BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof

Shweta Shinde, *University of California, Berkeley;* Shengyi Wang and Pinghai Yuan, *National University of Singapore;* Aquinas Hobor, *National University of Singapore & Yale-NUS College;* Abhik Roychoudhury and Prateek Saxena, *National University of Singapore*

https://www.usenix.org/conference/usenixsecurity20/presentation/shinde

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof

Shweta Shinde *†
University of California, Berkeley

Shengyi Wang*
National University of Singapore

Pinghai Yuan
National University of Singapore

Aquinas Hobor
National University of Singapore & Yale-NUS College

Abhik Roychoudhury
National University of Singapore

Prateek Saxena
National University of Singapore

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 still compromise the integrity of an enclave by tampering with the system call return values. In fact, it has been shown that a subclass of these attacks, called Iago attacks, enables arbitrary logic execution in enclave programs. Existing enclave systems have very large TCB and they implement ad-hoc checks at the system call interface which are hard to verify for completeness. To this end, we present BESFS—the first filesystem interface which provably protects the enclave integrity against a completely malicious OS. We prove 167 lemmas and 2 key theorems in 4625 lines of Coq proof scripts, which directly proves the safety properties of the BESFS specification. BESFS comprises of 15 APIs with compositional safety and is expressive enough to support 31 real applications we test. BESFS integrates into existing SGX-enabled applications with minimal impact to TCB. BESFS can serve as a reference implementation for hand-coded API checks.

1 Introduction

Existing computer systems encompass millions of lines of complex operating system (OS) code, which is highly susceptible to vulnerabilities, but is 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 [41], Sanctum [24], Keystone [38]) 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 CPU-protected memory regions called enclaves; with various adversary models supported in individual designs [24, 25, 38, 41, 47].

Enclave systems promise 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 [25, 50], as well as verified enclave confidentiality properties [48, 49]. One critical gap remains unaddressed: verifying the integrity of the application from a *hostile* OS. Applications are increasingly becoming easier to port to enclaves [15, 16, 18, 46]; however, these legacy applications optimistically assume that the OS is benign. A hostile OS, however, can behave arbitrarily by violating assumptions inherent in the basic abstractions of processes or files and exchange malicious data with the application. This well-known attack was originally identified by Ports and Garfinkel as *system call tampering* [43], more recently discussed as a subclass called Iago attacks [19].

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 [16], Google Asylo [3], Microsoft Open Enclave [6], Intel SGX SDK [4], Panoply [46], Graphene-SGX [18], and Scone [15] 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. Reasoning about a *complete* set of checks that suffice to capture all attacks is difficult.

In this work, we take a step towards a formally verified TCB to protect the 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

*These joint first authors contributed equally to this work.
†Part of the research was done while at National University of Singapore.

in a high-level language. Client applications running in SGX enclaves interact with a commodity (e.g., Linux) OS via our BESFS implementation, running as a library (see Figure 4). Applications use the POSIX filesystem API transparently (see Table 3), requiring minimal integration changes. Being formally verified, BESFS specifications and implementation can further be used to test or verify other implementations 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 various 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 wide variety of real-world applications and benchmarks with a BESFS-enhanced system for running SGX applications. These tests show a modest impact on compatibility, showing that the BESFS specification is rich enough to run many practical applications on commodity OS implementations. The BESFS API has only 15 core operations. However, it is accompanied crucially by a 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. Various design challenges arise (Section 5) 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 Coq's automatic tactics.

BESFS is specified, implemented and formally verified in Coq which is a higher-level language. Converting Coq code to machine code is out of the scope of this paper. Most existing systems do not provide these guarantees even for non-enclave code. There are several intermediate challenges in such a conversion, especially when it is enclave-bound. Thus we resort to a hand-coded conversion of BESFS implementation from Coq-to-`C` and then use an Intel SGX compatible compiler to obtain machine code which can execute inside the enclave. For the completeness of the paper, we outline various challenges we faced in our attempt to generate enclave-bound machine code from our Coq implementation of BESFS. We discuss the existing alternatives and the required additions to the immediate state of systems to make this feasible.

Results. Our BESFSCoq proof comprises of 167 theorems and 4625 LOC. Our hand-coded `C` implementation of BESFS is 1449 LOC and we add 724 LOC of stubs for compatibility with enclave code. We use this `C` implementation for our performance evaluation. We demonstrate the expressiveness of BESFS by supporting 31 programs from benchmarks and real-world applications. We show that BESFS is compatible with state-of-the-art filesystems, benchmarks, and applications we tested. It aids in finding implementation mistakes in existing filesystem APIs exposed by Intel SGX frameworks. We hope BESFS will serve as a specification for future optimizations and other hand-coded implementations.

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 in Coq which are machine-checked for their soundness w.r.t. the API specifications. Our auto-generated run-time monitoring mechanism ensures that the concrete filesystem execution stays within the envelope of our specification.
- We prove 167 lemmas and 2 key theorems in 4625 LOC Coq. We evaluate correctness, compatibility, and expressiveness of BESFS. We showcase BESFS on 31 programs from real-world applications and standard benchmarks for CPU, I/O, and filesystem workloads.

2 Problem

There has been long-standing research on protecting the OS from user-level applications. In this work, the threat model is reversed; the applications demand protection against a malicious OS kernel. We briefly review Intel SGX specifics and highlight the need for a formal approach.

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*. The CPU allocates protected physical memory from *Enclave Page Cache (EPC)* that backs the enclave main memory and its content is encrypted. Only the owner enclave can access its own content at any point during execution. The hardware does not allow any other process or the OS to modify code and data or read plain text inside the enclave boundary. Interested readers can refer to [23] 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* (`OCALL`s) to executes system calls outside the enclave. The enclave code copies the `OCALL`

System Name	Release Date	Total LOC	# of APIs	FS API	API Level
Graphene-SGX	July 2016	1325978	28	5	syscall
Panoply	Dec 2016	20213	254	37	libc
Intel SDK	Dec 2016	119234	24	15	Custom
Google Asylo	May 2018	400465	39	7	Custom
BesFS	Aug 2018	1449	21	13	POSIX

Table 1: Comparison of existing SGX filesystem support.

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 from the untrusted memory to the protected enclave memory. This mechanism facilitates interactions between the enclave and non-enclave logic of an application. A large fraction of enclave applications need to dispatch `OCALL`s for standard (e.g., syscalls, libc) or application-specific APIs.

Syscall Parameter Tampering. This is a broad class of attacks and has been inspected in various aspects by Ports and Garfinkel [43]; a specific subclass of it is called as Iago attacks [19]. 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 by maintaining metadata such as a secure hash for file pages and protecting them 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 pathname to the associated protection metadata. This way, checking 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. In this paper, our focus is on the filesystem subset of the system calls. Further, we concentrate on enclave-like systems for Intel SGX, but our work applies equally well to other systems [22, 24, 25, 32, 38].

2.2 The Baseline: Existing Systems

All SGX-based systems such as Haven [16], Scone [15], Panoply [46], Graphene-SGX [18], Intel Protected File System [4] which either use SDK or hand-coded `OCALL` wrappers must address syscall parameter tampering attacks. Even non-SGX TEEs have been shown to face the same threat [10, 22, 38]. These systems are upfront in acknowledging this gap and employ ad-hoc checks for each API to address a subset of attacks. Using integrity preserving filesystems [13] and formally testing if a filesystem abides by POSIX semantics [44] are stepping stones towards our goal, but they do not reason about intentional deviations by a *completely Byzantine* OS. We demonstrate representative attack capabilities on state-of-the-art enclave systems with encrypted file storage to motivate why a provable approach down to the details is important.

Baseline. We assume that the filesystem API uses authenticated encryption and attestation to prevent the OS from directly tampering the file content. Further, we assume a setting where the enclave tunnels all file-related system / library calls to the untrusted OS. The untrusted OS simply reads and writes encrypted blocks of data to the disk such that the content can only be decrypted inside the enclave. Most publicly available enclave frameworks support such a baseline defense. For concreteness, we discuss specific details of the four open systems available today which support a filesystem interface for enclave applications: Graphene-SGX [18], Panoply [46], Intel Protected File System Library [4], and Google Asylo [3]. Table 1 shows the number of file APIs supported and the LOC of these systems indicating that custom implementations have large TCB irrespective of the APIs they support. More importantly, as we show in Section 2.3, they employ ad-hoc checks which do not completely defeat the attacks by the OS. As opposed to the state-of-the-art, BesFS provides provable guarantees. Our corresponding implementation in Coq as well as hand-coded `C`) lowers the TCB.

```
int log (char* fname, int mode, char* buf, int len) {
  int errnum, cnt = 0; FILE* fd = fopen(fname, mode);
  if (fd == NULL) {
    errnum = errno;
    if (errnum == EINVAL) fd = fopen (fname, "a"); // append
    if (errnum == ENOENT)
      if (fname == NULL) fname = "default.log";
      fd = create_log(fname); // create empty log file
    if (errnum == EINTR) fd = fopen (fname, mode); // retry
  }
  if (fd) cnt = fwrite(buf, 1, len, fd); // write log
  return cnt;
}
void cast_vote () { // each tor node ...
  status = log(log_file, mode, &vote, vote_len);
  if (status) start_election();
```

Figure 1: Example enclave logic. The enclave opens a log file and attempts recovery on failure by either changing the mode (`EINVAL`), opening a new file since the path does not exist (`ENOENT`), or reattempting the call (`EINTR`).

2.3 Is Encryption Sufficient?

Our baseline system encrypts and adds MAC tags to file content. We show that this is not enough to protect against a malicious OS. We recall attack examples from prior work and present new attacks to show that BesFS is needed to defeat a broad class of attacks that go well beyond memory safety.

Memory-safety Iago Attacks. Iago attacks show a subclass of concrete attacks on memory allocation interfaces, wherein the malicious OS overlaps memory-mapped (via `mmap`) pages. The attack results in subverting the control flow in the enclaved application. Iago attacks demonstrate that verifying return values may require user-level defenses to carefully enforce invariants on the virtual memory layout.

Logic Bugs via Return Value Tampering. We show how the OS can mislead the application-level into taking incorrect actions, without causing a crash, by exploiting the semantic

```
1 static int file_open (..., const char * uri, int access, int share,
      int create, int options) {
2    int fd = ocall_open(uri, access|create|options, share);
3    if (fd < 0)
4       return fd;
5    ...
6 }
7 static int sgx_ocall_open(void * pms) {
8    ms_ocall_open_t * ms = (ms_ocall_open_t *) pms;
9    int ret;
10   ODEBUG(OCALL_OPEN, ms);
11   ret = INLINE_SYSCALL(open, ...);
12   return IS_ERR(ret)?unix_to_pal_error(ERRNO(ret)):ret;
13 }
```

(a) Graphene-SGX. Checks on success; otherwise forwards the error.

```
1 SGX_FILE* sgx_fopen
2 (const char* filename, const char* mode) {
3  return sgx_fopen_internal(filename, mode, NULL, key);
4 }
5 static SGX_FILE* sgx_fopen_internal
6 (const char* filename, const char* mode) {
7  protected_fs_file* file = NULL;
8  if (filename == NULL || mode == NULL) {
9    errno = EINVAL;
10   return NULL;
11 }
12   ...
13 }
```

(c) Intel Protected File System. Returns EINVAL instead of ENOENT.

```
1 SGX_WRAPPER_FILE sgx_wrapper_fopen(const char* filename, const char
      * mode) {
2  SGX_WRAPPER_FILE f = 0;
3  sgx_status_t status = ocall_fopen(&f, filename, mode);
4   CHECK_STATUS(status);
5  return f;
6 }
```

(b) Panoply. Forwards the fd and errors as-is if `OCALL` *fails.*

```
1 int secure_open(const char *pathname, int flags, ..){
2  ...
3  bool is_new_file = (enc_untrusted_access(pathname, F_OK) == -1);
4  int fd = enc_untrusted_open(pathname, flags, mode);
5  if (fd == -1)
6    return -1;
7  ...
8 }
9 int enc_untrusted_open(const char *path_name, int flags) {
10  uint32_t mode = 0;
11  int result;
12  sgx_status_t status = ocall_enc_untrusted_open(&result, path_name
        , flags, mode);
13  if (status != SGX_SUCCESS) {
14    errno = EINTR;
15    return -1;
16  }
17  return result;
18 }
```

(d) Google Asylo. Suppresses the error on failure and returns EINT.

Figure 2: SGX filesystem API support. Code snippets from four systems which support file open operation inside the enclave.

gap between SGX guarantees and POSIX API. This attack works on encrypted filesystems since it perpetrates by returning inconsistent return values. Figure 1 shows a simplified enclave code which is executed by a node in a Tor-like service [9]. The enclave logic casts votes, appending it to a log file at each epoch, say in a sub-step of a consensus process. Specifically, the enclave function `log_vote` opens an existing log file in append mode. The enclave checks if the open was successful or were there any errors. The function handles the error conditions and once the `fopen` is successful, it writes the vote content to the file via `fwrite`. As per POSIX standard, the library should return a `NULL` file pointer on `fopen` failure and set the `errno` is set to indicate the error. If the file name is invalid (e.g., empty string or a non-existing file path) the error is `ENOENT`. If the mode is invalid the error should be `EINVAL`, while `EINTR` indicates that the call was interrupted and may succeed on a re-attempt. Figure 1 performs error handling assuming a POSIX-reliant filesystem.

Figure 2 shows the implementation code snippets of file open operation in four existing SGX platforms which implement four different types of checks. Both Graphene-SGX and Panoply simply forward the `errno` to the caller without performing any checks (Figure 2a, 2b). In our example (Figure 1), the OS can trick the enclave into creating an empty file by falsely sending `ENOENT` error code, even though the log file exists. Both the systems cannot detect this attack. Intel's Protected File System (Figure 2c) returns an incorrect error code as per the POSIX standards. If the enclave passes the log name to be an empty string, the application will incorrectly receive `EINVAL` and will not be able to log the vote. Google Asylo (Figure 2d), does not perform any pre-checks on the parameters and if the `OCALL` returns any errors, the system always overwrites it with `EINTR` (Line 14). Thus, our example demonstrates that although the existing systems employ encryption on file content, they are vulnerable to logic bugs due to incomplete interface security checks.

Glibc Logic Vulnerability due to Bad Initialization. We present another attack which cannot be defeated by using an encrypted file system or sealing within the enclave. The glibc malloc subsystem allocates large chunks of memory via anonymous `mmap` (Figure 3 Line 13). It then distributes and collects parts of these chunks via `malloc` and `free` calls. For glibc's internal buffer management, the first 8 bytes of each mmaped region are reserved for meta-data (e.g., tracking the sizes of the allocated chunks in Figure 3 Line 5). The POSIX specification dictates that if the `mmap` syscall requests for anonymous memory regions, which are not file-backed, the OS must initialize the memory contents to 0. Thus, when glibc acquires a large buffer via anonymous-mapped memory region it assumes that this region is filled with 0s by the kernel. The glibc implementation then updates the size of the current block by writing to the size field. For the first block being mmaped, glibc does *not* write 0 to the `prev_size` as it assumes those bytes are already set to 0.

In the implementation of `free`, glibc unmaps chunks if all slots in those chunks are unallocated. For this, it performs some arithmetic computation over the start address of a chunk as well as the sizes of the current and previous chunks. Sup-

```
1 /* There is only one instance of the malloc params. */
2 static struct malloc_par mp_ = {...};
3 typedef struct malloc_chunk {
4     size_t prev_size; /* previous chunk size(if free) */
5     size_t size;      /* Size (bytes) including metadata */
6     ...
7 }mchunkptr;
8 static void *sysmalloc (INTERNAL_SIZE_T nb, mstate av) {
9     ...
10    mchunkptr p;      /* the allocated/returned chunk */
11    char *mm;         /* return value from mmap call */
12    ...
13    mm = (char *) (mmap (0, size, PROT_READ | PROT_WRITE, 0));
14    ...
15    p = (mchunkptr) mm;
16    p->size = size | IS_MMAPPED;
17    ....
18    return chunk2mem (p);
19    ...
20 }
21 static void munmap_chunk (mchunkptr p) {
22    ...
23    uintptr_t block = (uintptr_t) p - p->prev_size;
24    size_t total_size = p->prev_size + size;
25    ...
26    munmap ((char *) block, total_size);
27    ...
28 }
```

Figure 3: Glibc Attack. OS corrupts `prev_size` via `mmap` (Line 13). It can trick glibc into inadvertently unmapping larger memory range (Line 26) without the updating glibc's internal metadata which violates its constraints.

pose the allocated region is $[P, P+s)$ where P and s denote the start address and length respectively. Further, let X denote the value of the first 8 bytes of a chunk i.e., variable `prev_size`. Lines 23-26 in Figure 3 invoke the `unmap` syscall for the address range $[P-X, P+s)$. In the case of the first chunk, the value of X is 0 and glibc will unmap $[P, P+s]$ which is correct. Note that if the OS returns mmaped memory which is filled with non-0s, it can control the value of X. For example, if the OS selects $X \neq 0 \wedge X < P$, it will trick glibc into unmapping not only $[P, P+s]$ but also $[P-X, P]$. Neither glibc nor the application is aware of this inadvertent unmapping and their internal metadata will no longer reflect the correct state.

In general, such an attack can break the consistency enforced by various program components. For instance, a garbage collector which maintains invariants about how objects are traced by reference chains may use the memory mapping information to mark the memory occupied by freed objects to avoid use-after-free. More broadly, many security primitives (e.g., control-flow integrity, fat pointers, taint analyses) maintain shadow metadata at fixed offsets from program objects, which could be affected by such inconsistency bugs.

3 BESFS Design

All the classes of filesystem API attacks presented in Section 2.3 stem from the fact that the OS can deviate from its expected semantics. Our goal is to design a filesystem interface, called BESFS, which protects the enclave from a broad category of such attacks. These attacks include (but are not limited to) Iago attacks, file content manipulation such as mapping multiple file blocks of the same or different files to single physical block, operating on content at the wrong offset or block, and misaligned sequences of file blocks in a file. Further, the OS can perpetrate mismatch attacks by ignoring the user-provided parameters such as paths, file descriptor, or size (e.g., violate the size requested in the operations). Lastly, it can change the error codes returned by the filesystem and force the enclave to execute a different control-flow path.

3.1 Approach

We seek for the right abstraction which is necessary to capture the filesystem behavior inside the enclave as well as sufficient to detect any deviation from the Byzantine OS. Attacks on an enclave can arise at multiple layers of the filesystem stack. Our choice of the layer where we formally proof-check the BESFS API is guided by the observation that the higher the layer we safeguard, the *larger* the attack surface (i.e., TCB) we can eliminate, and the more implementation-agnostic the BESFS API becomes. One could include all the layers starting at the disk kernel driver, where content is finally mapped to persistent storage, in the enclave TCB. Enforcing safety at this interface will require simply encrypting/decrypting disk blocks with correct handling for block positions [37]. 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 [16, 18]. 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 TCB. 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). For instance, this is 1.29 MLOC and 88 KLOC in glibc and musl-libc, respectively. 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 these layers, excluding them from the TCB.

BESFS models the POSIX standard for file sub-systems. POSIX is a documented standard, with small variations across implementations on various OSes [44]. 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 API choice reduces the proof complexity as they are dispatched for simpler data structures.

Solution Overview. BESFS is an abstract filesystem interface which ensures that the OS follows the semantics of a benign filesystem—it is exhibiting observationally equivalent behavior 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 not only includes POSIX-compliance but also a set of safety properties expected from the underlying filesystem implementation. We design a set of 15 core filesystem APIs along with a safety specification. Table 3 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* (SP_1-SP_5 and TP_1-TP_{15}) 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 7 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 includes all the above attacks. Thus, BESFS is a state transition system. We define a good start state that satisfies the state properties (SP_1-SP_5). Our transition properties (TP_1-TP_{15}) ensure that the file system is in a good state after executing a BESFS API call.

Importantly, we prove that the safety of BESFS API is *serially composable*. This composability is crucial to allow executions of benign applications that 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. Thus, composition property allows us to reduce the size of the core APIs that have to be proved as well as reduce the attack surface for the OS. To ensure serial composition, the state safety properties (SP_1-SP_5) enforce 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 can be chained, 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 (TP_1-TP_{15}). We provide a machine-checked Coq implementation of the BESFS API (Section 3.2).

Theorem 1 (State Transition Safety.) *Given a good state S satisfying pre_i, if we execute f_i to reach state S', then S' is always a good state and the relation between S and S' is valid according to the transition relation τ_i:*

$$\forall S, S', i. \quad S \models SP_1\text{–}SP_5 \wedge pre_i(S) \wedge S \overset{f_i}{\rightsquigarrow} S' \quad \Rightarrow \\ \tau_i(S, S') \wedge S' \models SP_1\text{–}SP_5$$

We can verify sequences of calls to our API 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 1 to show that a method invocation in BESFS always produces a good state for a sequential composition of transitions. Coq proof assistant dispatches the proof.

Theorem 2 (Sequential Composition 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:*

$$S_0 \models SP_1\text{–}SP_5 \wedge S_0 \overset{f_{m_1}}{\rightsquigarrow} S_1 \wedge S_1 \overset{f_{m_2}}{\rightsquigarrow} S_2 \wedge \cdots \wedge S_n \overset{f_{m_n}}{\rightsquigarrow} S_n \\ \Rightarrow \wedge \tau_{m_1}(S_0, S_1) \wedge \tau_{m_2}(S_1, S_2) \wedge \ldots \wedge \tau_{m_n}(S_{n-1}, S_n) \wedge \\ S_n \models SP_1\text{–}SP_5$$

Scope. We limit the scope of BESFS goals in two ways:

- For safety and simplicity, BESFS filesystem state and API intentionally does not include all the features in a typical full-fledged filesystem. The enclave files can be concurrently accessed by non-enclave applications, as long as the applications abide by the safety restrictions enforced by BESFS. We detect if any entity (other enclaves, user applications, the OS) violates BESFS invariants and abort the enclave.
- BESFS aims strictly at integrity property. Several known side-channels and hardware mistakes impact the confidentiality guarantees of SGX [36, 52]. Out of the 167 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. Higher-level confidentiality guarantees are not within the scope of BESFS goals (c.f. [27, 48, 49]).

3.2 BESFS Interface

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

State. BESFS has type variables which together define a state. We choose to include minimal filesystem metadata in the BESFS state while providing maximum expressiveness in its APIs. This selection is inspired by our survey of previous filesystem verification efforts for various purposes [13, 34, 44]. Specifically, BESFS state comprises valid paths in the

filesystem ($\mathcal{P}$), mappings of paths to file / directory identifiers and metadata ($\mathcal{N}$), set of open files (O), memory maps of file content ($\mathcal{M}$), memory map of anonymously mmaped page content ($\mathcal{A}$), and anonymous page mapping metadata (Q). We define them as follows:

$$\mathcal{P} \coloneqq \{\mathtt{p} \mid \mathtt{p} : \mathtt{Path}\} \quad \mathcal{N} \coloneqq \mathtt{Path} \rightharpoonup \mathtt{Id} \times \mathtt{Permission} \times \mathtt{Size}$$
$$\mathcal{A} \coloneqq \mathbb{N} \rightharpoonup \mathtt{Byte} \quad Q \coloneqq \{(\mathtt{sAddr}, \mathtt{length}) \mid \mathtt{sAddr} : \mathbb{N}, \mathtt{length} : \mathbb{N}\}$$
$$\mathcal{M} \coloneqq \mathtt{Id} \times \mathbb{N} \rightharpoonup \mathtt{Byte} \quad O \coloneqq \{(\mathtt{fileId}, \mathtt{cursor}) \mid \mathtt{fileId} : \mathtt{Id}, \mathtt{cursor} : \mathbb{N}\}$$

All file and directory paths in the filesystem are captured by path set $\mathcal{P}$, where `Path` represents the path data type. A directory path type is denoted by $\mathcal{P}_{\text{DIR}}$, whereas a file path type is denoted by $\mathcal{P}_{\text{FILE}}$. We 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 $\text{Parent}(p)$ gives the parent path `/foo/bar`. BESFS captures the information about the files and directories via the node map $\mathcal{N}$. BESFS allocates 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 fields for all the directories are always set to 0. For a path p, we use the subscript notations $\mathcal{N}(p)_{\mathtt{Name}}$, $\mathcal{N}(p)_{\mathtt{Id}}$, $\mathcal{N}(p)_{\mathtt{perm}}$, and $\mathcal{N}(p)_{\mathtt{Size}}$ to denote the name, id, permissions, and size respectively. Each open file is tracked using O via its file id. O tracks the current cursor position for the open file to facilitate operations on the file content. Given a tuple $o \in O$, for simplicity, we use subscript notations $o_{\mathtt{Id}}$ and $o_{\mathtt{Cur}}$ to denote the id and the cursor position of that file. The file content is stored in a byte memory and each byte can be accessed using the tuple comprising file id and a specific position. The anonymously mapped memory is stored in a separate byte memory and can be accessed using a position. Q tracks the anonymous memory allocations which include the start position and total length of each mapping. Thus, BESFS state $\mathsf{S}_{\text{BESFS}}$ is defined by the tuple $\langle \mathcal{P}, \mathcal{N}, Q, \mathcal{A}, O, \mathcal{M} \rangle$. Note that the BESFS API includes calls to open and close the filesystem. The user can use these calls to persist the internal state of BESFS inside the enclave for reboots and crash recovery similar to traditional filesystems [21]. More importantly, these two APIs ensure that the filesystem has temporal integrity to prevent rollbacks. BESFS ensures that the enclave sees the last saved state on reboot/restart.

State Properties. The state variables cannot take arbitrary values. They must abide by a set of state properties defined by BESFS stated in Table 2. For path set $\mathcal{P}$, BESFS enforces that the entries in the path set are unique and do not contain circular paths. 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, stated formally as SP_1. All open file IDs have to be registered in the O (SP_2). O can only have unique entries (SP_3) and the cursor of an open file handle cannot take a value larger than that file's current size (SP_4). As per SP_5, $\mathcal{M}$ and $\mathcal{A}$ do not allow any overlaps between addresses and have a one-to-one mapping from the virtual address to content. The partial functions for $\mathcal{M}$ and $\mathcal{A}$ ensures this by definition. All file operations are bounded by the file size and all anonymous memory dereferences are bounded by the size of the allocated memory. Specifically, the file 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 and is represented by the symbol $\perp$. Similarly, the current cursor position can only take values between 0 and `EOF` (SP_5).

SP_i	State Property Definition
SP_1	$\text{dom}(\mathcal{N}) = \mathcal{P} \forall (p, p') \in \mathcal{P} \times \mathcal{P}, p \neq p' \Rightarrow \mathcal{N}(p)_{\mathtt{Id}} \neq \mathcal{N}(p')_{\mathtt{Id}}$
SP_2	$\forall o \in O, \exists p \text{ s.t. } p \in \mathcal{P} \wedge \mathcal{N}(p)_{\mathtt{Id}} = o_{\mathtt{Id}}$
SP_3	$\forall (o, o') \in O \times O, o_{\mathtt{Id}} = o'_{\mathtt{Id}} \Rightarrow o = o'$
SP_4	$\forall p \in \mathcal{P}, o \in O, \mathcal{N}(p)_{\mathtt{Id}} = o_{\mathtt{Id}} \Rightarrow o_{\mathtt{Cursor}} < \mathcal{N}(p)_{\mathtt{Size}}$
SP_5	$\forall \mathtt{f}, \forall \mathtt{o}, \exists p \text{ s.t. } p \in \mathcal{P} \wedge \mathtt{f} = \mathcal{N}(p)_{\mathtt{Id}} \wedge \mathtt{o} < \mathcal{N}(p)_{\mathtt{Size}} \Rightarrow \mathcal{M}(f, o) \neq \perp$

Table 2: BESFS *State Properties. Formal definitions of the state properties enforced at any point in time.*

Transition Properties. BESFS interface specifies a set of methods listed in BESFS API in Table 3. Each of these methods takes in a valid state and user inputs to transition the filesystem to a new state. 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 must be reflected; if the operation fails, then BESFS does not reflect any change to the filesystem state.

BESFS **Safety Guarantees.** BESFS satisfies the state properties at initialization 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 ensure that BESFS state properties (SP_1-SP_5) still hold. Further, each interface itself dictates a set of constraints (e.g., the file should be opened first to close it). Thus, interface-specific properties not only ensure that the state is valid but also specify the safe behavior for each interface. Transition properties TP_1-TP_{15} (Table 3) define type map, state, and state transition for BESFS interface.

3.3 How Do Our Properties Defeat Attacks?

Our state properties in Section 3.2 and transition properties in Table 3 are strong enough to defeat the OS attacks.

File & Memory Content Manipulation (A1). Our baseline

TP_i	BESFS Interface		Pre-condition $\mathsf{Pre}_i(\mathbf{S})$	Transition Relation $\tau_i(\mathbf{S},\mathbf{S}')$		
TP_1	fs_close →	(h : Id) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_2	fs_open →	(p : Path) (h : Id, e : Error)	$p \in \mathcal{P} \wedge$ $\forall o \in \mathcal{O}, \mathcal{N}(p)_{\texttt{Id}} \neq o_{\texttt{Id}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} + \{(\mathcal{N}(p)_{\texttt{Id}}, 0)\}]$	$\wedge$	$e = \texttt{ESucc} \wedge$ $h = \mathcal{N}(p)_{\texttt{Id}}$
TP_3	fs_mkdir →	(p : Path, r : Perm) (e : Error)	$p \notin \mathcal{P} \wedge \mathsf{Parent}(p) \in \mathcal{P}_{\text{DIR}} \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} + \{p\},$ $\mathcal{N}/\mathcal{N} \bigoplus (p \mapsto \langle h, r, 0 \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_4	fs_create →	(p : Path, r : Perm) (e : Error)	$p \notin \mathcal{P} \wedge \mathsf{Parent}(p) \in \mathcal{P}_{\text{DIR}} \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} + \{p\},$ $\mathcal{N}/\mathcal{N} \bigoplus (p \mapsto \langle h, r, 0 \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_5	fs_remove →	(p : Path) (e : Error)	$p \in \mathcal{P}_{\text{FILE}} \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} - \{p\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_6	fs_rmdir →	(p : Path) (e : Error)	$p \in \mathcal{P}_{\text{DIR}} \wedge \forall q \in \mathcal{P}, \mathsf{Parent}(q) \neq p \wedge$ $\mathcal{N}(\mathsf{Parent}(p))_{\text{W}} = \texttt{True}$	$\mathsf{S}' = \mathsf{S}[\mathcal{P}/\mathcal{P} - \{p\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_7	fs_stat →	(h : Id) (r : Perm, n : String, $l : \mathbb{N}, e$: Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge p \in \mathcal{P}_{\text{FILE}}$	$\mathsf{S}' = \mathsf{S}$	$\wedge$	$e = \texttt{ESucc} \wedge$ $r = \mathcal{N}(p)_{\texttt{Perm}} \wedge$ $l = \mathcal{N}(p)_{\texttt{Size}} \wedge$ $n = \mathcal{N}(p)_{\texttt{Name}}$
TP_8	fs_readdir →	(p : Path) (l : [*String*], e : Error)	$p \in \mathcal{P}_{\text{DIR}}$	$\mathsf{S}' = \mathsf{S}$	$\wedge$	$e = \texttt{ESucc} \wedge$ $\forall n \in l, p + n \in \mathcal{P}$
TP_9	fs_chmod →	(p : Path, r : Perm) (e : Error)	$p \in \mathcal{P}$	$\mathsf{S}' = \mathsf{S}[\mathcal{N}/\mathcal{N} \bigodot (p \mapsto \langle \mathcal{N}(p)_{\texttt{Id}}, r, \mathcal{N}(p)_{\texttt{size}} \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_{10}	fs_seek →	(h : Id, $l : \mathbb{N}$) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, l)\}]$	$\wedge$	$e = \texttt{ESucc}$
TP_{11}	fs_read →	(h : Id, $l : \mathbb{N}$) (b : [Byte], e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge o_{\texttt{Cur}} + l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, o_{\texttt{Cur}} + l)\}]$	$\wedge$	$e = \texttt{ESucc} \wedge$ $b = \mathcal{M}(h, o_{\texttt{Cur}}), \ldots,$ $\mathcal{M}(h, o_{\texttt{Cur}} + l))$
TP_{12}	fs_write →	(h : Id, $l : \mathbb{N}$, b : [Byte]) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{O}/\mathcal{O} - \{o\} + \{(h, l + b_{\texttt{len}})\},$ $\mathcal{M}/\mathcal{M} \bigodot ((h, l) \mapsto b[0], \ldots,$ $((h, l + b_{\texttt{len}}) \mapsto b[b_{\texttt{len}}])]$	$\wedge$	$e = \texttt{ESucc}$
TP_{13}	fs_truncate →	(h : Id, $l : \mathbb{N}$) (e : Error)	$\exists o, o_{\texttt{Id}} = h \wedge o \in \mathcal{O} \wedge$ $\exists p, \mathcal{N}(p)_{\texttt{Id}} = h \wedge l < \mathcal{N}(p)_{\texttt{Size}}$	$\mathsf{S}' = \mathsf{S}[\mathcal{N}/\mathcal{N} \bigodot (p \mapsto \langle \mathcal{N}(p)_{\texttt{Id}}, \mathcal{N}(p)_{\texttt{perm}}, l \rangle)]$	$\wedge$	$e = \texttt{ESucc}$
TP_{14}	fs_mmap →	($l : \mathbb{N}$) ($a : \mathbb{N}, e$: Error)	$l > 0$	$\mathsf{S}' = \mathsf{S}[\mathcal{Q}/\mathcal{Q} + \{(a, l)\}, \mathcal{A}/\mathcal{A} \bigodot ([a] \mapsto 0, \ldots, [a + l - 1] \mapsto 0)]$	$\wedge$	$e = \texttt{ESucc}$
TP_{15}	fs_unmmap →	($a : \mathbb{N}$) (e : Error)	$\exists q, q_{\texttt{sAddr}} = a \wedge q \in \mathcal{Q}$	$\mathsf{S}' = \mathsf{S}[\mathcal{Q}/\mathcal{Q} - \{(a, q_{\texttt{length}})\}]$	$\wedge$	$e = \texttt{ESucc}$

Table 3: BESFS *Interface. Method API, pre-conditions, transition relations and post-conditions.* $\mathsf{S}' = \mathsf{S}[\mathcal{K}/\mathcal{K}']$ *denotes everything in* 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.*

encrypts all the file data blocks and anonymously mmapped content which prevents direct tampering from the OS. However, there are other avenues of attacks beyond this which BESFS captures. Specifically, the unique mapping property (SP_5) of $\mathcal{M}$ and $\mathcal{A}$ ensures that the OS cannot go undetected if it reorders or overlaps the underlying pages of the file content or anonymous mmaps.

Path Mismatch (A2a). BESFS state ensures that each path is uniquely mapped to a file or directory node. All methods which operate on paths first check if the path exists and if the operation is allowed on that file or 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. SP_1 ensures that file directory paths are distinct, unique, and 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. The corresponding state is updated if all the pre-conditions are met (SP_4).

File Descriptor Mismatch (A2b). Once the file is opened successfully, all file-content related operations are facilitated via the file descriptor. BESFS ensures that the mappings from the file name to the descriptor 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 such that it detects any mapping corruption attempts from the OS (SP_5).

Size Mismatch (A3). BESFS's atomicity property ensures that the filesystem completely reflects the semantics of the interface during the state transition. Our file operations have

properties which ensure that BESFS operates on the size specified in the input. `fs_read`, `fs_write`, and `fs_truncate` post-conditions reflect this in Table 3.

Error Code Manipulation (A4). All state or transition property violations in the interface execution map 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 SP_3 and TP_{11} properties are violated and BESFS return an `eBadF` error code. If there are no violations and the state transition succeeds, BESFS returns the new filesystem state and `ESucc`. BESFS interface performs its own checks to identify error states. This way, we ensure that the OS cannot go undetected if it attempts to manipulate the enclave with wrong error codes.

Iago & Libc Attacks. BESFS defends against a broader class of attacks, including Iago attacks, because we check all the return values after a file-related system call. We ensure that the values are correct by checking it against the in-enclave state of the filesystem. For anonymous mmap, BESFS checks if the untrusted memory region returned by the OS is indeed zeroed out. BESFS makes a copy of the mmaped memory inside the enclave and all accesses to the mmaped memory are redirected to the in-enclave address.

4 BESFS Implementation

BESFS defines a collection of data structures that implement the BESFS interface design in Section 3.2. Our implementation in Coq is mechanically proof-checked and is the first such system of its kind for enclaves. We build BESFS types by composition and/or induction over pre-defined Coq types `ascii`, `list`, `nat`, `bool`, `set`, `record`, `string`, `map` in Coq libraries. 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, 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. Mta represents the permissions and size metadata. We give their simplified definitions:

$$\mathsf{f} := \mathbb{N} \qquad \mathsf{d} := \mathbb{N}$$
$$\mathsf{Pg} := [\mathtt{Byte}]_{\mathrm{PG\_SIZE}} \qquad \mathsf{Pmn} := \mathsf{W} \times \mathsf{R} \times \mathsf{E}$$
$$\mathsf{Mta} := \mathsf{Pmn} \times \mathbb{N} \qquad \mathsf{PgId} := \mathbb{N}$$
$$\mathsf{Fda} := \mathtt{Str} \times \mathsf{Mta} \times [\mathsf{PgId}] \qquad \mathsf{Dda} := \mathtt{Str} \times \mathsf{Mta}$$
$$\mathsf{T} := \textsc{File}\colon \mathsf{f} \mid \textsc{Dir}\colon \mathsf{d} \times [\mathsf{T}] \qquad \mathsf{O} := [\mathsf{f} \times \mathbb{N}]\ \mathsf{Q} := [\mathbb{N} \times \mathbb{N}]$$

The BESFS filesystem layout T stores 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 the typical size of a page[1] and has a unique page number PgId. Finally, the entire

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

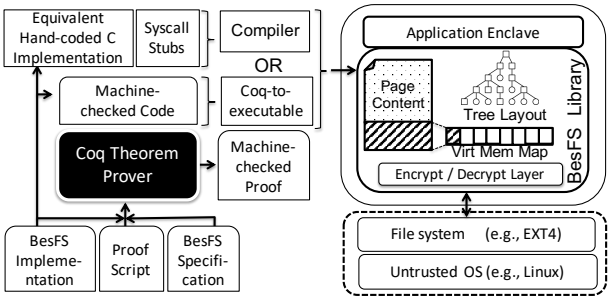


Figure 4: BESFS *Overview. Thick and dotted represents trusted and untrusted components respectively.*

filesystem memory map is stored as a list v. BESFS uses v to track the metadata for each page allocated outside the enclave to the filesystem. v does not save the actual page content of the file inside the enclave, but only saves the metadata such as file id, page id, and AES-GCM authentication tags (Figure 4). To summarize, BESFS implementation state comprises of:

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

BESFS implementation must satisfy the state properties SP_1-SP_5 and transition properties TP_1-TP_{15} outlined in Section 3.2. Table 4 summarizes the enforced invariants. Next, we discuss how we achieve this for each data structure.

Virtual Memory Map ($\mathcal{M}$). Each file is an ordered sequence of pages. BESFS assigns page ids to each page in the filesystem. BESFS virtual memory map $\mathcal{M}$ is completely independent and unrelated to the OS-allocated virtual address. For BESFS, the filesystem memory is represented by a set of virtual memory pages. Each page is a sequence of PG_SIZE bytes and is represented by a unique page id PgId. $\mathcal{M}$ tracks the virtual memory layout by storing the page metadata in the filesystem. 4000 bytes of each page comprises of the page content while the remaining 96 bytes are metadata for integrity protection and can be used to store other metadata currently not traced by BESFS. Pages are stored outside the enclave in an encrypted form and are decrypted at the enclave boundary. BESFS uses the virtual memory map $\mathcal{M}$ inside the enclave to track and verify the integrity of the page content returned by the OS. This mechanism is similar to merkle tree implementations for encrypted filesystems [51]. BESFS further ensures that a page belongs only to a single file and files do not have page overlaps. The $\mathcal{M}$ map implementation marks the unallocated page metadata slots as free in the pool.

Anonymous Memory Mapping ($\mathcal{A}$) & Handles (Q). When an anonymously mmaped memory region is first allocated in the untrusted memory, BESFS first checks if the allocation is valid i.e., the memory returned by the OS is indeed zeroed out. BESFS then makes a copy of it into its enclave protected memory. [2] During this step, BESFS registers a handle for the

[2] The scalability of such a virtual address space mapping duplication is not affected by the current limit on the EPC size (90 MB), because SGX does not limit the enclave virtual memory to 90 MB.

Virtual Memory Map	$\mathcal{M}$	$\forall i\,j, i \neq j \Rightarrow \mathsf{F}(i)[2] \cap \mathsf{F}(j)[2] = \emptyset$
Files & Directories	$\mathcal{N}$	$\textsc{Fids}(\textsc{File}: i) := [i]$ $\textsc{Fids}(\textsc{Dir}: i\,s) := \textsc{Fids}(s[1]) + \cdots + \textsc{Fids}(s[n])$ $\textsc{Dids}(\textsc{File}: i) := []$ $\textsc{Dids}(\textsc{Dir}: i\,s) := [i] + \textsc{Dids}(s[1]) + \cdots + \textsc{Dids}(s[n])$
Layout & Paths	$\mathcal{P}$	$\textsc{TreeName}(\textsc{File}: i) := \mathsf{F}(i)[0]$ $\textsc{TreeName}(\textsc{Dir}: i\,s) := \mathsf{D}(i)[0]$ $\text{NoDupName}(t : \mathsf{T}) := \exists i, t = \textsc{File}: i \vee$ $\exists d\,s, t = \textsc{Dir}: d\,s \wedge (\forall i, \text{NoDupName}(s[i])) \wedge$ $(\forall i\,j, i \neq j \Rightarrow \textsc{TreeName}(s[i]) \neq \textsc{TreeName}(s[j]))$ $\text{NoDup}([\ldots s_i \ldots s_j \ldots]) := \forall i\,j, i \neq j \Rightarrow s_i \neq s_j$
Open file handles	$\mathcal{O}$	$\textsc{Ids}([\ldots, (f_i, p_i), \ldots, (f_j, p_j), \ldots] : \mathsf{O}) := [\ldots, f_i, \ldots, f_j, \ldots]$ $\text{NoDup}(\textsc{Ids}[\ldots s_i \ldots s_j \ldots]) := \forall i\,j, i \neq j \Rightarrow s_i \neq s_j$
Anon Mmaps & Handles	$\mathcal{A}$ $\mathcal{Q}$	$\textsc{Mids}(\mathrm{Q}: i) := [i] \wedge \text{NoDup}(\textsc{MIDs}[\ldots (a_i, l_i) \ldots (a_j, l_j) \ldots]) :=$ $\forall i\,j, , i \neq j, \exists k \in (0, l_i) \Rightarrow a_j \neq a_i + k$

Table 4: BESFS *data structures definitions & invariants.*

new mapping which consists of the start address and the total length of the mapped memory. BESFS allocation ensures that the mmaped regions do not overlap with existing allocations. All accesses to the mmaped region are redirected to the protected memory. When the region is unmmaped, BESFS deletes the handle, marks the pages in protected memory as available, and relays the unmap call to the OS. Further, it ensures that the memory layout does not overlap after unmap.

Files & Directories ($\mathcal{N}$). Each file's information including the file name, the current size, and the permission bits are 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 [Pgld]. 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 maintains a map F which associates each file node Fda with a unique file identifier f. Similar to file nodes, BESFS 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 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 and each directory can have its own tree layout. BESFS does not allow cycles in the tree layout and all levels have non-duplicate directory/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.

Good State. BESFS must satisfy all the data structure invariants in Table 4 before and after any interface invocation to be in a good state. A state is good if the following holds true:

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

Known Limitations. BESFS implementation does not support a small set of filesystem operations, such as symbolic links, which are unsafe as per our safety properties. Although our currently BESFS does not reason about other metadata information such as time-stamps (e.g., `mtime`, `atime`, `ctime`). There is no fundamental limitation in adding them to BESFS for detecting potential attacks from a malicious OS. SGX does not support shared memory between enclaves. Typical enclave applications do not concurrently access protected files. Thus, we do not consider multi-enclave or concurrent access to shared enclave files. BESFS enforces an atomicity property and does not reason about APIs for explicit synchronization (e.g., `sync`, `fsync`, and `fdatasync`).[3] Nonetheless, it is compatible with them and detects any violation by the OS. We have consciously decided to not support these functionalities in our first version of BESFS to maintain simplicity.

5 BESFS Safety Proof & Modeling Challenges

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 (SP_1-SP_5) from Section 3.2 and the transition properties (TP_1-TP_{15}) outlined in Table 3. We assume BESFS is running on a hostile OS that can take any actions permitted by the hardware.

As one can readily see, our implementation uses recursive data structures and its state properties require second-order logic. For example, the BESFS filesystem layout T in Section 4 is defined 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 Gallina as well. The proofs of the statement, called *proof scripts* are written in a language called LTAC. LTAC's library of tactics, or one-line commands, encode standard proof strategies for ease of writing proofs.

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,

[3] For non-explicit synchronization, the enclave has to explicitly invoke them to ask the OS to persist the changes.

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 the calls. It 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 the 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 $\texttt{read\_dir}(\texttt{p},\texttt{n})$ and $\texttt{read\_dir}(\texttt{p},\texttt{n}+1)$ so that Coq treats them as different.

Atomicity. The purely functional nature of Coq proofs helps to prove the atomicity of each method call. In an enclave, its internal state is not accessible by the OS even if it gets interrupted; so, in a way, the enclave behaves like a pure function between two OS calls. This simplifies our proof for atomicity. 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. As a concrete example, the `write` method progressively checks 5 conditions (1: argument `id` is in the handler; 2: the specified position is correct; 3: iut writes to the copied virtual memory successfully; 4: the external call to `seek` succeeds; and 5: the external call to `write` succeeds.) before changing the state.

Non-deterministic Recursive Termination. Gallina guarantees that any theorem about a Gallina program is consistent, i.e., it cannot be both proved and disproved. Further, all programs in Gallina must terminate, since the type of the program is the statement of a theorem.[4] 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, `write_to_buffer` does not admit a syntactic check for termination, as there is a recursive call. To prove termination, via induction, we show that the input buffer size strictly reduces for each invocation of `write`. Effectively, we establish that there are no infinite chains of nested recursive calls.

Mutually Recursive Data Structures. Most of our data structure proofs are by induction 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 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)—in this case.

```
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
```

We provide an inductive statement `Tree_ind2` that is stronger than Coq-provided induction scheme `Tree_ind`, shown in the above 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 above code listing shows that the sub-property `P` is an input argument to the main property. In our full proof, a number of specific properties instantiate `P`.

External Calls to the OS. 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 decide not to terminate which constitutes as denial-of-service.

Odds & Ends. Out of the 167 lemmas, we prove 75 of them using inductions and the rest of them by logical deductions. There are two kinds of inductions in our proofs: strong induction and weak induction, the 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`. 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 (39 and 36 lemmas by strong and weak induction respectively).

We do not implement `get_next_free_page` but enforce that an implementation must satisfy the property that the new page allocated by the function 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. It is formally stated as $\texttt{new\_fid}(\texttt{t}) \notin \text{FIDS}(\texttt{t})$ and $\texttt{new\_did}(\texttt{t}) \notin \text{DIDS}(\texttt{t})$ 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 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.

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

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 5: LOC for various components of BESFS.

6 Coq to Executable Code

BESFS Coq definitions and proof script comprise 4625 LOC with 167 lemmas and 2 main theorems. The development effort for BESFS was approximately two-human years for designing the specifications and proving them. The Coq implementation has a machine-checked proof of correctness, i.e., matching the specification. The Coq code, however, needs to be converted to executable code to run in an enclave. Currently, Coq supports automatic extraction to three high-level languages: OCaml, Haskell, and Scheme [1]. We can successfully compile our code to executables; however, none of these three functional languages have runtime support for Intel SGX, primarily due to the lack of a memory manager (e.g., garbage collector) that is compatible with SGX.

Further, we have tried to run our compiled code in these three languages on existing library OSes with SGX, but without success. Specifically, we find that two state-of-the-art frameworks, Graphene-SGX [18] and Panoply [46], are not robust enough to run compiled Haskell or OCaml "hello world" programs. Our investigation reveals that supporting these functional language runtimes in entirety would require extensive foundational work, such as porting memory managers, and SGX support on existing library OSes misses several critical OS abstractions. Specifically, Graphene-SGX does not support `create_timer`, `set_timer`, `delete_timer`, and `sigaction` syscalls. We attempted to add support for these syscalls, but it is a non-trivial amount of work to add support for an entire subsystem to Graphene-SGX. In Section 9, we discuss why certified compilation from Coq to machine code is currently not practical, but a promising future direction.

With no publicly available enclave system supporting compiled programs for high-level language that Coq extracts to, we resorted to a manual line-by-line translation of our machine-checked Coq implementation to `C` code. Our `C` implementation comprises of 863 LOC core logic and 586 LOC helper functions, totaling 1449 LOC (Table 5). Our Coq code intentionally leaves out the implementation of untrusted POSIX calls. At enclave runtime, these calls have to be redirected to an actual filesystem provided by the OS (whose behavior is not trusted).

Ease of Integration. Our `C` implementation can be integrated with any SGX framework [15, 18, 46] as well as stand-alone SGX applications [28] and SGX SDK [7] (See Section 7.4). We choose Panoply as the SGX framework to integrate and test BESFS. For adding BESFS support, we wrap the application's file system 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. BESFS checks the return values and our wrapper then 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, else we flag a safety violation. We add 724 LOC to Panoply.

7 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.
- Performance of BESFS for (a) I/O intensive benchmarks; (b) CPU intensive benchmarks; (c) per-call latencies for files; and (d) real-world application workloads in typical enclave deployments.

Experimental Setup. All our experiments were conducted on a machine with Intel Skylake i7-6600U CPU (2.60 GHz, 4 cores) with 12 GB memory and 128 MB EPC of which 96MB is available to user enclaves. We execute our benchmark on Ubuntu 18.04 LTS. We use our hand-coded `C` implementation of BESFS and Panoply (unless stated otherwise) to run our benchmarks in an enclave. Panoply internally uses Intel SGX SDK Linux Open Source version 2.4 [7]. [5] BESFS uses ext4 as the underlying POSIX compliant filesystem.

Benchmarks Selection Criteria & Description. Our selection is aimed at showcasing how well BESFS fares in reaching its design goals. Since our evaluation goals for BESFS are multi-faceted, we selected a wide variety of micro-benchmarks, benchmarks, and real-world applications. First, we use the micro-benchmark suite from FSCQ [21]. It comprises workloads to test each file-related system call via different sequences of filesystem operations on large and small files. Second, we use IOZone [42], a well-known and a broad filesystem benchmark for measuring bandwidth for different file access patterns with 13 tests for 7 standard operations. Third, for testing BESFS on non-I/O intensive applications, we use CPU-intensive programs from SPEC CINT2006 [8]. 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

[5] We have also benchmarked BESFS on Ubuntu 14.04, SGX SDK 1.6.

LibC Calls	SPEC CINT 2006							FSCQ		Total
	astar	mcf	bzip2	hmmer	libqu	h264	sjeng	small	large	
BESFS Core Calls										
open	3	0	1	0	0	7	0	2	1	14
read	27	0	4	0	0	129	0	1	3072	3233
write	0	0	0	0	0	0	0	1	66560	66561
lseek	0	0	0	0	0	75	0	0	66563	66638
remove	0	0	0	0	0	0	0	2	1	3
close	3	0	1	0	0	7	0	2	1	14
mkdir	0	0	0	0	0	0	0	100	0	100
BESFS Auxiliary Calls										
fopen	1	2	0	5	0	6	1	0	0	15
fread	1	0	0	1	0	1	0	0	0	3
fwrite	0	1035	0	6	0	13	2	0	0	1056
fgets	0	90435	0	108	0	0	5	0	0	90548
fscanf	12	0	0	0	0	24	0	0	0	36
fprintf	0	5985	0	605	0	17	162	0	0	6769
fseek	0	0	0	0	0	2	0	0	0	2
ftell	0	0	0	4	0	1	0	0	0	5
rewind	0	0	0	3	0	0	0	0	0	3
Unsafe Calls										
fsync	0	0	0	0	0	0	0	0	2	2
rename	0	0	0	0	0	0	6	0	0	6
Total	47	97457	6	732	0	282	176	108	136200	**235008**

Table 6: 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.*

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. Fourth, we use all applications from Panoply—a system to execute legacy applications in enclaves. These 4 real-world applications (H2O web server, TDS database client, OpenSSL library, and Tor) have a mix of CPU, memory, and file, and network IO workloads. We successfully port $3/4$ case-studies to BESFS (see Section 7.4 for details) and use the same workloads as that in Panoply [46]. Lastly, we select all the 10 real-world applications from Privado [28] which perform inference over CIFAR10 and ImageNet using state-of-the-art neural network models. Thus, our final evaluation is on a total of 31 applications: (a) 10 programs from FSCQ for micro-benchmarking per-call latencies for file operations, (b) IOZone and 7 programs from SPEC for measuring the overhead of BESFS on IO-intensive and CPU-intensive benchmarks. (c) 3 applications from Panoply and 10 applications from Privado for demonstrating the effect of BESFS on real-world enclave usage. All our results are aggregated over 5 runs.

7.1 Expressiveness & Compatibility

We empirically demonstrate that if the underlying filesystem and the OS are POSIX compliant and benign then BESFS is not overly restrictive in the safety conditions. We first analyze all syscalls and `libc` calls made by our benchmarks for various workloads using `strace` and `ltrace` respectively. We then filter out the fraction of filesystem related calls. Table 6 shows the statistics of the type of filesystem call and its frequency for our benchmarks. We observe a total of 235008 filesystem calls comprising of 18 unique APIs. BESFS can protect $235000/235008$ of them.

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 7: Expressiveness of BESFS. *Row represents a* `libc` *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. ✓ *represents that the API is used to compose* `libc` *API.*

Compositional Power of BesFS. BESFS directly reasons about 15 calls using the core APIs outlined in Section 3.2. We use BESFS's composition theorem and support additional 22 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 7). Since we do not know the location of the newline character, we read the input file character-by-character and stop when we see a new line, EOF, or if the buffer size reaches the value `size`. Similarly, we already know the total size of the buffer when writing the content to the output file (e.g., after resolving the format specifiers in `fprintf`). Thus we write the complete buffer in one single call. `libc` calls use flags to dictate what operations the API must perform. For example, the application can use the `fopen` API to open a file for writing. If the application specifies the append flag ("`a`"), the library creates the file if it does not exist and positions 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. We implement and support a total of 16 flags in total for our 3 APIs which require flags. Our implementation

currently supports the common flags used by applications and can be extended in the future using our core APIs.

BESFS does not reason about the safety of the remaining 2 APIs which amount to a total of 8 calls in our benchmarks. Although BESFS does not support these unsafe calls, it still allows the enclave to perform those calls. 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 such as `sync`, `fsync`, and `fdatasync` have to be implicitly invoked for the OS after each function call to persist the changes. We experimentally confirm that the program produces the same output with and without BESFS, thus reaffirming that our safety checks do not alter the program behavior.

7.2 Do Proofs Help in Eliminating Bugs?

We encountered many mistakes and eliminated them during the development as a part of our proof experience. This highlights the importance of a machine-proved specification.

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 had ignored the corner case by mistake: when the file is just created with 0 bytes size, the only valid position is also 0.

Example 2: `write` Implementation Bug. BESFS's `write` function input includes the position (`pos`) at which the buffer is to be written. In our initial Coq implementation of `write`, we used the name `pos` for the cursor stored in the open handles (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. This bug in `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. Panoply's `fread` and `fwrite` calls pass the size of the buffer and a pointer to the buffer. BESFS piggybacks on these 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 ($>$ 64 KB 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 ($<$ 64 KB) to read / write large buffers. After our fix, the implementation passed BESFS checks.

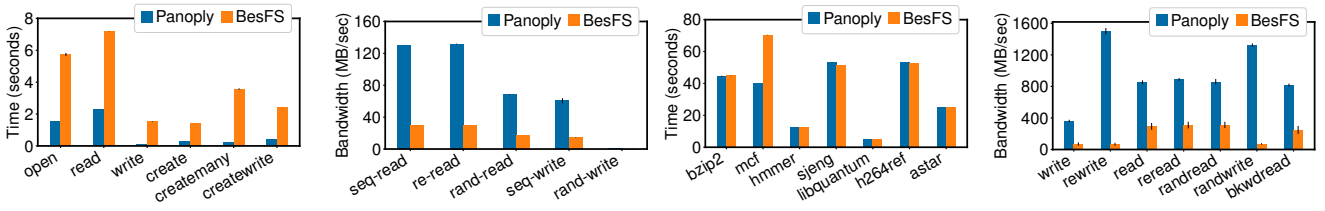
Example 4: Panoply Error Code Bugs. According to `fopen` POSIX specification, the function fails with `ENOENT` if the filename does not name an existing file or is an empty string. When we used Panoply's `fopen` interface, it did not return the expected error code when the file did not exist. Our BESFS check after the external call flagged a warning of a safety condition violation because BESFS did not have a record of this file but the external call claimed that the file existed. On investigation, we 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. We tested against the 7 attacks / bugs in Panoply after integrating BESFS to ensure that it did not violate any invariants.

Simulating a Malicious OS. First, we hand-crafted a suite of around 687 tests cases in the form of `assert` statements embedded in 40 test-driver `C` programs that make a series of filesystem calls. To generate these asserts and test drivers, we took our proof invariants and systematically generated asserts which checked the given constraint. We then coded the tests along with the assert statements. Second, to simulate the malicious OS, we manually crafted and planted known-bad return values at the system call interface. We semi-randomly generated these values, similar to SibylFS [44]. When simulating the OS, it does not matter if the victim binary is executing inside or outside of an enclave. This observation simplified our testing setup. For a clean way to hook on the syscalls and libc calls made by our victim test-driver programs, we used the `ld_preload` environment variable to optionally link the test case victim binaries with our malicious syscall and libc return values.[6] We then performed three sets of executions of the victim binaries: (a) without our malicious library and without BESFS for ensuring that the victim binary executes in the baseline case and recording the benign path for a given input; (b) with our malicious library but without BESFS to show that the lack of checks causes the victim binary to execute unintended paths i.e., assertion failures; (c) with our malicious library and BESFS to check if BESFS can detect the bad return values. We investigated the resulting assertion failures in these runs. We report that all of the failures observed in (b) but not (a) were due to lack of checks; while they did not occur in case (c). This shows that BESFS invariants were able to prune all the planted bad return values.

7.3 Performance

BESFS is the first formally verified filesystem for SGX. Although our primary goal is not performance, we report performance on our benchmarks for completeness. First, we re-

[6] Another way is to write a malicious Linux kernel module to intercept calls made by the victim enclave binary.



(a) FSCQ *Single Syscalls.* *(b)* FSCQ *Large IO.* *(c) SPEC CINT 2006.* *(d) IOZone.*

Figure 5: BESFS *Performance on micro-benchmarks, standard CPU, and IO benchmarks with respect to Panoply. (a) Execution overhead for each system call in* FSCQ. *(b) File operation bandwidth reported by* FSCQ *large IO suite. (c) Execution overhead on SPEC2006 CPU benchmarks. (d) File operation bandwidth reported by IOZone benchmarks.*

port the per-call latencies and file access pattern latencies with the FSCQ micro-benchmark. Our main take away from this experiment is that BESFS overhead is dominated by the encryption-decryption of the file content. Next, we demonstrate this phenomenon systematically by reporting 12.22% overhead and 4.8× bandwidth slowdown on standard CPU (SPEC CINT2006) and I/O benchmarks respectively. Lastly, we report the overheads on real-world applications in Section 7.4. Future optimizations can use BESFS API specification as an oracle for golden implementation.

Micro-benchmarking Single File-related Operations. We use FSCQ to measure the per-system call overhead of BESFS. Figure 5a shows that it averages to 3.1×. We observe that read-write operations incur a large overhead. The read operation is slowed down by 3.7× and create+write is 5.4× slower because 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 at least a page of encryption-decryption and integrity computation.

Micro-benchmarking Access Patterns. Next, we run all the large tests in FSCQ with 8 KB block size, 1 KB I/O transfer size, and 1 MB total file size. FSCQ performs a series of sequential write, sequential read, re-read, random read, random write, multi-write, and multi-read operations. We perform each type of operation 100K times on the files. We observe an average overhead of 6.7× because of BESFS checks. Figure 5b shows 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 are more expensive because each read / write may cause a page-level encryption-decryption. BESFS does not cache page content so re-reads and sequential reads incur similar overheads.

I/O Intensive Benchmark: IOZone. We use IOZone to test BESFS for file sizes up to 512 KB while varying the record size from 4 KB to 512 KB and report the aggregate performance in Figure 5d. We observe an average of 4.8× decrease in the IO bandwidth over all the operations. Write operations are significantly more expensive in comparison to reads. This is because BESFS performs reads over the page for decrypting the content and then does a write, which requires encryption.

CPU Intensive Benchmark: SPEC CINT2006. SPEC benchmarks take in a configuration file and optionally an input file to produce an output file. Figure 5c and shows the performance per-application overhead; the average overhead is 12.22%. `hmmer`, `href`, `sjeng`, and `libquantum` have relatively less overhead whereas `astar`, `bzip2`, and `mcf` exhibit larger overhead. `astar` and `mcf` use `fscanf` to read the configuration files. Thus, reading each character read leads to a page read and corresponding decryption and integrity check. Further, `astar` reads a binary size of 65 KB for processing. As shown by our single syscall measurements (Figure 5a), reads are expensive. Both these factors amplify the slowdown for `astar`. `bzip2` and `mcf` output the benchmark results to new files 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.

7.4 Real-world Case Studies

We showcase the ease of integration and usage of BESFS in real-world enclave programs with two case-studies: (a) 4 applications from Panoply; (b) 10 applications from Privado [28] which is built directly on Intel SGX SDK.

Secure Micron Execution with Panoply. We use the 4 applications from the Panoply paper and evaluate them under the same workloads [46]. We do not observe any significant slowdown for OpenSSL(±0.2%) and Tor nodes (±0.8%). Both these applications use file operations to load configurations (e.g., signing keys, certificates, node information) only once during their lifetime, while the rest of the execution does not interact with files. On the other hand, we observe an overhead of 72.5% for the FreeTDS client. We attribute this overhead to the nature of the application which performs file operations for each of the 48 SQL queries in the benchmarks. Lastly, we report that the H2O web server logic violates BESFS safety properties. Specifically, H2O duplicates the file descriptors across worker threads and concurrently accesses the file content to be served to the clients. Thus, we deem H2O as non-compatible with BESFS.

Secure Inference with Privado. As a second case study, we

integrate BESFS with Privado [28]—an SGX-compatible machine-learning framework. It uses Torch library to infer labels of images from standard datasets using 10 well-known deep neural net architectures (LeNet, VGG19, Wideresnet, Resnet110, Resnext29, AlexNet, Squeezenet, Resnet50, Inceptionv3, and Densenet). These applications vary from 230 LOC to 13.4 KLOC and have enclave memory footprint between 0.6 MB to 392 MB. We use Cifar-10 and ImageNet datasets, as done in Privado, where each image is 3.1 KB and 155.6 KB respectively. For each of the application, we integrate BESFS interface with 20 LOC changes to Privado. We observe an overhead of $\pm 1\%$ relative to the baseline for all the networks and their corresponding datasets. We see such low overheads because, unlike Panoply, Privado decrypts the file input after reading it. Thus, the baseline includes the cost of decryption. In this case, BESFS only adds a fixed startup cost of checks proportional to the number of file operations on the input file and the number of images in a batch, while keeping the decryption time constant across both the systems. This shows that BESFS is compatible and easy to integrate with enclaves which already use file encryption-decryption.

8 Related Work

We survey the existing SGX defenses including verification as well as filesystem hardening work in the non-SGX setting.

SGX Attacks & Defenses. BESFS ensures the filesystem integrity based on hardware integrity guarantees of SGX. It assumes the confidentiality of SGX only in one lemma, i.e., the secrecy of a cryptographic key. This is an important design choice in light of the side-channels [36, 39, 45, 52]. BESFS assumes secure hardware implementation and is agnostic to confidentiality defenses [29].

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 (a) port the entire filesystem inside the enclave [11, 33]; (b) keep the files encrypted outside the enclave [15, 18, 46] and, for each return parameters, check the data types, bounds on the IO buffers, and valid value ranges of API specific values (e.g., error codes, flags, and structures). As one concrete comparison, Intel SGX SDK PFS Library [4] is dedicated solely to the filesystem layer. Although it leaves the enclave vulnerable to Iago-like attacks as we showed in Section 2.3, it is better than approaches which bloat the TCB to support all syscalls. It is not transparent to existing legacy applications; the enclave has to use APIs with the non-standard interface for explicit key management (e.g., `sgx_fopen_auto_key`) as well as traditional file operations (e.g., `sgx_fopen(filename,mode,key)`). More importantly, while these systems reduce the attack surface of file syscall return value tampering, none of them provably thwart all the attacks in Section 2.2. Other filesystems with untrusted OS in a non-enclave setting are not formally verified [37].

Verified Guarantees for Enclaves. Formal guarantees have been studied for enclaved applications to some extent. They provide provable confidentiality guarantees for pieces of code executing inside the enclave. Most notably, Moat [49], /Confidential [48], and IMPe [27] formally model various adversary models in SGX and ensures that the enclave code does not leak confidential information. These confidentiality efforts are orthogonal to BESFS's integrity goals. Another line of verification research has focused on certifying the properties of the SGX hardware primitive itself, which BESFS assumes to be correctly implemented. Komodo [25] is a formally specified and verified monitor for isolated execution which ensures the confidentiality and integrity of enclaves. TAP [50] does formal modeling and verification to show that SGX and Sanctum [24] provide secure remote execution which includes integrity, confidentiality, and secure measurement. The existing works on verified filesystems do not reason about an untrusted OS so they cannot be simply added on top of these enclave systems. BESFS is above these hardware abstractions.

Filesystem Verification. Formal verification for large-scale systems such as operating systems [30, 35], hypervisors [12], driver sub-systems [20] and user-applications [31] has been a long-standing area of research. None of these works consider a Byzantine OS, which leads to completely different modeling of properties. Filesystem verification for benign OS, however, is in itself a challenging task [34]. This includes building abstract specifications [26], systematically finding bugs [53], POSIX non-compliance [44] in filesystem implementations, end-to-end verified implementations [13], crash consistency [17], and crash recovery [21].

9 Discussion

While BESFS has a machine-checked Coq implementation of our filesystem API specification, it would be desirable to have machine-checked enclave-executable code. We believe this is feasible, in principle, but requires significant advances in state-of-the-art certified language techniques to become immediately practical. There are at least three different promising future work directions to enable certified executable BESFS code: (1) directly certifying the enclave machine code [2]; (2) using a certified compiler to convert Coq code to machine code [14]; and (3) using a simulation proof of `C` or machine code implementation with the Coq code in the spirit of K [5].

Compiling Coq to `C`. The most promising direction is to have certified compilation from Coq to `C` code and then from `C` to machine code. CertiCoq [14] is a certified compiler from

Gallina (Coq) to CompCert-C. CompCert [40] is one of the most mature certified C compiler which ensures that the generated machine code for various processors behaves exactly as prescribed by the semantics of the source program. With help from the CertiCoq team, we report that we have successfully compiled BESFS to executable C code. However, we point out that CertiCoq is a very early stage compiler at present. The produced code is incomplete which causes segmentation faults. Further, it cannot be interfaced with external function calls (e.g. system calls) due to missing foreign function interfaces (FFI). Nonetheless, we expect that as CertiCoq matures, certified machine code for BESFS (and similar systems) will become a practical possibility.

Verified Machine Code. The second possibility is to verify the machine code directly. Given that BESFS is written at a higher level of abstraction (Gallina), our subsequent verification has to reason about the language abstraction gap between Gallina and machine code. Coq supports extraction to OCaml, Haskell, Scala, and C. The most mature extraction techniques are to OCaml and Haskell, so we tried to port their runtimes to SGX. For reasons reported in Section 6, porting such language runtimes to SGX certifiably merits a separate research effort in its own right.

Bisimulation. A third possibility is a bisimulation of the C or machine code and the Coq code. For maintaining such proofs, when the BESFS specification expands in the future, the best way is to specify the operational semantics of the machine code (or C) and Coq in a common framework. We believe this is possible but entails significant future work.

10 Conclusion

BESFS is the first formally proved enclave specification and implementation for integrity-protecting POSIX filesystem API. BESFS API is expressive to support real applications, minimizes the TCB, and eliminates bugs.

Acknowledgments

We thank our shepherd Vasileios Kemerlis and the anonymous reviewers for their feedback; Andrew Appel and Olivier Belanger for discussions and help with CertiCoq; Privado team at Microsoft Research for sharing the torch code for our case-study; Shruti Tople, Shiqi Shen, Teodora Baluta, and Zheng Leong Chua for their help on improving earlier drafts of the paper. 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. This work was funded in part by Yale-NUS College grant R-607-265-322-121. This material is in part based upon work supported by the National Science Foundation under Grant No. DARPA N66001-15-C-4066 and Center for Long-Term Cybersecurity. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Availability

BESFS specification and implementation in Coq is available at https://shwetasshinde24.github.io/BesFS/

References

[1] Code Extraction from Coq. https://coq.inria.fr/library/Coq.extraction.Extraction.html.

[2] Frama-C. https://frama-c.com/index.html.

[3] Google Asylo. https://asylo.dev.

[4] Intel Protected File System Library Using SGX. https://software.intel.com/en-us/sgx-sdk-dev-reference-intel-protected-file-system-library.

[5] K Framework. http://www.kframework.org.

[6] Open Enclave SDK. https://openenclave.io/.

[7] SGX SDK. https://github.com/intel/linux-sgx/.

[8] SPEC 2006. https://www.spec.org.

[9] Tor. https://www.torproject.org.

[10] Syscall wrappers should sanity-check return values from untrusted ocalls · issue #21 · keystone-enclave/keystone-runtime. https://github.com/keystone-enclave/keystone-runtime/issues/21, August 2019.

[11] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee. OBLIVIATE: A Data Oblivious File System for Intel SGX. NDSS'18.

[12] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova. Automated Verification of a Small Hypervisor. VSTTE'10.

[13] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. Heiser. Cogent: Verifying High-Assurance File System Implementations. ISCA'16.

[14] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. CoqPL'17.

[15] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure Linux Containers with Intel SGX. OSDI'16.

[16] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. OSDI'14.

[17] J. Bornholt, A. Kaufmann, J. Li, A. Krishnamurthy, E. Torlak, and X. Wang. Specifying and checking file system crash-consistency models. ASPLOS '16.

[18] C. che Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. ATC'17.

[19] S. Checkoway and H. Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. ASPLOS '13.

[20] H. Chen, X. N. Wu, Z. Shao, J. Lockerman, and R. Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. PLDI '16.

[21] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. SOSP '15.

[22] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. ASPLOS'08.

[23] V. Costan and S. Devadas. Intel SGX Explained. ePrint 2016/086.

[24] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. USENIX Security'16.

[25] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. SOSP'17.

[26] P. Gardner, G. Ntzik, and A. Wright. Local Reasoning for the POSIX File System. ESOP'14.

[27] A. Gollamudi and S. Chong. Automatic Enforcement of Expressive Security Policies Using Enclaves. OOPSLA'16.

[28] K. Grover, S. Tople, S. Shinde, R. Bhagwan, and R. Ramjee. Privado: Practical and Secure DNN Inference with Enclaves. *CoRR*, abs/1810.00602, 2019.

[29] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. USENIX Security'17.

[30] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. OSDI'16.

[31] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end Security via Automated Full-system Verification. OSDI'14.

[32] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. ASPLOS '13.

[33] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. OSDI'16.

[34] G. Keller, T. Murray, S. Amani, L. O'Connor, Z. Chen, L. Ryzhyk, G. Klein, and G. Heiser. File Systems Deserve Verification Too! PLOS '13.

[35] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. SOSP '09.

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. S&P'19.

[37] Y. Kwon, A. M. Dunn, M. Z. Lee, O. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. ASPLOS'16.

[38] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song. Keystone: An Open Framework for Architecting TEEs. *CoRR*, abs/1907.10119, 2019.

[39] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. USENIX Security'17.

[40] X. Leroy. The CompCert verified compiler. http://compcert.inria.fr/.

[41] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. HASP '13.

[42] W. Norcott and D. Capps. IOzone Filesystem Benchmark.

[43] D. R. K. Ports and T. Garfinkel. Towards Application Security on Untrusted Operating Systems. HotSec'08.

[44] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems. SOSP '15.

[45] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing Page Faults from Telling Your Secrets. ASIACCS'16.

[46] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux Applications With SGX Enclaves. NDSS'17.

[47] S. Shinde, S. Tople, D. Kathayat, and P. Saxena. PodArch: Protecting Legacy Applications with a Purely Hardware TCB. Technical report, National University of Singapore, February 2015.

[48] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani. A design and verification methodology for secure isolated regions. PLDI '16.

[49] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying Confidentiality of Enclave Programs. CCS '15.

[50] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. CCS '17.

[51] S. Tople, A. Jain, and P. Saxena. LEVEEFS: Securing Access to Untrusted Filesystems in Enclaved Execution. Technical report, National University of Singapore, 2015.

[52] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. S&P'15.

[53] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. OSDI'04.