in a byte memory and each byte can be accessed using the tuple file id and the specific position in the file.

$$\mathcal{M} \coloneqq \mathtt{Id} \times \mathbb{N} \twoheadrightarrow \mathtt{Byte}$$

Thus, the BesFS state $\mathsf{S}_{\text{BesFS}}$ can be defined by the tuple $\langle \mathcal{P}, \mathcal{N}, \mathcal{O}, \mathcal{M} \rangle$. The state variables cannot take arbitrary values, instead, they must abide by a set of state properties defined by BesFS. For path set $\mathcal{P}$, BesFS enforces that the entries in the path set are unique and do not contain circular paths [18, 20]. This ensures that each directory contains unique file and directory names by the definition of a path set. All files and directories in BesFS have unique identifiers and are mapped by the partial function $\mathcal{N}$ to their metadata such as permission bits and size. Formally, this is defined as:

$$\begin{gathered}\mathrm{dom}(\mathcal{N}) = \mathcal{P} \\ \forall (p, p') \in \mathcal{P} \times \mathcal{P}, p \neq p' \Rightarrow \mathcal{N}(p)_{\mathtt{Id}} \neq \mathcal{N}(p')_{\mathtt{Id}} \end{gathered} \tag{SP1}$$

All open file IDs have to be registered in the $\mathcal{O}$. $\mathcal{O}$ can only have unique entries and the cursor of an open file handle cannot take a value larger than that file's current size.

$$\forall o \in \mathcal{O}, \exists p \text{ s.t. } p \in \mathcal{P} \wedge \mathcal{N}(p)_{\mathtt{Id}} = o_{\mathtt{Id}} \tag{SP2}$$

$$\forall (o, o') \in \mathcal{O} \times \mathcal{O}, o_{\mathtt{Id}} = o'_{\mathtt{Id}} \Rightarrow o = o' \tag{SP3}$$

$$\forall p \in \mathcal{P}, o \in \mathcal{O}, \mathcal{N}(p)_{\mathtt{Id}} = o_{\mathtt{Id}} \implies o_{\mathtt{Cursor}} < \mathcal{N}(p)_{\mathtt{Size}} \tag{SP4}$$

The $\mathcal{M}$ does not allow any overlap between addresses and has a one-to-one mapping from virtual address to content. The partial function $\mathcal{M}$ ensures this by definition. All file operations are bounded by the file size. Specifically, the memory can be dereferenced only for offsets between 0 and the `EOF`. Any attempts to access file content beyond `EOF` are invalid by definition in BesFS. Similarly, the current cursor position can only take values between 0 and `EOF`. We represent such invalid memory accesses by the symbol $\perp$. Formally, this is defined as:

$$\begin{aligned}\forall \mathsf{f}, \forall \mathsf{o}, \exists p \text{ s.t. } p \in \mathcal{P} \ \wedge \ \mathsf{f} = \mathcal{N}(p)_{\mathtt{Id}} \ \wedge \ \mathsf{o} < \mathcal{N}(p)_{\mathtt{Size}} \\ \Rightarrow \mathcal{M}(f, o) \neq \perp \end{aligned} \tag{SP5}$$

State Transitions. BesFS interface specifies a set of methods listed in BesFS API in Table 1. Each of these methods takes in a valid state and user inputs to transition the filesystem to a new state. Thus, BesFS interface facilitates safe state transitions. Formally, we represent it as $\tau_{m_i}(\mathsf{S}, \mathsf{S}', \overrightarrow{\text{out}})$, where τ_{m_i} is the interface method invoked on state S to produce a new state S′. The vector $\overrightarrow{\text{out}}$ represents the explicit results of the interface. This way, BesFS enforces *state transition atomicity* i.e., if the operation is completed successfully then all the changes to the filesystem because of the operations must be reflected; if the operation fails, then BesFS does not reflect any change to the filesystem state. Formally,

$$\frac{\overrightarrow{\text{out}}_{error} \neq \mathit{ESucc}}{\mathsf{S} = \mathsf{S}'} \qquad \tau_{m_i}(\mathsf{S}, \mathsf{S}', \overrightarrow{\text{out}})$$

Safety Properties. BesFS satisfies the state properties at the initialization. This is because the start state ($\mathsf{S}_{\mathtt{init}}$) is empty. Specifically, all the lists are empty and the mappings do not have any entries. So, they trivially abide by the state properties in ($\mathsf{S}_{\mathtt{init}}$). Once the user starts interfacing with the BesFS state, we need to ensure that the BesFS state properties (SP1-SP5) still hold true. Further, each interface itself dictates a set of constraints – for example, a file should be opened first in order to close it. Thus, such interface-specific properties not only ensure that the state is valid but also specify the safe behavior for each interface. Transition properties TP1-TP13 in Table 1 specify the relation between type map, state and the state transition in BesFS.

$\mathbf{TP}_i$	BesFS **Interface**	**Pre-condition** $\mathbf{Pre}_i(,)$	**Transition Relation** $\tau_i(,,\mathbf{S}')$
TP_1	fs_close (h : Id) → (e : Error)	$\exists o,\ o_{\mathtt{Id}} = h \wedge o \in \mathcal{O}$	$\mathrm{S}' = \mathrm{S}[\mathcal{O}/\mathcal{O} - \{o\}] \wedge e = \mathtt{ESucc}$
TP_2	fs_open (p : Path) → (h : Id, e : Error)	$p \in \mathcal{P} \wedge \forall o \in \mathcal{O},\ \mathcal{N}(p)_{\mathtt{Id}} \neq o_{\mathtt{Id}}$	$\mathrm{S}' = \mathrm{S}[\mathcal{O}/\mathcal{O} + \{(\mathcal{N}(p)_{\mathtt{Id}}, 0)\}] \wedge e = \mathtt{ESucc} \wedge h = \mathcal{N}(p)_{\mathtt{Id}}$
TP_3	fs_mkdir (p : Path, r : Perm) → (e : Error)	$p \in \mathcal{P} \wedge \mathrm{Parent}(p) \in \mathcal{P}_{\mathrm{DIR}} \wedge \mathcal{N}(\mathrm{Parent}(p))_{\mathrm{W}} = \mathtt{True}$	$\mathrm{S}' = \mathrm{S}[\mathcal{P}/\mathcal{P} + \{p\}, \mathcal{N}/\mathcal{N} \bigoplus (p \mapsto \langle h, r, 0\rangle)] \wedge e = \mathtt{ESucc}$
TP_4	fs_create (p : Path, r : Perm) → (e : Error)	$p \in \mathcal{P} \wedge \mathrm{Parent}(p) \in \mathcal{P}_{\mathrm{DIR}} \wedge \mathcal{N}(\mathrm{Parent}(p))_{\mathrm{W}} = \mathtt{True}$	$\mathrm{S}' = \mathrm{S}[\mathcal{P}/\mathcal{P} + \{p\}, \mathcal{N}/\mathcal{N} \bigoplus (p \mapsto \langle h, r, 0\rangle)] \wedge e = \mathtt{ESucc}$
TP_5	fs_remove (p : Path) → (e : Error)	$p \in \mathcal{P}_{\mathrm{FILE}} \wedge \mathcal{N}(\mathrm{Parent}(p))_{\mathrm{W}} = \mathtt{True}$	$\mathrm{S}' = \mathrm{S}[\mathcal{P}/\mathcal{P} - \{p\}] \wedge e = \mathtt{ESucc}$
TP_6	fs_rmdir (p : Path) → (e : Error)	$p \in \mathcal{P}_{\mathrm{DIR}} \wedge \forall q \in \mathcal{P},\ \mathrm{Parent}(q) \neq p \wedge \mathcal{N}(\mathrm{Parent}(p))_{\mathrm{W}} = \mathtt{True}$	$\mathrm{S}' = \mathrm{S}[\mathcal{P}/\mathcal{P} - \{p\}] \wedge e = \mathtt{ESucc}$
TP_7	fs_stat (h : Id) → (r : Perm, n : String, l : $\mathbb{N}$, e : Error)	$\exists o,\ o_{\mathtt{Id}} = h \wedge o \in \mathcal{O} \wedge \exists p, \mathcal{N}(p)_{\mathtt{Id}} = h \wedge p \in \mathcal{P}_{\mathrm{FILE}}$	$\mathrm{S}' = \mathrm{S} \wedge e = \mathtt{ESucc} \wedge r = \mathcal{N}(p)_{\mathtt{Perm}} \wedge l = \mathcal{N}(p)_{\mathtt{Size}} \wedge n = \mathcal{N}(p)_{\mathtt{Name}}$
TP_8	fs_readdir (p : Path) → (l : [*String*], e : Error)	$p \in \mathcal{P}_{\mathrm{DIR}}$	$\mathrm{S}' = \mathrm{S} \wedge e = \mathtt{ESucc} \wedge \forall n \in l,\ p + n \in \mathcal{P}$
TP_9	fs_chmod (p : Path, r : Perm) → (e : Error)	$p \in \mathcal{P}$	$\mathrm{S}' = \mathrm{S}[\mathcal{N}/\mathcal{N} \bigodot (p \mapsto \langle \mathcal{N}(p)_{\mathtt{Id}}, r, \mathcal{N}(p)_{\mathtt{size}}\rangle)] \wedge e = \mathtt{ESucc}$
TP_{10}	fs_seek (h : Id, l : $\mathbb{N}$) → (e : Error)	$\exists o,\ o_{\mathtt{Id}} = h \wedge o \in \mathcal{O} \wedge \exists p, \mathcal{N}(p)_{\mathtt{Id}} = h \wedge l < \mathcal{N}(p)_{\mathtt{Size}}$	$\mathrm{S}' = \mathrm{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, l)\}] \wedge e = \mathtt{ESucc}$
TP_{11}	fs_read (h : Id, l : $\mathbb{N}$) → (b : [Byte], e : Error)	$\exists o,\ o_{\mathtt{Id}} = h \wedge o \in \mathcal{O} \wedge \exists p, \mathcal{N}(p)_{\mathtt{Id}} = h \wedge o_{\mathtt{Cur}} + l < \mathcal{N}(p)_{\mathtt{Size}}$	$\mathrm{S}' = \mathrm{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, o_{\mathtt{Cur}} + l)\}] \wedge e = \mathtt{ESucc} \wedge b = \mathcal{M}(h, o_{\mathtt{Cur}}), \ldots, \mathcal{M}(h, o_{\mathtt{Cur}} + l))$
TP_{12}	fs_write (h : Id, l : $\mathbb{N}$, b : [Byte]) → (e : Error)	$\exists o,\ o_{\mathtt{Id}} = h \wedge o \in \mathcal{O} \wedge \exists p, \mathcal{N}(p)_{\mathtt{Id}} = h \wedge l < \mathcal{N}(p)_{\mathtt{Size}}$	$\mathrm{S}' = \mathrm{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, l + b_{\mathtt{len}})\}, \mathcal{M}/\mathcal{M} \bigodot ((h, l) \mapsto b[0], \ldots, ((h, l + b_{\mathtt{len}}) \mapsto b[b_{\mathtt{len}}])] \wedge e = \mathtt{ESucc}$
TP_{13}	fs_truncate (h : Id, l : $\mathbb{N}$) → (e : Error)	$\exists o,\ o_{\mathtt{Id}} = h \wedge o \in \mathcal{O} \wedge \exists p, \mathcal{N}(p)_{\mathtt{Id}} = h \wedge l < \mathcal{N}(p)_{\mathtt{Size}}$	$\mathrm{S}' = \mathrm{S}[\mathcal{N}/\mathcal{N} \bigodot (p \mapsto \langle \mathcal{N}(p)_{\mathtt{Id}}, \mathcal{N}(p)_{\mathtt{perm}}, l\rangle)] \wedge e = \mathtt{ESucc}$

Table 1: BesFS **Interface. Method API, pre-conditions, transition relations and post-conditions.** $\mathbf{S}' = \mathbf{S}[\mathcal{K}/\mathcal{K}']$ **denotes everything in** $\mathbf{S}'$ **is the same as S, only** $\mathcal{K}$ **is replaced with** $\mathcal{K}'$**. In Column 4, the − and + symbols denote set addition and deletion operations.** $\bigoplus$ **denotes new mapping is added and** $\bigodot$ **denotes update of a mapping in relation.**

3.3 How Do Our Properties Defeat Attacks?

Our state properties in Section 3.2 and transition properties in Table 1 are strong enough to defeat attacks described in Section 2.2.

Path Mismatch (A1a). BesFS state ensures that each path is uniquely mapped to a directory or a file node. All methods which operate on paths first check if the path exists and if so is the operation allowed on that file/directory path. For example, for a method call `readdir("foo/bar")`, the path `foo/bar` may not exist or can be a file path instead of a directory path. SP1 ensures that file directory paths are distinguished, are unique and are mapped to the right metadata information. Subsequently, any queries or changes to the path structure ensure that these properties are preserved. For example, `fs_create` checks if the parent path is valid and if the file name pre-exists in the parent path. When all the pre-conditions are met, the corresponding state variables are updated (SP4).

File Descriptor Mismatch (A1b). Similar to path resolution, file descriptor resolution is critical as well. Once the file is opened successfully, all file-content related operations are facilitated via the file descriptor. BesFS ensures that the file name to descriptor mappings are unique and are preserved while the file is open. Further, BesFS maps any updates to the metadata or file content via the file descriptor in such a way that it can detect any mapping corruption attempts from the OS (SP5).

Size Mismatch (A2). BesFS's atomicity property ensures that the filesystem completely reflects the semantics of the interface during the state transition. Our file content specific operations have properties which ensure that BesFS performs the operation on the

entire size specified in the input. The post-conditions of `fs_read`, `fs_write` and `fs_truncate` reflect this in Table 1.

File Content Manipulation (A3). The unique mapping property (SP5) of $\mathcal{M}$ ensures that the OS cannot reorder or overlap the underlying pages of the file content.

Error Code Manipulation (A4). All violations of state or transition properties during the execution of the interface correspond to a specific error code. Each of these error codes distinctly represents which property was violated. For example, if the user tries to read using an invalid file descriptor, the SP3 and TP11 properties are violated and BesFS return an `eBadF` error code. All error types in BesFS map to standard error codes in the POSIX API specification. If there are no violations and the state transition succeeds, BesFS returns the new filesystem state and a `ESucc` error code. Since BesFS interface performs its own checks to identify error states, so the enclave does not rely on the OS to return the right error codes. This way, we ensure that the OS cannot manipulate the enclave logic by returning wrong error codes.

3.4 BesFS Implementation

BesFS defines a collection of data structures that are sufficient to capture the filesystem state and completely model the interfaces in Section 3.2. We build BesFS types by using pre-defined types built from `ascii`, `list`, `nat`, `bool`, `set`, `record`, `string`, `map` and by composing or induction over one or more types in standard Coq libraries. We give their simplified definition below.

$$\mathsf{f} \coloneqq \mathbb{N} \qquad \mathsf{d} \coloneqq \mathbb{N}$$
$$\mathsf{Pg} \coloneqq [\mathsf{Byte}]_{\text{Pg\_Size}} \qquad \mathsf{Pmn} \coloneqq \mathsf{W} \times \mathsf{R} \times \mathsf{E}$$
$$\mathsf{Mta} \coloneqq \mathsf{Pmn} \times \mathbb{N} \qquad \mathsf{PgId} \coloneqq \mathbb{N}$$
$$\mathsf{Fda} \coloneqq \mathtt{Str} \times \mathsf{Mta} \times [\mathsf{PgId}] \qquad \mathsf{Dda} \coloneqq \mathtt{Str} \times \mathsf{Mta}$$
$$\mathsf{T} \coloneqq \textsc{File}\colon \mathsf{f} \mid \textsc{Dir}\colon \mathsf{d} \times [\mathsf{T}] \qquad \mathsf{O} \coloneqq [\mathsf{f} \times \mathbb{N}]$$

All files and directories in BesFS have ids f and d respectively. These ids are mapped to the corresponding file and directory nodes Fda and Dda. Specifically, Fda stores the file name, permissions and all the pages that belong to this file and the size of the file; Dda stores the directory name, permission bits, and the number of files and directories inside it. The BesFS filesystem layout T stores the f and d in a tree form to represent the directory tree structure. The list of open file handles O stores tuples of f and cursor position. Lastly, each page is a sequence of Pg_Size bytes which is typical size of a page [1] and has a unique page number PgId. Finally, the entire filesystem memory is stored as a list of pages v. In summary, BesFS implementation represents its filesystem state as below:

$$\mathsf{Fsys} \coloneqq (\mathsf{t} : \mathsf{T}, \mathsf{h} : \mathsf{O}, \mathsf{v} : [\mathsf{Pg}], \mathsf{F} : \mathsf{f} \rightarrow \mathsf{Fda}, \mathsf{D} : \mathsf{d} \rightarrow \mathsf{Dda})$$

Our BesFS implementation must satisfy the state properties SP1-SP5 and transition properties TP1-TP13 we outlined in Section 3.2. We discuss how we achieve this for each data structure. Table 2 summarizes the invariants for our data structure implementation.

Virtual Memory ($\mathcal{M}$). The filesystem memory is represented by a set of virtual memory pages such that each page is a sequence of Pg_Size bytes and is represented by a unique page id PgId. Unallocated pages are marked as free in the pool. Each file comprises an ordered sequence of pages allocated from a pool of free pages. One page can belong only to a single file. This ensures that no two files have overlapping page memory.

[1] We set the page size (Pg_Size) to 4096 bytes.

Virtual Memory	$\mathcal{M}$	$\forall i\, j,\ i \neq j \Rightarrow \mathsf{F}(i)[2] \cap \mathsf{F}(j)[2] = \emptyset$
Files & Directories	$\mathcal{N}$	$\textsc{Fids}(\textsc{File}\colon i) \coloneqq [i]$ $\textsc{Fids}(\textsc{Dir}\colon i\ s) \coloneqq \textsc{Fids}(s[1]) + \cdots + \textsc{Fids}(s[n])$ $\textsc{Dids}(\textsc{File}\colon i) \coloneqq []$ $\textsc{Dids}(\textsc{Dir}\colon i\ s) \coloneqq [i] + \textsc{Dids}(s[1]) + \cdots + \textsc{Dids}(s[n])$
Layout & Paths	$\mathcal{P}$	$\textsc{TreeName}(\textsc{File}\colon i) \coloneqq \mathsf{F}(i)[0]$ $\textsc{TreeName}(\textsc{Dir}\colon i\ s) \coloneqq \mathsf{D}(i)[0]$ $\text{NoDupName}(t : \mathsf{T}) \coloneqq \exists\, i, t = \textsc{File}\colon i \vee$ $\exists\, d\, s, t = \textsc{Dir}\colon d\ s \wedge (\forall i,\ \text{NoDupName}(s[i])) \wedge$ $(\forall i j,\ i \neq j \Rightarrow \textsc{TreeName}(s[i]) \neq \textsc{TreeName}(s[j]))$ $\text{NoDup}([\ldots s_i \ldots s_j \ldots]) \coloneqq \forall i\, j,\ i \neq j \Rightarrow s_i \neq s_j$
Open file handles	$\mathcal{O}$	$\textsc{Ids}([\ldots, (f_i, p_i), \ldots, (f_j, p_j), \ldots] : \mathsf{O}) \coloneqq [\ldots, f_i, \ldots, f_j, \ldots]$ $\text{NoDup}(\textsc{Ids}[\ldots s_i \ldots s_j \ldots]) \coloneqq \forall i\, j,\ i \neq j \Rightarrow s_i \neq s_j$

Table 2: BesFS **data structures invariants from Section 3.4.**

Files & Directories ($\mathcal{N}$). Each file's information including file name, the current size of the file, permission bits of the file is stored in a file node Fda. Each file's content is a sequence of bytes, partitioned into uniformly sized pages. This content is tracked by keeping an ordered list of virtual memory page ids [PgId]. For example, the first id in a file node's page list points to the exact page in the virtual memory where the first n bytes of the page are stored. BesFS also maintains a map F which associates each file node Fda with a unique file identifier f. Similar to file nodes, BesFS also has directory nodes Dda to track directory information such as names and permissions. Each directory is associated with a unique directory id d. The directory map D tracks the one-to-one relationship between ids and nodes.

Layout & Paths ($\mathcal{P}$). BesFS tracks the paths for all files and directories via a tree layout T. Each node in the tree can be a file node id f or a directory node id d. Files are leaf nodes. On the other hand, each directory, in turn, can have its own tree layout. Note that BesFS does not allow cycles in the tree layout. Also, each level of the layout tree has non-duplicate directory and file names.

Open File Handles ($\mathcal{O}$). Each open file has a file handle which is allocated when the file is first opened. The file handle comprises the file id f and the current cursor position for that file. BesFS tracks all the list of open files via the open file handles list O. All operations on an open file are done via its file handle. When the file is closed, the file handle is removed from the list. Further, the O list cannot have any duplicate f because each open file can have only one handle.

Error Codes. In cases where the filesystem cannot complete the operation successfully, the enclave should receive the right error code to know the exact reason for failure. BesFS models a subset of 15 error codes as specified by POSIX. This ensures that the attacker cannot alter the enclave's behavior by via error codes.

Good State. BesFS must satisfy all the data structure invariants in Table 2 before and after any interface invocation to be in a good state. We can summarize a state as good if the following holds true:

$$\text{NoDupName(t)} \wedge \text{NoDup}(\textsc{Fids}(\text{t})) \wedge \text{NoDup}(\textsc{Dids}(\text{t})) \wedge$$
$$\text{NoDup}(\textsc{Ids}(\text{h})) \wedge \exists\, d\, s \text{ s.t. } \text{t} = \textsc{Dir}\colon d\ s \wedge$$
$$\forall i\, j,\ i \neq j \Rightarrow \text{F}(i)[2] \cap \text{F}(j)[2] = \emptyset$$

Known Limitations. BesFS implementation does not support a small set of filesystem operations such as symbolic links, file-backed mapping, shared files, and rename because they violate our safety properties. We have consciously decided to not support these functionalities in our first version of BesFS to maintain simplicity. However, there is no fundamental limitation in extending BesFS specification and proofs to a broader set of file operations in the future.

4 BESFS SAFETY PROOF

The key theorems for our BesFS implementation are that the functions meet our interface specifications. For each method of our interface, we must prove that the implementation satisfies the state properties (SP1-SP5) from Section 3.2 and the transition properties (TP1-TP13) outlined in Table 1. We assume BesFS is running on a hostile OS that can take any actions permitted by the hardware.

Theorem 4.1 (State Transition Safety). *Given a good state S satisfying pre_i, then if we execute f_i to reach state S', then S' is always a good state and relation between S and S' is valid according to the transition relation τ_i:*

$$\forall S, S', i.\ S \models \mathit{SP1–SP5} \ \wedge\ \mathit{pre}_i(S) \ \wedge\ S \overset{f_i}{\rightsquigarrow} S' \ \Rightarrow$$
$$\tau_i(S, S') \ \wedge\ S' \models \mathit{SP1–SP5}$$

We can verify sequences of calls to our functions by inductively chaining this theorem. Our second theorem states that the state property is preserved for a composition of any sequence of interface calls. We close the proof loop with induction by starting in a good initial state and using Theorem 4.1 to show that a method invocation in BesFS always produces a good state for a sequential composition of transitions. The proof is dispatched using the Coq proof assistant.

Theorem 4.2 (Compositional Safety). *Given a good initial state S_0 subject to a sequence of transitions $\tau_{m_1}, \ldots, \tau_{m_n}$ always produces a good final state S_n:*

$$S_0 \models \mathit{SP1–SP5} \wedge S_0 \overset{f_{m_1}}{\rightsquigarrow} S_1 \wedge S_1 \overset{f_{m_2}}{\rightsquigarrow} S_2 \wedge \cdots \wedge S_n \overset{f_{m_n}}{\rightsquigarrow} S_n \Rightarrow$$
$$\tau_{m_1}(S_0, S_1) \wedge \tau_{m_2}(S_1, S_2) \wedge \ldots \wedge \tau_{m_n}(S_{n-1}, S_n) \wedge$$
$$S_n \models \mathit{SP1–SP5}$$

4.1 Coq Proof Assistant

As one can readily see, our implementation uses recursive data structures, and its state properties require second-order logic. For example, in the BesFS filesystem layout T in Section 3.4 is defined represented mutually recursively in terms of a forest (a list of trees). This motivates our choice of Coq, an interactive proof assistant supporting calculus of inductive constructions. Coq allows the prover to write definitions of data structures and interface specification in a language called Gallina, which is a purely functional language. The statements of the theorems are written in Galina as well. The proofs of the statement, called *proof scripts* are written in a language called Ltac. Ltac makes writing proofs less tedious as it supports a library of "tactics", or one-line commands that encode standard proof strategies.

The Coq system performs two operations with proof scripts. First, it mechanically checks that the proof script entails the statement of the theorem. If the proof cannot go through, it interacts with the prover by showing parts of the proof that are not complete as "holes", prompting the human prover to provide a proof script for each hole. Second, after the entire proof is checked, the proof script is converting to a Gallina program. The type of that program is the statement of the theorem. Coq proof system embodies the Curry-Howard correspondence between typing and programming, enabling rich statements to be written as mathematic types [64].

Gallina. Gallina is a functional programming language similar to OCaml and Haskell. The following code listing shows Gallina code snippet for the implementation of `write` method. It starts with the keyword `Definition`. It can be split into two parts: the signature (arguments and return type separated by colon) before `:=` and the body after `:=`. Most part of the code is self-explanatory; so, we turn attention to specific features of Gallina for readers. In line 2, the return type `State FSState ErrCode` indicates that we adopt the state monad to ease the state passing coding style [63, 81]. Lines 3 and 13 can be seen as getting and setting actions of the state. They are classical syntactic sugar in monadic style programming: the `fs` is not a variable but an argument representing the state. On line 10, the action `externalCall` actually does 4 tasks: increase the global counter retrieved from the state, perform the external call with the global counter as the additional argument, append logs to the state and put back the new counter to the state.

```
1 Definition fs_write (fId: Fid) (buf: string)
2   (pos: nat) : State FSState ErrCode :=
3  do fs <- getFS;
4  let opos := find (fun x => (fst x) =? fId) fs.(open_handles) in
5  match opos with
6  | None => return_ eBadF
7  | Some _ => ...
8    do err <- externalCall (Call_VLSeek fId (pos_to_vpage pos)) (v_lseek fId
           (pos_to_vpage pos));
9    if (isNotSucc err) then return_ err else ...
10   putFMap ... >>
11 end.
```

Ltac **Language.** Ltac allows the programmer to write lemmas, which have a representation in Gallina. In the following code listing, we can see the statement is written in Gallina, which actually declares a program whose type is the statement of the lemma. The script called tactics between `Proof` and `Qed` is written in Ltac. During the interactive development, human provers can see the effect of each tactic on the proof goals and finally prove it by trial and error. From the perspective of Coq, those tactics guide it to construct a program written in Gallina, then Coq will check whether its type is the statement after `Lemma`, we call this step as "mechanized verification". We also prove helper lemmas to simplify proofs.

```
1  Lemma fs_write_ok: forall fId buf pos fs err fs',
2    (err, fs') = fs_write fId buf pos fs -> good_file_system (fsFS fs) ->
3    (~ In fId (map fst (openhandleFS fs)) /\ ...) \/ ...
4    (... /\ In (fId, ...) (openhandleFS fs') /\
5    (forall id, id <> fId -> (fMapFS fs) id = (fMapFS fs') id) /\ ...)
6  Proof.
7    intros. unfold fs_write in H. ...
8    - right. rm_hif_eqn H. 1: left; ...
9      + right. destruct p0 as ...
10   - left. inversion H. intuition..
11 Qed.
```

4.2 Challenges

Purely Functional. The programming language provided by Coq is purely functional, having no global state variables. However, the filesystem is inherently stateful. So, we use *state passing* to bridge this gap. The state resulting from the operation of each method is explicitly passed as a parameter to the next call. If we explicitly pass these state in each call, it is prone to clutter and accidental omission; therefore, we define them as a monad. As we can see in the definition of `fs_write`, the code is purely functional but it looks like the traditional imperative program. The benefit of this monadic style programming is that it hides the explicit state passing, which makes the code more elegant and less error-prone.

While proof script checking, if Coq encounters a memoized expression for $f(z)$, it will skip proving $f(z)$ again. This is a challenge because in a sequence of system calls the same call to f with identical arguments may return different values. Therefore, we have to force Coq to treat each call as different. To implement this, we introduce an implicit counter as an argument to all calls, which increments after each call completes. For example, consider the consecutive external calls `read_dir`, `create_dir`, and `read_dir`. The two `read_dir` commands may read the same directory (the same argument) but with different return values because of the `create_dir` command. To reason about such cases, the real arguments passed to the external calls contain not only common arguments but also an ever-increasing global counter. Thus, in our read_dir example, the two commands with original argument p will be represented as read_dir(p, n) and read_dir($p, n+1$) so that Coq will treat them as the different commands.

Atomicity. The purely functional nature of Coq proofs helps to prove atomicity of each method call. In an enclave, the internal state of the enclave is not accessible by the OS; so, in a way, the enclave behaves as a pure function between two OS calls. This allows us to prove atomicity directly. We structure the proof script to check if an error state is reachable from the input state and the OS returned values; if so, the input state is retained as the output state. If no error is possible, the output state is set to the new state. For concrete illustration, the write method illustrates it progressively checks 5 conditions (1: Argument id is in the handler. 2: The specified position is correct. 3: It writes to the copied virtual memory successfully. 4: The external call to `seek` succeeds. 5: The external call to `write` succeeds.) before changing the state.

Non-deterministic Recursive Termination. Gallina's consistency guarantees that any theorem about a Gallina program is consistent, i.e., it cannot be both proven and disproved. Further, all programs in Gallina must terminate, since the type of the program is the statement of a theorem [2]. Coq uses a small set of syntactic criteria to ensure the termination. Gallina's termination requirement poses challenges for writing a BesFS implementation, which uses recursive data structures. In most cases, the termination proof for BesFS properties are automatic; however, for a small number of properties, we have to provide an explicit termination proof. For instance, the `write_to_buffer` does not admit a syntactic check for termination, as there is a recursive call. To prove termination, we show via induction that size of the input buffer strictly reduces for each invocation of write. Effectively, we establish that there are no infinite chains of nested recursive calls for that program.

Mutually Recursive Data Structures. Most of our data structure proofs are based on the induction principle, and Coq always provides an induction scheme for each inductively declared structure. The automatically generated induction scheme from Coq is *not* always strong enough to prove for some of our properties. Specifically, a key data structure in our design is a tree, the leaves of which are a list of trees — this represents the directory and file layouts (Section 3.2) — is the case. We provide an inductive statement `Tree_ind2` that is stronger than Coq-provided induction scheme `Tree_ind`, shown in the following listing. `Tree_ind` is correct but useless. We dispatch the proof by the principle of strong induction, which is `Tree_ind2`. Our induction property uses Coq's second-order logic capability, as the following code listing shows that the sub-property `P` is an input argument to the main property. A number of specific instances of properties instantiate `P` in our full proof.

```
1 Tree_ind: forall P : Tree -> Prop,
2   (forall f : Fid, P (Fnode f)) -> (forall (d : Did) (l : list Tree),
3   P (Dnode d l)) -> forall t : Tree, P t
4 Tree_ind2: forall P : Tree -> Prop,
5   (forall f : F, P (Fnode f)) -> (forall (d : Did) (l : list Tree),
6   Forall P l -> P (Dnode d l)) -> forall t : Tree, P t
```

External Calls to the OS. In our proof, we assume that calls to the OS always terminate to allow Coq to provide a proof. If the call terminates, the safety is guaranteed; the OS can, of course, decide not to terminate which constitutes as a denial-of-service attack.

Odds & Ends. Out of 118 lemmas, 75 of them are proved using inductions while the resort of them are proved by logical deductions. There are two kinds of inductions in our proofs: strong induction and weak induction. Their difference is the proof obligation. For example, in weak induction we need to prove "if $P(k)$ is true then $P(k+1)$ is true" while in strong induction it is "if $P(i)$ is true for all i less than or equal to k then $P(k+1)$ is true". Our customized induction principle for `Tree` is a typical strong induction. In all, we proved 75 lemmas by induction of which 39 are by strong induction and the rest 36 are by weak induction.

We do not implement the function `get_next_free_page` but enforce that an implementation must satisfy the property that the new page allocated by `get_next_free_page` is not used for existing files and is a valid page (less than the upper bound limit). Similarly, for functions `new_fid` and `new_did` we enforce the new ids are unique to avoid conflict which is formally stated as `new_fid(t)` $\notin$ Fids(t) and `new_did(t)` $\notin$ Dids(t) respectively. Note that we only give a specification for allocating new pages and ids for files and directories because we do not want to restrict the page management and namespace management algorithm. This way, the implementation can use a naive strategy of just allocating a new id/page for each request, employ a sophisticated re-use strategy to allocated previously freed ids, or use temporal and spatial optimizations for page allocation as long as they fulfill our safety conditions.

5 Coq TO EXECUTABLE CODE

BesFS's Coq definitions and proof script comprise 3676 LOC with 118 lemmas and 2 main theorems [3]. The development effort for

[2] A non-terminating program such as `let`$f(x) := f(x)$ has an arbitrary type, and hence any theorem is valid about it.

[3] BesFS will be released at https://github.com/shwetasshinde24/BesFS

Component	Language	LOC	Size (in KB)
Machine-proved Implementation			
Coq definitions & Proofs	Gallina	3676	1757.38
Hand-coded Implementation			
Implementation	C	863	172.39
External Call Interface	C	469	201.55
SGX Utils	C	117	667.04
Total		**1449**	**1040.98**

Table 3: LOC for various components of BesFS.

BesFS was approximately one year man hours for designing the specifications and proving them. Our proofs are complete without any unproven axioms. Coq implementation has been machine checked to prove the safety theorems. But we cannot execute the Coq code directly inside the enclave. Currently, Coq supports automatic extraction to OCaml, Haskell, and Scheme [6]. In our first round of evaluation, we extracted our Coq code to Haskell and compiled it with GHC along with wrappers to tunnel the syscalls to the underlying untrusted OS. This setup ran out of the box in a non-SGX environment. However, we failed to execute our Haskell compiled binary implementation inside SGX using existing systems (e.g., Graphene-SGX [21] or Panoply [73]). Our investigation shows that Graphene-SGX cannot support a simple hello-world Haskell binary. This is because Graphene-SGX does not support a set of syscalls (`create_timer`, `set_timer`, `delete_timer`) used by the Haskell runtime. We attempted to add support for these system calls, but they depend on `sigaction` handling which is not yet supported in Graphene-SGX. We ran into similar problems with OCaml implementation of BesFS. Currently, no other publicly available system supports Haskell, OCaml or Scheme run-time inside SGX. In fact, all the current public system for SGX only support C code. Thus, we have resorted to manual extraction from Coq-to-C. We first convert Coq implementation to C manually by hand-coding line by line from Coq-to-C. Our C implementation comprises of 863 LOC core logic and 586 LOC helper functions, totaling 1449 LOC. Our Coq code leaves out the implementation of untrusted POSIX calls. While executing the code inside the enclave, these calls have to be redirected to an actual filesystem provided by the OS.

Our implementation can be integrated with any SGX framework [12, 21, 73]. We tested Graphene-SGX as our first choice for integration and checked if it can execute our unmodified benchmarks inside an enclave. However, Graphene-SGX segfaults on our a large subset of our benchmark. Next, we chose Panoply as our underlying enclave-execution system [73]. We tunnel the POSIX calls from enclave to the untrusted environment using Panoply's `OCALL` interface. By default, Panoply converts the application call arguments to its own representation, makes the `OCALL` and converts the return values to the data type expected by the application. For example, Panoply has an internal representation of file descriptors and directory descriptors. But the actual `libc` API invoked by the application and implemented in the external `libc` library use file pointers (`FILE*`) or integers for descriptors. Panoply maintains a mapping between its own representation and the `libc` descriptors. For adding BesFS support, we wrap the application calls and marshal its arguments to make them compatible with BesFS interface described in Section 3.2. Once Panoply collects the return values from the external `libc` call, we unmarshal the return values and give it back to BesFS. Our wrapper then performs its checks on the return values and converts back the results to a data type expected by the application. If BesFS deems the results as safe, we return the final output of the API call to the application. Otherwise, we flag a safety violation. We add a total of 724 LOC to the Panoply code-base, which is within the realm of auditing. Readers can refer to Appendix A.3 for the detailed breakdown of LOC.

Future work can certify the process of creating machine code from our implementation. Existing certified compilers do not support extraction from Coq to enclave executable code; however, a roadmap to this feasibility is discussed in Appendix A.4.

6 EVALUATION

Our evaluation goal is to demonstrate the following:

- BesFS safety definition is compatible with the semantics of POSIX APIs expected by benign applications.
- Our API has the right abstraction and is expressive enough to support a wide range of applications.
- The bugs uncovered in our implementation due to BesFS formal verification efforts.
- BesFS can be integrated into a real system.

Experimental Setup. All our experiments were conducted on a machine with Intel Skylake i7-6600U CPU (2.60GHz, 4 cores) with 12GB memory and 128MB EPC of which 96MB is available to user enclaves. We execute our benchmark on Ubuntu 14.04 LTS with Linux Kernel 4.2. We use Panoply to run our benchmarks in an enclave, which internally uses Intel SGX SDK Linux Open Source version 1.6 [4]. Our system uses ext4 [1] as the underlying POSIX compliant filesystem for our experiments.

Benchmarks. We use the benchmark suite from FSCQ [25] — a filesystem written and verified in the Coq proof assistant for crash tolerance. It comprises applications to test each system call and different sequences of filesystem operations on large and small files. For testing BesFS on real-world applications, we use programs from SPEC CINT2006 [5]. Panoply's available case studies do not include any of our benchmarks. So we port all 10 of our target benchmarks to Panoply successfully. However, for our CPU bound benchmarks, we were able to port 7/12 programs from SPEC. We were unable to port the rest of the benchmarks because some programs from SPEC (`omnetpp`, `perlbench`, `xalancbmk`) use non-C APIs which are not supported in Panoply. Other limitations such as lack of support for `longjmp` in Panoply's SDK version prevent us from running the `gobmk` and `gcc` programs. Our final evaluation is on a total of 17 applications: 7 programs from SPEC and 10 programs from FSCQ.

6.1 Expressiveness & Compatibility

BesFS maintains compatibility with 99.26% of the filesystem API calls in our benchmarks. We empirically demonstrate that if the underlying filesystem and OS are POSIX compliant and benign then BesFS is not overly restrictive in the safety conditions. We first analyze all filesystem `libc` calls made by our benchmarks for various workloads using `strace` and `ltrace` respectively. We then filter out the fraction of calls related to filesystem. Table 4 shows the statistics of the type of filesystem call and its frequency for

Libc Calls	SPEC CINT 2006							FSCQ			Total
	astar	mcf	bzip2	hmmer	libqu	h264	sjeng	single	small	large	
BesFS **Core Calls**											
close	3	0	5	0	0	4	0	5	4	2	23
open	6	0	5	0	0	2	0	6	4	2	25
mkdir	0	0	0	0	0	0	0	0	1	0	1
remove	0	0	6	4	0	0	0	0	0	0	10
stat	0	0	0	1	0	0	0	1	0	0	2
chmod	0	0	1	0	0	0	0	0	0	0	1
lseek	0	0	0	0	0	6	0	0	0	4	10
read	33	0	1	0	0	3	0	1	2	3	43
write	0	0	3	0	0	4	0	2	2	2	13
BesFS **Auxiliary Calls**											
fread	0	0	3	68	12	1	0	0	0	0	84
fscanf	12	0	0	0	0	9	0	0	0	0	21
fwrite	0	0	4	84	1	4	0	0	0	0	93
fprintf	0	5	89	304	3	308	13	0	1	22	745
fopen	1	2	10	23	2	19	3	0	0	0	60
fseek	0	0	0	11	0	2	0	0	0	0	13
rewind	0	0	1	7	0	0	0	0	0	0	8
ftell	0	0	0	4	0	1	0	0	0	0	5
fgetc	0	0	2	0	1	0	0	0	0	0	3
fgets	0	3	0	47	0	0	0	3	0	0	53
Unsafe Calls											
fsync	0	0	0	0	0	0	0	0	0	2	2
rename	0	0	0	1	0	0	6	0	0	0	7
Total	55	10	130	554	19	363	22	18	14	37	1222

Table 4: Frequency of filesystem calls. Rows 3 – 11 **and** 13 – 22 **represent the frequency of core and auxiliary calls supported by** BesFS **respectively. Rows** 24–26 **show the frequency of unsafe calls for each of our benchmarks.**

Libc API	LOC	BesFS **Core API used for composition of LibC API**												
		fstat	read	open	close	seek	create	mkdir	rmdir	remove	chmod	readdir	truncate	write
read	7		✓											
fread	25		✓											
fscanf	34		✓											
fwrite	12	✓												✓
write	20	✓												✓
fprintf	15	✓												✓
fopen	78	✓		✓		✓	✓						✓	
open	60	✓		✓		✓	✓						✓	
fclose	9				✓									
close	17				✓									
fseek	31	✓				✓								
lseek	39	✓				✓								
rewind	5					✓								
creat	30			✓			✓							
mkdir	25							✓						
unlink	21									✓				
chmod	23										✓			
ftruncate	5												✓	
ftell	12	✓												
fgetc	9		✓											
fgets	25		✓											
readdir	10											✓		

Table 5: Expressiveness of BesFS**. Row represents a `libc` file system API used by our benchmarks. Column** 2 **represents the LOC added to implement the `libc` API. Columns** 3 – 15 **represent the** 13 **core APIs supported by** BesFS**. A ✓ in a cell represents that the** BesFS **API is used to compose `libc` API.**

each of our 17 benchmarks. We observe a total of 1222 filesystem calls comprising of 21 unique APIs. BesFS can protect 1213/1222 of these calls. Table 5 shows how we support the remaining 12 calls by composing using BesFS's API.

Compositional Power of BesFS. BesFS directly reasons about 9 calls using the core APIs outlined in Section 3.2. We use BesFS's composition theorem and support all 21 set of auxiliary APIs that have to be intercepted such that BesFS checks all the file operations for safety. For example, `fgets` reads a file and stops after an EOF or a newline. The read is limited to at most one less character than `size` parameter specified in the call. We implement `fgets` by using BesFS's core API for read (see Table 5). Since we do not know the location of the newline character, we read the input file character-by- character and stop reading only when we see a new line, end of file or the buffer size reaches the value `size`. Similarly, when writing the content to the output file we already know the total size of the buffer (e.g., after resolving the format specifiers in `fprintf`) thus we write the complete buffer in one single call. Many of the `libc` calls allow the application to specify flags in order to decide what all operations the API must perform. For example, the application can use the `fopen` API to open the file for writing. If the application specifies the append flag (`"a"`), the library will create the file if it does not exist, and position the cursor at the end of the file. To achieve the same functionality using BesFS, we first try to open the file, if it fails with an `ENOENT` error, we check if the parent directory exists. If so, we first create a new file. If the file exists, we open the file and then explicitly seek the cursor to the end of the file. Thus, even if there exists a one-to-one mapping from BesFS to `libc` APIs, we still have to use multipleBesFS APIs to realize the semantics of various modes/flags supported by `libc`. We implement and support a total of 16 flags in total for our 3 APIs which require flags. Note that our implementation currently supports only the common flags used by applications. However, the support can be extended to other flags if necessary for an application.

BesFS does not reason about the safety of the remaining 2 APIs which amount to a total of 9 calls in our benchmarks. Although BesFS does not support these unsafe calls, it still allows the enclave to perform those calls. Only 4/11 of our benchmarks invoke at least one unsafe API. Importantly, these unsupported calls do not interfere with the runs in our test suite and do not affect our test executions. By the virtue of BesFS's atomicity property, synchronization calls `sync/fsync/fdatasync` have to be implicitly invoked for the OS after each function call to persist the changes by each call. We experimentally confirm that the program produces the same output with and without BesFS, thus reaffirming that we do not alter the program behavior because of our safety check.

6.2 Do Proofs Help in Eliminating Bugs?

We encountered many mistakes that our proof eliminates during the development process as a part of our proof experience. They highlight the importance of a machine-proved implementation.

Example 1: Seek Specification Bug. In at least two of our functions, we need to test whether the position of the current cursor is within the range of the file, in other words, less than the length of the file. If the cursor is beyond the scope of a specific file, any further operation such as read or write is illegal. In the early versions of our Coq implementation, we simply put "`if pos < size`" as a judgment. But during the proof, we found we cannot prove certain assertions because we ignore the corner case: when the file is just created with 0 size, the only valid position is also 0. In this sense, the proof helped us to find a bug.

Example 2: Write Implementation Bug. The function `write` in BesFS takes in `pos` as an argument, which represents the position at which the buffer is to be written. In our initial Coq implementation of `write`, we were using the name `pos` for the cursor stored in the open handles ($\mathcal{O}$). Thus, we had two different variables being referred to by the same name. As a result, the second variable value (the cursor) shadowed the write position. Due to this bug, our Coq implementation of `write` was violating the specification for the

argument pos. We uncovered it when our proof was not going through. However, once we fixed the bug by renaming the input argument, we were able to prove the safety of write.

Example 3: Panoply & Intel SGX SDK Overflow Bugs. When Panoply makes fread and fwrite calls, it passes the size of the buffer and a pointer to the buffer. The default Intel SDK generated code is then responsible for copying the buffer content from the enclave to the untrusted part for write or the other way around for fread. BesFS piggybacks on the Panoply calls to read and write encrypted pages. While integrating BesFS code in Panoply, our integrity checks after read/write calls were failing. On further inspection, we identified stack corruption bugs in both fread and fwrite implementations of Panoply. Specifically, if the buffer size is larger than the maximum allowed stack size in the enclave configuration file (greater than 64KB in our experiments), even if we pass the right buffer size, the enclave's stack is corrupted. To fix this issue, we changed the SDK code to splice the buffer into smaller sizes (less than 64KB) to read/write large buffers. After our fix, the implementation passed BesFS checks.

Example 4: Panoply Error Code Bugs. POSIX specification for fopen call states that the function shall fail with error code ENOENT if a component of the filename does not name an existing file or filename is an empty string. When we used Panoply's fopen interface to tunnel BesFS's open call, Panoply did not return the expected error code when the file did not exist. BesFS checks after the external call flagged a warning of a safety condition violation. This was because BesFS did not have a record of this file but the external call claimed that the file existed. We investigated this case and discovered that Panoply had a bug in its errno passing logic. In fact, on further testing of other functions using BesFS, we found 7 distinct functions where Panoply's error codes were incorrect.

6.3 Performance

BesFS is the first formally verified filesystem for SGX and performance is not our primary goal. Future optimizations can use BesFS API as an oracle for golden implementation. For completeness of the paper, we report our preliminary performance measurements. We observe average overhead of 16.22% for the 7 SPEC CINT2006 benchmarks with our highly unoptimized implementation. For the I / O intensive benchmarks the overhead is larger. Interested readers can refer to Appendix A.5 for more details. There is ample scope for SGX optimization using well-known techniques discussed in the previous literature [12, 62, 83]. We outline a set of 5 optimization strategies in Appendix A.6 for interested readers.

7 RELATED WORK

SGX Attacks & Defenses. BesFS reasons about the integrity of its filesystem APIs and relies on SGX's integrity guarantees from the hardware. It makes an assumption on the confidentiality properties of SGX only in one of its lemmas, assuming secrecy of a cryptographic key. This design choice is an important one in light of the many side-channels that have been discovered on the SGX platform [16, 17, 26, 35, 38, 53, 57, 60, 70, 72, 82, 84] and more recently hardware mistakes in speculative execution [49, 56]. BesFS assumes that the hardware is securely implemented, and is agnostic to the defenses the enclave might deploy for ensuring confidentiality [15, 32, 36, 50, 67, 71, 79], on top of BesFS integrity properties.

Filesystem Support in SGX. Ideally, the enclave should not make any assumptions about the faithful execution on the untrusted calls and should do its due diligence before using any (implicit or explicit) results of each untrusted call. The effects of malicious behavior of the OS on the enclave's execution depends on what counter-measures the enclave has in place to detect and / or protect against an unfaithful OS. Currently, the common ways to facilitate the use of filesystem APIs inside an enclave are:

- Port the entire filesystem inside the enclave [7, 44].
- Keep the filesystem outside the enclave [21, 73]; and for each return parameters, check the data types, bounds on the IO buffers, valid value ranges of API specific values such as error codes, flags, and structures.
- Implement a filesystem shield [12], such that the enclave encrypts all the file data before writing it outside and decrypts the data being read.

All 3 methods help to reduce the attack surface of file syscall return value tampering but do not provably thwart all the attacks in Section 2.2. Appendix A.1 details how their claims lack formal proofs of comprehensiveness. There are several other protected filesystems designed to defend against an untrusted OS in a non-enclave setting, but none of them are formally verified [41, 51].

Verified Guarantees for Enclaves. Formal guarantees have been a subject of investigation in the context of enclaved applications. Various efforts are underway to provide provable confidentiality guarantees for pieces of code executing inside the enclave. Most notably, Moat [78] formally models various adversary models in SGX and ensures that the enclave code does not have any vulnerabilities which leak confidential information. /Confidential [76] builds on Moat to provide a narrow information release channel for enclaves to reduce the attack surface. IMPe builds a type-system to provides a strong non-interference-based information security guarantee for enclave code [34]. All these efforts are towards confidentiality and are orthogonal to BesFS's integrity goals.

Another line of verification research has focussed on certifying the properties of the SGX hardware primitive itself, which BesFS assumes to be correctly implemented. Accordion [55] provides a DSL and uses model checking to ensure that the concurrent interactions between SGX instructions and the shared hardware state maintain linearizability property [40]. Komodo [30] is a formally specified and verified monitor for isolated execution which ensures the confidentiality and integrity of enclaves. TAP [80] does formal modeling and verification to show that SGX and Sanctum [29] provide secure remote execution which includes integrity, confidentiality, and secure measurement. However, the existing works on verified filesystems cannot be simply added on top of TAP [80] because they do not reason about an untrusted OS. BesFS is a layer above the hardware abstractions provided by TAP and Komodo.

Filesystem Verification. Formal verification for large-scale systems such as operating systems [37, 48, 61, 85], hypervisors[8], driver sub-systems [24] and user- applications [39] has been a long-standing area of research. None of these works consider a Byzantine OS, which leads to a completely different modeling of properties.

Filesystem verification for benign OS, however, is in itself a challenging task [46, 47] and is well studied. This includes building abstract specifications [11, 33, 69], systematically finding bugs [86] and POSIX non-compliance [66] in filesystem implementations. Apart from end-to-end verified implementations [9, 68], filesystems are also built to provided crash consistency [14, 31], refinement [75], recovery [25] and safety [23].

8 CONCLUSION

BesFS is a formal and provably Iago-safe API specification for the file-system subset of the POSIX interface. We prove 118 lemmas and two key theorems for safety properties of BesFS implementation. BesFS API is expressive enough to support 17 real applications we test and our principled approach eliminates several bugs.

ACKNOWLEDGMENTS

We thank Michael Steiner from Intel for his feedback. Thanks to Shruti Tople, Shiqi Shen, Teodora Baluta and Zheng Leong Chua for their feedback and assistance in the preparation of this draft. This research was partially supported by a grant from the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] 2018. Ext4 Filesystem Documentation. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt. (2018).
[2] 2018. Intel SGX edger8r Tool. https://github.com/intel/linux-sgx/tree/master/sdk/edger8r/. (2018).
[3] 2018. Intel Software Guard Extensions SDK - Documentation | Intel Software. https://software.intel.com/en-us/sgx-sdk/documentation. (2018).
[4] 2018. intel/linux-sgx-driver at sgx_driver_1.6. https://github.com/intel/linux-sgx-driver/tree/sgx$_{d}$river$_{1}$.6. (2018).
[5] 2018. SPEC CINT2006 Benchmarks. https://www.spec.org/cpu2006/CINT2006/. (2018).
[6] 2018. Standard Library | The Coq Proof Assistant. https://coq.inria.fr/library/Coq.extraction.Extraction.html. (2018).
[7] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious File System for Intel SGX. In *25th Annual Network and Distributed System Security Symposium, NDSS*.
[8] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. 2010. Automated Verification of a Small Hypervisor. In *Verified Software: Theories, Tools, Experiments*, Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–54.
[9] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Atlanta, GA, USA, 175–188. https://doi.org/10.1145/2872362.2872404
[10] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq - POPL 2017. In *CoqPL 2017 The Third International Workshop on Coq for Programming Languages (CoqPL'17)*.
[11] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. 2004. Verifying a File System Implementation. In *Formal Methods and Software Engineering*, Jim Davies, Wolfram Schulte, and Mike Barnett (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 373–390.
[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI '16*.
[13] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*.
[14] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models *(ASPLOS '16)*.
[15] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *CoRR* abs/1709.09917 (2017). arXiv:1709.09917 http://arxiv.org/abs/1709.09917
[16] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser
[17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck
[18] Ran Canetti, Suresh Chari, Shai Halevi, Birgit Pfitzmann, Arnab Roy, Michael Steiner, and Wietse Venema. 2011. Composable Security Analysis of OS Services. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security (ACNS'11)*. Springer-Verlag, Berlin, Heidelberg, 431–448. http://dl.acm.org/citation.cfm?id=2025968.2026002
[19] D. Champagne and R. B. Lee. 2010. Scalable architectural support for trusted software. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416657
[20] Suresh Chari, Shai Halevi, and Wietse Z. Venema. 2010. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *NDSS*. The Internet Society.
[21] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai
[22] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/2451116.2451145
[23] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*. Shanghai, China.
[24] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 431–447. https://doi.org/10.1145/2908080.2908101
[25] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402
[26] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. ACM, New York, NY, USA, 7–18. https://doi.org/10.1145/3052973.3053007
[27] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1346281.1346284
[28] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086. (2016). http://eprint.iacr.org/2016/086.
[29] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security '16*.
[30] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software, In 26th ACM Symposium on Operating Systems Principles (SOSP'17). https://www.microsoft.com/en-us/research/publication/komodo-using-verification-disentangle-secure-enclave-hardware-software/
[31] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying File System Consistency at Runtime. *Trans. Storage* 8, 4, Article 15 (Dec. 2012), 29 pages. https://doi.org/10.1145/2385603.2385608

[32] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. 2017. Sgx-Lapd: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *Research in Attacks, Intrusions, and Defenses*, Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer International Publishing, Cham, 357–380.

[33] Philippa Gardner, Gian Ntzik, and Adam Wright. 2014. Local Reasoning for the POSIX File System. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–188.

[34] Anitha Gollamudi and Stephen Chong. 2016. Automatic Enforcement of Expressive Security Policies Using Enclaves. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 494–513. https://doi.org/10.1145/2983990.2984002

[35] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3065913.3065915

[36] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 217–233. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[37] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 653–669. http://dl.acm.org/citation.cfm?id=3026877.3026928

[38] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 299–312. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahnel

[39] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-end Security via Automated Full-system Verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 165–181. http://dl.acm.org/citation.cfm?id=2685048.2685062

[40] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[41] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/2451116.2451146

[42] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the 24th USENIX Security Symposium*.

[43] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 969–986. https://doi.org/10.1109/SP.2016.62

[44] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 533–549. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt

[45] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1251375.1251384

[46] Rajeev Joshi and Gerard J. Holzmann. 2008. *A Mini Challenge: Build a Verifiable Filesystem*. Springer Berlin Heidelberg, Berlin, Heidelberg, 49–56. https://doi.org/10.1007/978-3-540-69149-5$_6$

[47] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. 2013. File Systems Deserve Verification Too!. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems (PLOS '13)*. ACM, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/2525528.2525530

[48] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[49] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[50] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 205–221. https://doi.org/10.1145/3064176.3064192

[51] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen Hofmann, Yuanzhong Xu, and Emmett Witchel. 2016. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *ASPLOS*.

[52] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 523–539. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-jaehyuk

[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 557–574. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[54] Xavier Leroy. 2005 - 2018. The CompCert verified compiler. http://compcert.inria.fr/. (2005 - 2018).

[55] Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. 2015. Verifying Linearizability of IntelÂő Software Guard Extensions. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 144–160.

[56] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[57] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R.B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE S&P*.

[58] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS '13)*. ACM, New York, NY, USA, 311–324. https://doi.org/10.1145/2508859.2516692

[59] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. ACM, New York, NY, USA, Article 10, 1 pages. https://doi.org/10.1145/2487726.2488368

[60] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. *CoRR* abs/1703.06986 (2017). arXiv:1703.06986 http://arxiv.org/abs/1703.06986

[61] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 252–269. https://doi.org/10.1145/3132747.3132748

[62] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 238–253. https://doi.org/10.1145/3064176.3064219

[63] Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*. ACM, New York, NY, USA, 71–84. https://doi.org/10.1145/158511.158524

[64] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2017. *Software Foundations*.

[65] Dan R. K. Ports and Tal Garfinkel. 2008. Towards Application Security on Untrusted Operating Systems. In *HOTSEC*.

[66] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems *(SOSP '15)*.

[67] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2017. ZeroTrace : Oblivious Memory Primitives from Intel SGX. Cryptology ePrint Archive, Report 2017/549. (2017). https://eprint.iacr.org/2017/549.

[68] Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, Dominik Haneberg, and Wolfgang Reif. 2014. Development of a Verified Flash File System. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Yamine Ait Ameur and Klaus-Dieter Schewe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 9–24.

[69] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. 2009. Abstract Specification of the UBIFS File System for Flash Memory. In

Proceedings of the 2Nd World Congress on Formal Methods (FM '09). Springer-Verlag, Berlin, Heidelberg, 190–206. https://doi.org/10.1007/978-3-642-05089-3_13

[70] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. *CoRR* abs/1702.08719 (2017). arXiv:1702.08719 http://arxiv.org/abs/1702.08719

[71] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs *(NDSS)*. Internet Society.

[72] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16)*. ACM, New York, NY, USA, 317–328. https://doi.org/10.1145/2897845.2897885

[73] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves. In *24th Annual Network and Distributed System Security Symposium, NDSS*.

[74] Shweta Shinde, Shruti Tople, Deepak Kathayat, and Prateek Saxena. *PodArch: Protecting Legacy Applications with a Purely Hardware TCB*. Technical Report.

[75] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button Verification of File Systems via Crash Refinement *(OSDI'16)*.

[76] Rohit Sinha, Manuel Costa, Akash Lal, Nuno Lopes, Sanjit Seshia, Sriram Rajamani, and Kapil Vaswani. A Design and Verification Methodology for Secure Isolated Regions. In *PLDI '16*.

[77] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A Design and Verification Methodology for Secure Isolated Regions *(PLDI '16)*.

[78] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying Confidentiality of Enclave Programs *(CCS '15)*.

[79] R. Strackx and F. Piessens. 2017. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *ArXiv e-prints* (Dec. 2017). arXiv:cs.CR/1712.08519

[80] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. 2017. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 2435–2450. https://doi.org/10.1145/3133956.3134098

[81] Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

[82] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. 2017. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. *CoRR* abs/1705.07289 (2017). arXiv:1705.07289 http://arxiv.org/abs/1705.07289

[83] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 81–93. https://doi.org/10.1145/3079856.3080208

[84] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P '15'*.

[85] Jean Yang and Chris Hawblitzel. 2010. Safe to the Last Instruction: Automated Verification of a Type-safe Operating System. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 99–110. https://doi.org/10.1145/1806596.1806610

[86] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423. https://doi.org/10.1145/1189256.1189259

A APPENDIX

A.1 Defenses Against Iago Attacks in Existing Systems.

Following are the verbatim quotes from the research papers of existing systems, which do not make any concrete claims.

Haven. We use established techniques to correctly implement the OS primitives in the presence of a malicious host: careful defensive coding, exhaustive validation of untrusted inputs, and encryption and integrity protection of any private data exposed to untrusted code.

Scone. The enclave code handling system calls also ensures that pointers passed by the OS to the enclave do not point to enclave memory. This check protects the enclave from memory-based Iago attacks [12] and is performed for all shield libraries.

Panoply. The shim library performs checks for Iago attacks, safeguarding against low-level data-tampering for OS services.

Graphene-SGX. Any SGX framework must provide some shielding support, to validate or reject inputs from the untrusted OS. The complexity of shielding is directly related to the interface complexity: inasmuch as a library OS or shim can reduce the size or complexity of the enclave API, the risks of a successful Iago attack are reduced.

Ryoan. Ryoan allows files to be preloaded in memory, and the list of preloaded files must be determined before the module is confined; e.g., they can be listed in the DAG specification, or requested by the module during initialization. Ryoan presents POSIX-compatible APIs to access preloaded files that are available even after the module is confined. Second, a confined module can create temporary files and directories (which Ryoan keeps in enclave memory). When the module is destroyed or reset, all temporary files and directories are destroyed, and all changes to preloaded files are reverted.

A.2 Layers in Filesystem Stack

The higher the layer we safeguard, the *larger* the attack surface we can eliminate, and the more implementation-agnostic the BesFS API becomes. Figure 2 shows various layers where one can intercept the filesystem operations for integrity checks with the application being the topmost layer and the device driver is the lowest layer.

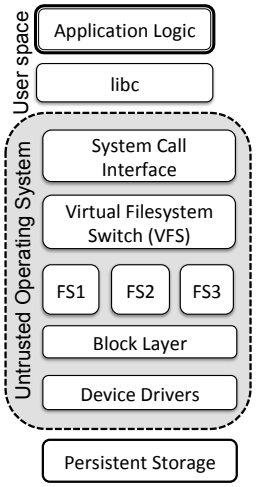

Figure 2: Layers of the filesystem where the highest layer is enclave application and lowest layer is the persistent device storage. The dotted area shows the components within an untrusted OS.

A.3 Implementation Details

Table 6 shows the detailed break down of LOC added for adding support for each BesFS API in Panoply.

Test	Time (usec)		Overhead
	Panoply	BesFS + Panoply	
multicreate	517264	939727	0.8x
multiwrite	424193	1025797	1.4x
multiread	1007232	4756286	3.7x
multicreatewrite	245901	1578016	5.4x
multiopen	668430	2868140	3.3x
multicreatemany	21607	102655	3.8x

Table 7: Single syscall Performance. Execution time for FSCQ **single syscall benchmarks in** Panoply **and** BesFS**.**

BesFS API Name	BesFS Trusted Custom	Panoply Trusted Custom	Panoply Trusted Auto	Panoply Untrusted Custom	Panoply Untrusted Auto	Total
close	13	4	16	1	3	37
create	33	4	28	1	3	69
open	63	13	13	1	3	93
mkdir	28	4	28	1	3	64
remove	5	4	27	1	3	40
rmdir	5	4	27	1	3	40
stat	1	4	40	1	3	49
readdir	12	4	16	2	3	37
chmod	26	4	28	1	3	62
lseek	11	4	18	1	3	37
read	8	39	30	4	3	84
write	12	39	29	2	3	85
ftruncate	2	4	17	1	3	27
TOTAL	**219**	**131**	**317**	**18**	**39**	**724**

Table 6: LOC for implementing BesFS **APIs in** Panoply**. Column** 1 **represents the code wrapper code to integrate** BesFS **implementation in** Panoply**. Column** 3 − 6 **represent the addition to** Panoply **for integration or fixing bugs. Custom implies hand-written code and Auto implies that the code was generated by Intel SGX SDK's `edger8r` tool. The Trusted code runs inside the enclave whereas Untrusted code runs outside the enclave.**

A.4 Feasibility of Machine-checked Executable Code

Note that our primary goal in this paper is not to generate certified assembly code but to certify higher-level properties of BesFS implementation. Currently, BesFS only guarantees certified correctness for its Coq implementation. However, multiple projects have shown that it is possible to extend certification all the way to assembly. Thus, there are no fundamental limitations for certifying BesFS's machine code in the future. For our specific setup, one option is to use a CertiCoq (Coq to C) and CompCert (C to assembly). CertiCoq [10] is a certified compiler from Gallina to CompCert C light. CompCert [54] is a certified, optimized C compiler which ensures that the generated machine code for various processors is efficient and behaves exactly as prescribed by the semantics of the source program. Thus, both these certified compilers can be composed to give a certified Coq-to-assembly compiler which we can use to certify our machine code for BesFS. We have contacted authors of CertiCoq who report that the tool is under active development, not available publicly, and cannot be used for our C implementation yet. We believe that once CertiCoq is fully functional, BesFS can ensure that its conversion from machine proved Coq implementation to assembly code executing inside the enclave is certified end-to-end; however, this is beyond the realm of demonstration today.

A.5 Performance

We perform the following measurements for our benchmarks:

(1) Enclaved execution in Panoply without BesFS checks.
(2) Enclaved execution in Panoply with BesFS checks.

All our results are aggregated over 5 runs. For non-I/O intensive benchmarks (SPEC CINT2006) we observe an overhead of 16.22% for the 7. For FSCQ benchmarks, our average overhead is 3.1× for single syscall tests and 6.7× for large I / O workloads. Thus, BesFS incurs an average of 3.3× CPU overhead compared to the baseline. Our break down shows that a large fraction of BesFS's overhead is because of page-level AES-GCM encryption-decryption for preserving integrity and system call latency in Panoply's synchronous `OCALL` mechanism. We present a set of optimizations for real world applications so as not to incur excessive overhead.

Single-syscalls. We use the FSCQ micro-benchmark to measure the performance of the I/O intensive calls in BesFS. Table 7 shows the overhead of BesFS when a single system call is called multiple times. The average overhead over 3.1×. We observe that read-write operations incur a large overhead. Specifically, our read operation is slowed down by 3.7×, while create+write is 5.4× slower. The primary reason for this is that BesFS performs page-level AES-GCM authenticated encryption when the file content is stored on the disk. Thus, each read and write operation leads to encryption-decryption and integrity computation of at least one page.

Large I/O Workloads. We test the performance of BesFS under various file access patterns. We run all the tests in FSCQ with the configuration of the block size of 8 KB, I/O transfer size is 1KB and total file size to be 1 MB. We perform 100000 number of each type of operations on files. We observe an average overhead of 6.7× because of BesFS checks. FSCQ performs a series of sequential write, sequential read, re-read, random read, random write, multi-write and multi-read operations. Figure 3 the bandwidth for each of these operations. Sequential access incurs relatively less performance overhead because they consolidate the page-level encryption-decryption for every 4K bytes. Random accesses on the other hand are more expensive because each read / write may cause a page-level encryption-decryption. Since BesFS does not cache any page content, re-reads incur the same overhead as sequential read.

SPEC CINT2006 Benchmarks. We test 7 benchmarks namely `astar`, `bzip2`, `h264ref`, `hmmer`, `libquantum`, `mcf`, `sjeng` from SPEC. Each of these benchmarks takes in a configuration file and optionally input file to produce an output file. Figure 4 shows the performance for each of these benchmarks. With our `libc` analysis in Table 4, we also measure the frequency of each call per application. Programs `hmmer`, `href`, `sjeng`, and `libquantum` have relatively less overhead. On the other hand, `astar`, `bzip2`, and `mcf` exhibit larger overhead. On further inspection, we notice that `astar` and `mcf` use `fscanf` to read the configuration files. Thus, reading each character leads to a page read and corresponding decryption and integrity

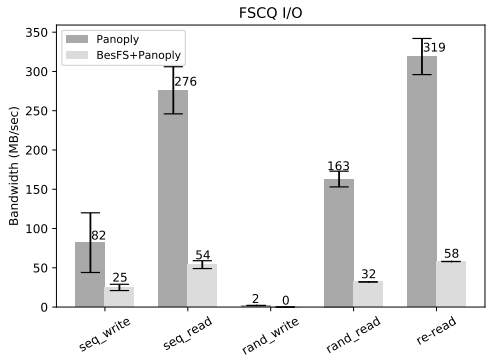

Figure 3: Performance for FSCQ **Large I/O Workloads.**

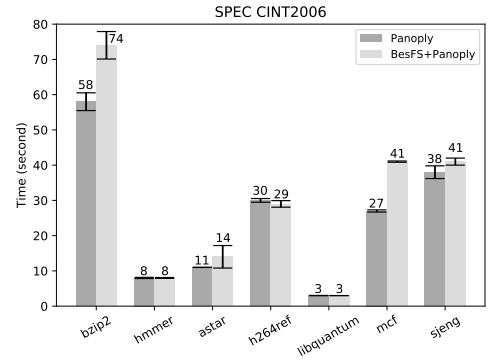

Figure 4: Performance for SPEC CINT2006 Benchmarks.

check. Further, `astar` reads a binary size of 65 KB for processing. As shown by our single syscall measurements (Table 7), reads are expensive. Both these factors amplify the slowdown for `astar`. `bzip2`, and `mcf` output the benchmark results to a new file of sizes 274 and 32 KB respectively which leads to a slowdown. Specifically, `bzip2` reads input file in chunks of 5000 bytes which leads to a 2-page read / write and decrypt/encrypt per chunk. Finally, `libquantum` has the lowest overhead because it does not perform any file operations.

A.6 Optimizations

Note that we do not include any optimizations for caching or memory management at the moment. There is scope for improving BesFS's performance with various optimizations which are independent of the BesFS safety properties.

(O1) Reduce `OCALLs`. When the applications invoke a protected API, BesFS immediately relays the call to the OS. This results in a lot of `OCALLs`. For example, we implement `fgets` using `fgetc` because we don't know beforehand how much buffer size we need to read until we encounter a newline. This is safe but super slow — each character read causes an I/O of 4KB. We can see the effect of this in the `mcf` benchmark which reads character by character. An alternative is to maintain a buffer inside the enclave which reflects the changes, instead of doing an immediate BesFS call (and hence an `OCALL`) for each operation. Then the accumulated changes of batched calls can be flushed to the OS periodically. We can do similar optimizations for writes.

(O2) Batch Processing. Since we integrate BesFS with Panoply, we have to interface at the `libc` interface for tunneling the calls to the untrusted OS. However, other systems such as Scone, Haven, Graphene-SGX keep the C library inside the enclave and interface with the OS purely at the syscall level. All modern C libraries (e.g., `musl-libc`, `glibc`, `eglibc`) have optimized the number of syscalls. They do not invoke an underlying syscall for each `libc` API. Instead, they batch as many I/O calls as possible to avoid expensive context switches.

(O3) Optimized Page Allocation Algorithm. BesFS ensures that each page in the memory is being used only by a single file. Thus when BesFS wants a new page, its lemma states that page allocation algorithm should return an unused page. Similarly, when the page is unallocated, BesFS states that no file should use that page after allocation. To satisfy this lemma, BesFS implementation in Panoply keeps a page bitmap, which is used for allocation and deallocation.

(O4) Optimized Block Alignment. Our current implementation assumes a page of 4096 bytes. For each such page, BesFS uses the first 4000 bytes to store the file content and the rest of the 96 bytes for BesFS metadata such as the integrity tags. Thus any single page operation in an I/O intensive program, BesFS will incur a read / write (and hence decrypt/encrypt) of two pages. Applications which do custom block alignment to tune their performance will see an added slowdown. Our choice of BesFS page sizes was just a design choice and is independent of any BesFS proofs. BesFS proofs and implementation use a macro for these values, and if required, the developer can change them to suit their requirements. The developer can change the block size in the application to 4000 bytes.

(O5) Reduce `OCALL` Costs. Our current implementation is integrated with Panoply, which does synchronous `OCALLs`. It has been experimentally shown that asynchronous `OCALLs` are much faster and can speed up the applications by an order of magnitude [12, 62, 83]. As long as the asynchronous call implementations obey the syscall semantics enforced by BesFS, our implementation will work out of the box with this optimization.