I^{3}FS: An In-Kernel Integrity Checker and Intrusion Detection File System

Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok
Stony Brook University

Appears in the proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)

Abstract

Today, improving the security of computer systems has become an important and difficult problem. Attackers can seriously damage the integrity of systems. Attack detection is complex and time-consuming for system administrators, and it is becoming more so. Current integrity checkers and IDSs operate as user-mode utilities and they primarily perform scheduled checks. Such systems are less effective in detecting attacks that happen between scheduled checks. These user tools can be easily compromised if an attacker breaks into the system with administrator privileges. Moreover, these tools result in significant performance degradation during the checks.

Our system, called I^{3}FS, is an on-access integrity checking file system that compares the checksums of files in real-time. It uses cryptographic checksums to detect unauthorized modifications to files and performs necessary actions as configured. I^{3}FS is a stackable file system which can be mounted over any underlying file system (like Ext3 or NFS). I^{3}FS's design improves over the open-source Tripwire system by enhancing the functionality, performance, scalability, and ease of use for administrators. We built a prototype of I^{3}FS in Linux. Our performance evaluation shows an overhead of just 4% for normal user workloads.

1 Introduction

In the last few years, security advisory boards have observed an increase in the number of intrusion attacks on computer systems [2]. Broadly, these intrusions can be categorized as network-based or host-based intrusions. Defense against network-based attacks involves increasing the perimeter security of the system to monitor the network environment, and setting up firewall rules to prevent unauthorized access. Host-based defenses are deployed within each system, to detect attack signatures or unauthorized access to resources. We developed a host-based system which performs integrity checking at the file system level. It detects unauthorized access, malicious file system activity, or system inconsistencies, and then triggers damage control in a timely manner.

System administrators must stay alert to protect their systems against the effects of malicious intrusions. In this process, the administrators must first detect that an intrusion has occurred and that the system is in an inconsistent state. Second, they have to investigate the damage done by attackers, like data deletion, adding insecure Trojan programs, etc. Finally, they have to fix the vulnerabilities to avoid future attacks. These steps are often too difficult and hence machines are mostly reinstalled and then reconfigured. Our work does not aim at preventing malicious intrusions, but offers a method of notifying administrators and restricting access once an intrusion has happened, so as to minimize the effects of attacks. Our system uses integrity checking to detect and identify the attacks on a host, and triggers damage control in a timely manner.

In our approach, given that a host system has been compromised by an attack, we aim at limiting the damage caused by the attack. An attacker that has gained administrator privileges could potentially make changes to the system, like modifying system utilities (e.g., `/bin` files or daemon processes), adding back-doors or Trojans, changing file contents and attributes, accessing unauthorized files, etc. Such file system inconsistencies and intrusions can be detected using Tripwire [9, 10, 22]. Tripwire is one of the most popular examples of user mode software that can detect file system inconsistencies using periodic integrity checks. There are three disadvantages of any such user-mode system: (1) it can be tampered with by an intruder; (2) it has significant performance overheads during the integrity checks; and (3) it does not detect intrusions in real-time. Our work uses the Tripwire model for the detection of changes in the state of the file system, but does not have these three disadvantages. This is because our integrity checking component is in the kernel.

In this paper we describe an in-kernel approach to detect intrusions through integrity checks. We call our system **I^{3}FS** (pronounced as *i-cubed FS*), which is an acronym for **I**n-kernel **I**ntegrity checker and **I**ntrusion detection **F**ile **S**ystem. Our in-kernel system has two major advantages over the current user-land Tripwire.

First, on discovering any failure in integrity check, I^3FS immediately blocks access to the affected file and notifies the administrator. In contrast, Tripwire checks are scheduled by the administrator, which could leave a larger time-period open for multiple attacks and can potentially cause serious damage to users and their data. Second, I^3FS is implemented inside the kernel as a loadable module. We believe that the file system provides the most well-suited hooks for security modules because it is one level above the persistent storage and most intrusions would cause file system activity.

In addition to providing these advantages over Tripwire, our system is implemented as a stackable layer such that it can be stacked on top of any file system. For example, we can use stacking over NFS to provide a network-wide secure file system as well. Finally, it is easier to compromise user-level tools (like Tripwire) than instrumenting successful attacks at the kernel level.

We used a stackable file system template generated by FiST [28] to build an integrity checking layer which intercepts calls to the underlying file system. I^3FS uses cryptographic checksums to check for integrity. It stores the security policies and the checksums in four different in-kernel Berkeley databases [8]. During setup, the administrator specifies detection policies in a specific format, which are loaded into the I^3FS databases. File system specific calls trigger the integrity checker to compare the checksums for files that have an associated policy. Based on the results, the action is logged and access is allowed or denied for that file. Thus, our system design uses on-access, real-time intrusion detection to restrict the damage caused by an intrusion attack.

The rest of the paper is organized as follows. Section 2 outlines the design goals and architecture of I^3FS. Section 3 discusses the implementation specifics and key operations of I^3FS. Section 4 presents the evaluation of the system. Section 5 discusses related work. We conclude in Section 6 and discuss future directions.

2 Design

Checksumming using hash functions is a common way of ensuring data integrity. Recently, the use of cryptographic hash functions has become a standard in Internet applications and protocols. Cryptographic hash functions map strings of different lengths to short fixed size results. These functions are generally designed to be collision resistant, which means that finding two strings that have the same hash result is impractical. In addition to basic collision resistance, functions like MD5 [19] and SHA1 [4] also offer randomness, unpredictability of the output, etc. In I^3FS, we use MD5 for computing checksums.

We have designed I^3FS as a stackable file system [26]. File system stacking is a technique to layer new functionality on top of existing file systems, as can be seen in Figure 1. With no modification to the lower level file system, a stackable file system operates between the virtual file system (VFS) and another file system. I^3FS intercepts file system calls and normally passes them to the lower level file system; however, I^3FS also injects its integrity checking operations and based on return values to system calls, it affects the behavior that user applications see.

When designing I^3FS, we aimed at offering a good balance between security and performance. We offer configurable options that allow administrators to tailor the features of I^3FS to their site needs, trading off functionality for performance.

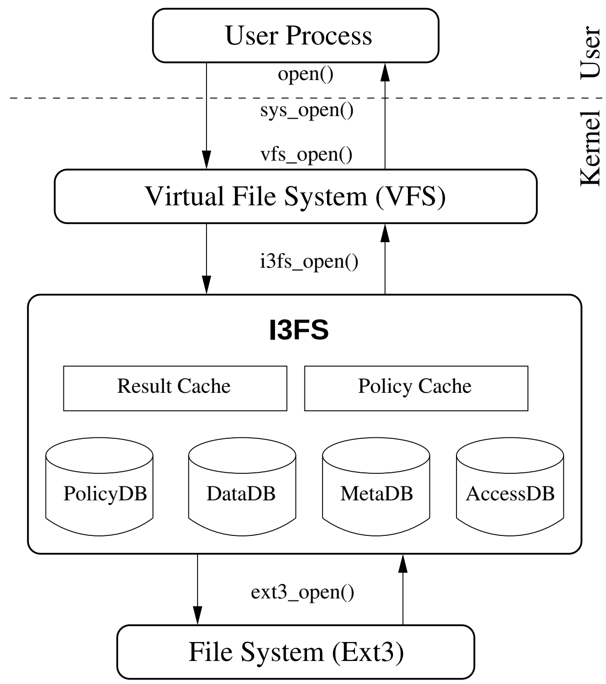

Figure 1: I^3FS Architecture

2.1 Threat Model

I^3FS is primarily aimed at detecting the following:

- Malicious replacement of vital files such as the ones in the `/bin` directory. Attackers could replace programs such as `ls` and `ps` with Trojans, without the knowledge of the system administrators. These kind of attacks can be tracked and prevented through I^3FS by setting up appropriate policies for important files.
- Unauthorized modification of data by an eavesdropper in the network, in the case of remote file systems, where the client file system communicates with the disk over an insecure network.
- Corruption of disk data due to hardware errors. In-

expensive disks such as IDE disks silently corrupt data stored in them due to magnetic interference or transient errors. These errors cannot always be detected by normal file systems. I^3FS can notify the administrator about disk corruption if there is a suitable policy associated with the file.

2.2 Policies

The two main goals we considered when designing the policies for I^3FS were versatility and ease of use. The policy syntax provided by I^3FS is similar to the user level Tripwire [10]. The general format of an I^3FS policy is as follows:

```
{-o|-e|-x} OBJECT -m FLAGS -p PROPERTIES
-a ACTION [-g GRANULARITY] [-f FREQ] [-r]
```

where,

- `-o` *`OBJECT`* specifies the object (file or directory) for which the rule is valid. If the object is a directory, then the rule applies recursively to all the files and sub-directories. The `-e` option is used to exclude an object (in most cases sub-directories) from the integrity checks, and the `-x` option is used to remove a policy.
- `-m` *`FLAGS`* represents the set of attributes of the respective object, used to calculate the checksum. The supported attributes are as follows:

  p *Permission and file mode bits*
  i *Inode Number*
  n *Number of Links*
  u *User id of the owner*
  g *Group id of the owner*
  s *File size*
  d *ID of the device on which the inode resides*
  b *Number of blocks allocated*
  a *Access time*
  m *Modification time*
  c *Inode change time*

- `-p` *`PROPERTIES`* represents the properties of the policies, used to calculate the checksum. The properties offered are as follows:

  D *Checksum file data*
  I *Inherit the policies for new files*

- `-a` *`ACTION`* determines the action taken if the integrity check failed. Our I^3FS implementation supports only two actions: BLOCK and NO-BLOCK. The BLOCK action returns a failure for any attempt to access the respective file and alerts the administrator about the inconsistency of this critical resource. The NO_BLOCK action lets the operation go through I^3FS to the underlying file system. All integrity check failures are logged in the I^3FS system.
- `-g` *`GRANULARITY`* specifies whether the checksumming is done on a per page basis or for the entire file at once. The available granularity options are PER_PAGE or WHOLE_FILE. PER_PAGE is useful for mostly-random file access patterns, and WHOLE_FILE is useful for mostly sequential small-file access patterns.
- `-f` *`FREQ`* is an integer value that determines the frequency of integrity checks. For example, a value of 50 for frequency would make I^3FS perform integrity checking for the file every 50 times it is opened. This option is available only if WHOLE_FILE checksumming is chosen.
- `-r` can be applied only if the object is a directory. This applies the policy recursively to all the files below the directory tree specified in *`OBJECT`*.

We have chosen the set of policy options such that it helps detect most kinds of attacks on the file system. Checksumming different fields of the meta data of files helps detect whether important files have been re-written by malicious programs through the file system. Checksumming file data helps detect unauthorized modification of data possibly made without the knowledge of the file system. An example of this is a malicious process that can write to the raw disk device directly in Unix-like operating systems.

2.3 I^3FS Databases

I^3FS configuration data is stored in four different in-kernel databases. KBDB [8] is an in-kernel implementation of the Berkeley DB [21]. Berkeley DB is a scalable, high performance, transaction-protected data management system that efficiently and persistently stores ⟨`key`, `value`⟩ pairs using hash tables, B+ trees, or queues. I^3FS stores four databases in the B+ tree format, so that we benefit from locality. The schema for the four databases is given in Table 1.

Database	Key	Value
`policydb`	inode#	Policy bits, freq#
`datadb`	inode#, page#	Checksum value
`metadb`	inode#	Checksum value
`accessdb`	inode#	Counter value#

Table 1: I^3FS: Database Schemas

Having separate databases for storing the data and meta data checksums is advantageous in certain situations. Generally, we expect that meta data checksumming would be used more commonly than data checksumming for two reasons. First, almost all modifications to a file made through the file system will result in modifications to its meta data. Second, meta data checksumming is less time-consuming than data checksumming as the number of bytes to be checksummed is smaller. Therefore, having the data and meta data checksums in two different databases results in less I/O and more efficient cache utilization as data checksums need not be fetched along with meta data checksums.

The policy database (`policydb`) contains the policy options associated with the files and optionally the frequency of check values. We use the inode number to refer to the policies instead of the path names so as to avoid unnecessary string comparisons. We have a user level tool that reads the policy file and populates the policy database. Further details about initialization and setup are given in Section 3. The policy database has the inode number of the file as the key, and the data is either a 4 byte or an 8 byte value containing the policy bits and optionally the frequency of integrity checks (if the frequency of checks policy option is chosen).

The data checksum database (`datadb`) contains the checksums of file data for those files that have a policy option for checksumming their data. Since there are two sub-options for checksumming file data, the per page and the whole file checksumming, this database either contains N of checksums for a single file, where N is the number of pages in the file, or a single checksum for the entire file. The inode number and the page number form the key for this database. If the option is to checksum the whole file, then the single checksum value will be indexed with page number zero. The data checksum database is populated during the I^{3}FS initialization phase through an `ioctl`, when the policies are added to the policy database.

The meta data checksum database (`metadb`) has a simple design. The key is the inode number and the data is the checksum value for the set of fields of the inode that are specified in the policy options. Information about the set of fields that are checksummed is not stored in the meta data checksum database. Instead it is retrieved from the policy database that stores the policy bits. This database is also populated during the initialization phase which we discuss in Section 3.

The access counter database (`accessdb`) contains a counter that represents the number of times a file has been opened after the last time it was checked for integrity. This is useful to set custom numbers for frequency of checks, so that less important files need not be checked for integrity every time they are accessed. For files that have a policy indicating a custom frequency of check number, every read will result in getting the previous counter value, increasing it by one and saving the new value, if the counter has not exceeded the frequency limit.

2.4 Caching in I^{3}FS

In I^{3}FS, each file access is preceded by a check whether that file has an associated policy or not. We expect that the number of files that have policies will be much less than the number of files without policies. Hence, it is important that we optimize for the common case of a file without an associated policy. Second, for those files that have policies and are accessed frequently, checksums should not be re-computed on each access.

I^{3}FS caches two kinds of information. First, whether a given file has a policy associated with it or not, and if so, the policy for that file. Second, it caches the result of the previous integrity check. All information is cached in the private data of the in-memory inode objects. The inode private data includes several new fields to cache policies, meta data checksum results, whole file data checksum results, and per page checksum results. While caching the policies, the result of the check for the existence of a policy is also cached. This mechanism serves the purpose of having both a positive and negative cache for the existence of policies, thereby expediting the check for both files that have and those that do not have policies associated with them.

As a per page integrity checking cache, the inode private data contains an integer array with ten elements which acts as a page bitmap. The cache can hold the integrity check results for the first 320 pages when run on an i386 system with page size of 4KB. Thus the results for files which are less than 5MB can be fully cached. This accounts for almost 90% of the files in a normal system [7].

The data and meta data integrity check result caches are invalidated every time there is a data or inode write for that file. Since all information is cached in the inode private data and not in an external list, the reclamation process for the inode cache will take care of the I^{3}FS configuration cache reclamation also. This method of caching is advantageous because the inodes for the frequently-accessed files will be present in the inode cache and hence the policies and results for those files will also be present in the cache.

2.5 Securing I^{3}FS Components

Securing the databases that store the configuration and setup of I^{3}FS is one of the prime requirements for making I^{3}FS a secure file system. There can be valid updates to the checksums needed when a file needs to be genuinely updated and these updates have to follow a secure channel so that there can be a clear differentiation between authorized and unauthorized updates to the files. I^{3}FS uses an authentication mechanism to ensure that updates to the checksums are made by authorized personnel.

I^{3}FS stores the four databases that it uses in an encrypted form. We use the in-built cryptographic API provided by the Berkeley Database Manager for encryption. We use the AES encryption algorithm [14] with a 128-bit key size. Since I^{3}FS requires a key to be provided for reading the encrypted database, we wrote a custom file system mount program that accepts the passphrase from the administrator. Having the database encrypted prevents unauthorized reading of the database file without going through the authentication process.

2.5.1 Authentication

An authentication mechanism is required for I^{3}FS for two reasons. First, mounting and setup of I^{3}FS should be done through a secure channel so that malicious processes that acquire super user privileges could not mount the file system with incorrect configuration options. Second, valid updates to the files that carry policies should be permitted only through a secure channel. This is because critical programs and files need to be updated occasionally and such updates should not require reinitialization of the file system.

Since the I^{3}FS databases are encrypted, reading them requires a passphrase. Therefore we provide a custom mount program that authenticates the person mounting the file system. The first time I^{3}FS is mounted, the administrator is prompted for a passphrase. This passphrase is used to compute the cryptographic hash for a known word, "i3fspassphrase." We store this hash as part of the policy database. During subsequent mounts, the passphrase entered is validated by computing the hash again and comparing it with the stored hash. Upon mismatch of hashes, the mount process is aborted and an error message is returned to the user-level mount program. If the passphrase entered is correct, it is stored in the private data of the super-block structure. Thus the passphrase is kept non-persistent and stored in the kernel memory only.

The checksums for all files that have a policy are computed during the initialization phase of I^{3}FS. To allow valid updates to files whose checksums have already been stored, we provide two modes of operations for I^{3}FS: one that allows updates and another that does not. This is implemented using a flag in the in-memory super block which can be set and unset from user level through an `ioctl`. This `ioctl` can be executed only after providing a valid passphrase. The passphrase passed to the `ioctl` is compared with the one that is stored in the super block private data and access is granted based on the result. A similar authentication method is implemented for the ADD_POLICY and REMOVE_POLICY `ioctl`s as well.

2.6 Actions Upon Failure

There are two kinds of actions that can be specified for files for which integrity checking fails. They are the BLOCK and NO-BLOCK options. The BLOCK option disallows access to files that fail integrity check, and a message is recorded in the log. In the case of NO-BLOCK option, access is allowed for files that fail integrity check but an appropriate message is logged. By default I^{3}FS logs messages through syslog. Optionally, a log file name can be given as a mount option to the custom mount program, and all log messages will be written directly to that file.

3 Implementation

I^{3}FS is implemented as a stackable file system that can be mounted on top of any other file system. Unlike traditional disk-based file systems, I^{3}FS is mounted over a directory, where it stores the files. In this section we discuss the key operations of I^{3}FS and their implementation.

3.1 Initialization and Setup

The first time the administrator mounts I^{3}FS, a passphrase needs to be entered for the file system to initialize itself. The first mount operation will store the HMAC hash of a known word, "i3fspassphrase," hashed using the passphrase entered, into the policy database for authenticating further mounts. After the file system is mounted, the administrator has to run a user level setup tool that takes a policy file as input. The format of the policy file is described in Section 2.2. The user level utility calls the corresponding `ioctl`s to set up the policies to the four databases:

- ADD_POLICY: This `ioctl` takes the passphrase, path name, and policy bits (an integer) as input. It verifies the passphrase, converts the path name to an inode number, and stores the inode number and the policy bits in the policy database. In addition, based on the policy bits, the `ioctl` computes meta data and data checksums appropriately and inserts them into the meta data and data databases.

- REMOVE_POLICY: This `ioctl` takes a passphrase and path name as input. It authenticates and then converts the path name to an inode number and removes all entries from all four databases that match the given inode number.

- ALLOW_UPDATES: This `ioctl` takes the passphrase as argument. It just authenticates and sets the AUTO_UPDATE flag in the in-memory super block which allows updates to files with a policy, along with the update of their checksum values.

- DISALLOW_UPDATES: This `ioctl` resets the update flag in the super block private data, so that further updates to checksums of files with policies are stopped.

Recursive policies can also be specified for directories, so that the policy is applied to all files inside the directory tree. In this case, the user level program uses `nftw(3)` to enumerate the set of files for which the policies should be applied. It then calls the `ioctl` for each of the files.

The usage of the user level program `i3fsconfig` is as follows:

```
i3fsconfig [-u ALLOW|DISALLOW] [-f POLICYFILE]
```

3.2 Mount Options

I^{3}FS is mounted using a custom mount program that uses the `mount(2)` system call. It uses `getpass(3)` to accept a passphrase from the administrator and passes the passphrase as a mount option. A custom mount program is used instead of the Linux mount program because the passphrase entered should not be visible at the user level after the file system is mounted.

There are three optional mount parameters.

- The `auto-update` option sets the AUTO_UPDATE flag in the kernel so that checksums will be updated every time a file with a policy is updated.
- The `logfile` option allows one to specify a separate log file where the I^{3}FS log messages can be written to.
- The `dbdir` option allows the administrator to set the location of the checksum databases. Normally these databases are stored inside the file system, and I^{3}FS prevents direct access to them and hides them from view. With this option, administrators can place the databases in a different directory than the checksummed file system; this is useful, for example, when I^{3}FS is stacked on top of NFS because the databases could be kept on a safer local directory.

3.3 Meta Data Integrity Checking

The flowchart for meta data integrity checking is shown in Figure 2. For checksumming the meta data, since we have a customizable set of inode fields to be checksummed, we need to use both the policy bits and the stored checksums for integrity checking. The meta data integrity checking is done in the file permission check function, `i3fs_permission`. The

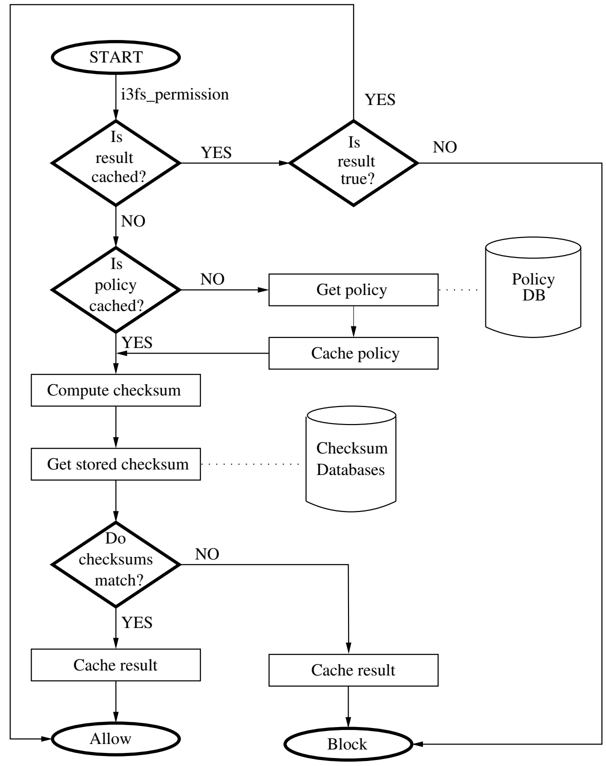

Figure 2: Flowchart for I^{3}FS permission checks

`i3fs_permission` function is called after lookup for every file that is accessed. Hence, the integrity check cannot be bypassed for files with a policy. The permission check function first checks the policy cache to determine if the file's inode has a policy associated with it. Upon a cache miss, it refers to the policy database and then decides the result. If there is a policy associated with the file, its policy bits are retrieved from the policy database. From the policy bits, the set of inode fields that have to be checksummed is found and the checksum for those fields is computed. This computed checksum is verified against the checksum value stored in the meta data checksum database. If both checksums match, then access is granted; if the checksums do not match, then the necessary action is performed as per the action policy bit.

Once it is determined if the inode has a policy associated with it or not, the information is stored in the in-memory inode as a cache for further accesses. As long as that inode is present in the inode cache, the policy information will also be cached.

3.4 Data Integrity Checking

We provide two options for checksumming file data. The first is PER_PAGE checksumming and the second is WHOLE_FILE checksumming. In the case of per page checksumming, integrity checking is done in the page level read function, `i3fs_readpage`. If the option is to checksum whole files, then the integrity checking is done in the open function, `i3fs_open`. In the case of WHOLE_FILE checksumming, whenever `i3fs_open` is called for a file with a policy, the policy bits present in the in-memory inode are checked. If the data checksum mode bit is set to WHOLE_FILE, then the checksum for the whole file is computed and verified against the checksum value stored in the data checksum database. If they match, `i3fs_open` succeeds; if not, the necessary action is performed as per the policy bits. The WHOLE_FILE integrity check results are cached in the inode private data in a field named `whole_file_result`.

In PER_PAGE integrity checking, during `i3fs_readpage`, I^{3}FS checks the policy bits to determine whether page-level checksumming is enabled. If yes, then the checksum is computed for that page and it is compared with the stored value in the data checksum database. The result is cached in the page bitmap present in the inode private data as explained in Section 2.4.

3.5 Frequency of Checks

Since whole file checksumming is a costly operation, we provide an option for specifying the frequency of integrity checks in the policy. For performance reasons, one can set up a policy for a file such that it will be checked for integrity every N times it is opened, where N is an integer value. Every time a file with a policy is opened, we check if it has a frequency number associated with it. If yes, the counter entry for the file in the access database is incremented by one. When the value is equal to N, integrity check is performed and the counter is then reset to zero.

3.6 Updating Policies

Policies that are enforced when the file system is initialized might not remain valid at all times. We provide a method by which the administrator can update the policies dynamically without reinitializing the system. This can be done using the following two `ioctl`s: ADD_POLICY and REMOVE_POLICY. The administrator can either add policies to new files or remove policies from existing files. If a policy for an existing file has to be modified, it has to be first removed and then reinserted using the ADD_POLICY `ioctl`.

3.7 Updating Checksums

Often it is required that files with policies be updated from time to time. For example, administrators need to install or upgrade system binaries. Such updates should also re-compute the checksums that are stored in the databases, so that I^{3}FS need not be reinitialized for every file update. However, these kinds of checksum updates should be allowed through a secure channel so as to prevent malicious programs from triggering checksum updates subsequent to an unauthorized modification to file data. In I^{3}FS, we provide a flag called AUTO_UPDATE which can be set and reset by the administrator after authenticating using the passphrase. This can also be set during mount as a mount option. When the AUTO_UPDATE flag is set, all updates to files with policies will update the checksums associated with them. If the flag is not set, file data updates will be allowed without updates to the checksums so that these are categorized as unauthorized changes. The AUTO_UPDATE flag can only be set from a console for security reasons; processes that are executing in a non-console shell are not allow to update checksums when the AUTO_UPDATE flag is set.

The checksum updates for meta data are done in the `put_inode` file system method of I^{3}FS. Whole file checksums are updated in the file `release` method and per page checksums are updated in the `writepage` and `commit_write` methods, respectively.

3.8 Inheriting Policies

To facilitate automatic policy generation for new files that get created after I^{3}FS is initialized, we provide a method for the policy of the parent directory to be inherited by the files and directories that are created under it. This can be used by setting the INHERIT policy bit for the directory in question. Whenever a file or a directory is created, the policy of the parent directory is copied for it, if its parent directory has the INHERIT bit set. However, for checksums to be updated for the new file, the AUTO_UPDATE flag must be set. If the flag is not set, then the policy of the parent directory will be copied for the new file, but the checksums will not be updated. Thus the next time such a file is accessed there will be a checksum mismatch.

4 Evaluation

We used the stackable templates generated by FiST [28] as our base, and it started with 5,670 lines of code. To implement I^{3}FS, we added 4,227 lines of kernel code and 300 lines of user level code. In addition to this, I^{3}FS includes 367 lines of checksumming code implemented by Aladdin Enterprises [5]. We wrote two user level tools: a custom mount program for I^{3}FS and another tool for setting up, initializing, and configuring I^{3}FS. I^{3}FS is implemented as a kernel module and requires the in-kernel Berkeley database [8] module to be loaded prior to using I^{3}FS.

To measure the performance of I^{3}FS, we stacked I^{3}FS on top of a plain Ext2 file system and compared its performance with native Ext2. All measurements were conducted on a 1.7GHz Pentium 4 with 1GB RAM and a 20GB Western Digital Caviar IDE disk. For the frequency of checks experiment, described in Section 4.3, we lowered the amount of memory to 64MB. The operating system we used was Red Hat Linux 9 running a vanilla 2.4.24 kernel. We unmounted and remounted the file systems before each run of benchmarks so as to ensure cold caches. All benchmarks were run at least ten times and we computed 95% confidence intervals for the mean elapsed, system, user, and wait time using the Student-t distribution. In each case, the half widths of the intervals were less than 5% of the mean. In the graphs in this section, we show the 95% confidence interval as an error bar for the elapsed time. Wait time is the elapsed time less CPU time and user time and consists mostly of I/O, but process scheduling can also affect it.

We calculated the overheads of I^{3}FS under several different configurations and a variety of system workloads. Based on the types of policies, we classified the tests as follows:

- Without any policies (NP)
- Only meta data checksumming (MD)
- Meta data and whole-file data checksumming (MW)
- Meta data and per-page data checksumming (MP)
- Meta data and whole-file checksumming with inheritable policies (MWI)
- Meta data and per-page checksumming with inheritable policies (MPI)

Each of the above configurations of I^{3}FS are used to identify the isolated overheads of the components of I^{3}FS. The NP configuration does not compute checksums, and is useful in finding the overheads due to the stackable layer and to check whether files have policies associated with them or not. The MD configuration is used to find the overhead of checksumming the meta data alone. The other configurations, MW, MP, and MPI, isolate the overheads associated with each of the checksumming options described on Section 2.

We tested I^{3}FS using a CPU-intensive benchmark, an I/O-intensive benchmark, and a custom read benchmark to test the frequency of checks performance.

For a CPU-intensive benchmark, we compiled the Am-utils package [16]. We used Am-utils 6.1b3: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. Then it builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Although the Am-utils compile is CPU intensive, it contains a fair mix of file system operations, which result in the creation of several files and random read and write operations on them. This compile benchmark was done for Ext2, as well as for I^{3}FS for the aforementioned six configurations.

For an I/O-intensive benchmark we used Postmark [23], a popular file system benchmarking tool. Postmark creates a large number of files and continuously performs operations that change the contents of the files to simulate a large mail server workload. We configured Postmark to create 20,000 files (between 512 bytes and 10KB) and perform 200,000 transactions in 200 directories. Postmark was run on Ext2 and I^{3}FS with NP, MDI, MWI, and MPI configurations. The other configurations, MD, MW, and MP, are not relevant for Postmark, as Postmark creates a lot of new files and these configurations only apply to existing files.

Finally, to measure the performance of I^{3}FS with frequency of checks, we wrote a custom program that repeatedly performs read operations on a single file. We conducted this test for I^{3}FS with frequency of checks set to 1, 2, 4 ,8, 16, and 32.

4.1 Am-utils Results

Figure 3 shows the overheads of I^{3}FS under different configurations for an Am-utils compile.

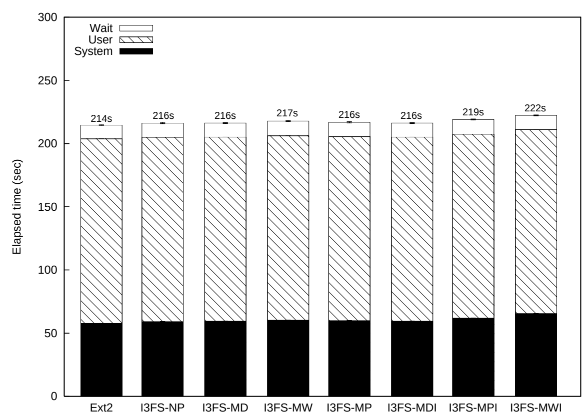

Figure 3: Am-utils results for Ext2 and I^{3}FS

The configuration of I^{3}FS that has the maximum overhead when compared to regular Ext2 is the MWI configuration. It has an overhead of 4% elapsed time and 13% system time. Most of the elapsed time overhead is due to system time increase because of the checksum computation. The MWI configuration calculates data checksums for whole files including the files that are newly created. Therefore, it has the highest overhead of all configurations. The elapsed time overheads of all other configurations are less than 1%. The MPI configuration has a system time overhead of 7% as this configuration computes data checksums for files including newly created ones. The system time overhead of other configurations range from 2% to 3%.

Since an Am-utils compile represents a normal user workload, we conclude that I^{3}FS performs reasonably well under normal conditions.

4.2 Postmark Results

Figure 4 shows the overheads of Ext2 and I^{3}FS for Postmark under the NP, MDI, MPI, and MWI configurations. Since Postmark creates and accesses files on its own, it can only exercise configurations that have inheritable policies.

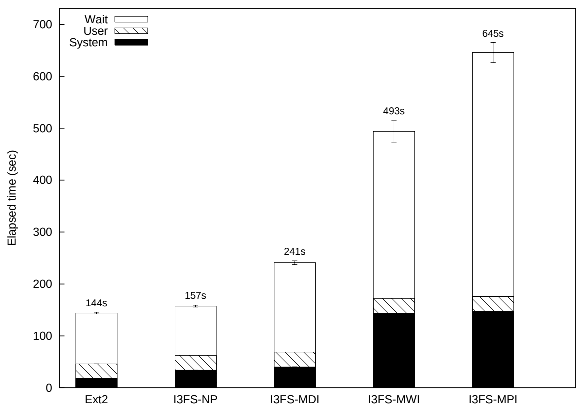

Figure 4: Postmark results for Ext2 and I^{3}FS

Unlike the Am-utils compile, for Postmark we were able to see a wide range of overheads for different configurations of I^{3}FS. The NP configuration had an elapsed time overhead of 9%. The system time overhead was 89%, which is mainly because of the check for the existence of policies. The overhead due to indirection of the stackable layer also adds to this overhead. The MDI configuration had an elapsed time overhead of 67%. This overhead is partly because of checksum computation for the meta data during file creation and accessing. Database operations for storing and retrieving meta data checksums also contribute to the overall overhead. I^{3}FS under the MWI configuration was 3.5 times slower than Ext2. This is because it computes, stores, and retrieves checksums for the data and meta data of all files, including newly created files. Finally, the MPI configuration was 4.5 times slower than Ext2. The MPI configuration checksums the meta data and the individual pages of all the files. The MPI configuration of I^{3}FS is slower than the MWI configuration because we configured Postmark to create files whose sizes range from 512 bytes to 10KB. Thus the maximum number of pages a file can have is three as the page size is 4KB, and computing the checksums for the three pages in one shot is more efficient than checksumming individual pages separately.

Since Postmark creates 20,000 files and performs 200,000 transactions within a short period of just 10 minutes, it generates a rather intensive I/O workload. In normal multi-user systems, such workloads are unlikely. The above benchmark shows a worst case performance of I^{3}FS. Under normal conditions, the overheads of I^{3}FS are reasonably good, as evident from the Am-utils compile results in Section 4.1.

4.3 Frequency of Checks

To measure the performance of I^{3}FS for whole file checksumming with frequency of checks enabled, we wrote a custom user level program that reads the first page of a 64MB file 500 times. We ran this test with 64MB RAM, so as to ensure that cached pages are flushed when the file is read sequentially during checksum computation phase. Since the file is read sequentially, by the time the last page of the 64MB file is read, we can be sure that the first page is flushed out of memory. We calculated the difference in speeds for frequency values of 1, 2, 4, 8, 16, and 32. These numbers reduce the frequency that the checksums are computed logarithmically. Figure 5 shows the results of our custom benchmark for the different values of frequency of checks.

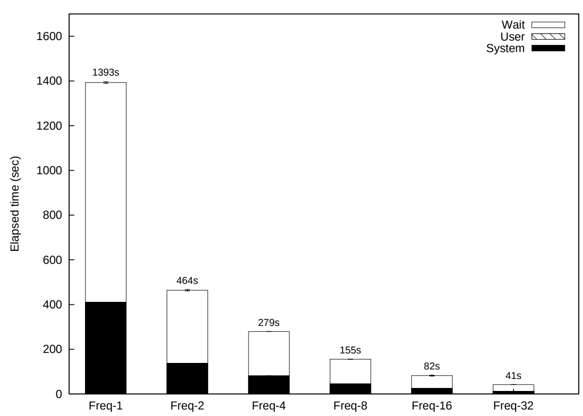

Figure 5: Frequency-of-checks benchmark results for frequency values of 1, 2, 4, 8, 16, and 32.

As evident from the figure, the time taken is reduced logarithmically as the frequency number increases exponentially. We can see that the rate of decrease of the elapsed time and the system time is almost equal. This is because both the I/O for reading the files and the checksum computation itself reduces as the frequency value increases. Without a custom frequency value (Freq-1), the program takes 1,393 seconds to complete, and as the frequency value increases, the time taken reduces to 464 seconds, 279 seconds, and so on.

Therefore, when system administrators are concerned about the system performance while checksumming whole files, they can set an appropriate frequency of check value.

5 Related Work

In recent years, systems researchers have proposed various alternatives to increase the security of computer systems. These solutions can be broadly categorized as user mode and kernel mode. Integrity checking of files is an important aspect of system security. Our work is an in-kernel approach to check integrity and detect intrusions in the file system.

In this section we briefly discuss some previous work that addresses integrity checking and file system security in a broader sense. We discuss this in three categories: user-mode utilities, in-kernel approaches and other approaches that increase the security of the computer systems.

User Tools and Utilities The open-source community has developed various user-mode tools for file system integrity checking. While we follow the semantics of Tripwire [9, 10, 22] for our integrity checker, there are other similar tools. These include Samhain [20], Osiris [15], AIDE [18], and Radmind [3]. Most of these user-mode tools were modeled along the lines of Tripwire. AIDE and Radmind have been developed for UNIX systems with some more functions like threaded daemons and easy system management. In addition, Samhain uses a stealth mode of intrusion detection with remote administration utilities.

In-Kernel Approach *Linux Intrusion Detection System* (LIDS) [12] is a more comprehensive system that modifies the Linux access control semantics, called discretionary access control (DAC), thus offering mandatory access control inside the kernel. In contrast, our work does not change any Linux semantics or any modifications to the kernel. We leverage file system level call interposition using a loadable kernel module for file systems.

A similar approach is used by *Linux Security Modules* (LSM) [24], which present an extensible framework with in-kernel hooks for adding new security mechanisms inside the kernel for file systems, memory management and the network sub-systems. LSM does not use any policies, but provides a foundation to add complete system security.

Other Systems Security Apart from integrity checkers, there are other ways to increase the security of any host. System call interposition [17] uses the indirection of any call to a kernel function. This is a powerful tool for monitoring application behavior as soon as the context switches to kernel mode. Ostia [6] presents a model that delegates certain system critical responsibilities to a sandbox. This helps in localizing the impact of any attack after a pseudo off-line detection process. In contrast, our approach uses interposition of calls made by the virtual file system (VFS) on behalf of a file system.

Another class of solutions uses call-graph analysis to backtrack any intrusions on a host [11]. These techniques aim to determine the vulnerability of the system used by the attacker to break into the system after an attack took place. In contrast, I^{3}FS tries to detect an intrusion or inconsistency in the system as it occurs.

Finally, more recent work uses Virtual Machine Monitors (VMM) to detect any intrusions by placing the IDS in a more secure hardware domain [6]. This approach aims to minimize the impact of an attacker on the intrusion detection system. This approach has been tested for passive attack scenarios and incurs system overhead due to context switches across the interface between the OS and the VMM. In contrast, our approach has less overhead since we use fine-grained indirection.

6 Conclusions

We have described the design, operation, security, and performance of a versatile integrity checking file system. A number of different policy options are provided with various levels of granularity. System administrators can customize I^{3}FS with the appropriate options and policies so as to get the best use of it, keeping in mind performance requirements. As evident from the benchmark results, I^{3}FS has a performance overhead of 4% compared to regular Ext2 under normal user workloads. The encrypted database and cryptographic checksums make I^{3}FS a highly secure and reliable system.

6.1 Future Work

Our group has previously developed secure and versatile file systems like NCryptfs [25, 27], Tracefs [1] and Versionfs [13] and we would like to integrate the features of these file systems together with I^{3}FS so as to provide a highly secure and versatile system.

Currently, I^{3}FS cannot be customized to individual users. We plan to add per user policies and options, so that individual users can set up security options for their own files, without requiring the intervention of the system administrators, but still allow administrators to override global policies.

7 Acknowledgments

We thank the anonymous Usenix LISA reviewers for the valuable feedback they provided, and our shepherd Michael Gilfix. Thanks go to Charles P. Wright for his suggestions; this work was based on integrity protection ideas and experience developed by Charles within the NCryptfs encryption file system. This work was partially made possible by an NSF CAREER award CNS-0133589, NSF Trusted Computing award CCR-0310493, and HP/Intel gift numbers 87128 and 88415.1.

The open-source software described in this paper is available in `www.fsl.cs.sunysb.edu/project-i3fs.html`.

References

[1] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

[2] CERT Coordination Center. CERT/CC Overview incident and Vulnerability Trends Technical Report. `www.cert.org/present/cert-overview-trends`.

[3] W. Craig and P. M. McNeal. Radmind: The Integration of Filesystem Integrity Checking with File System Management. In *Proceedings of the 17th USENIX Large Installation System Administration Conference (LISA 2003)*, October 2003.

[4] P. A. DesAutels. SHA1: Secure Hash Algorithm. `www.w3.org/PICS/DSig/SHA1_1_0.html`, 1997.

[5] P. Deutsch. Independent implementation of MD5 (RFC 1321). `www.opensource.apple.com/darwinsource/10.2.3/cups-30/cups`.

[6] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium*, February 2003.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, NY, October 2003.

[8] A. Kashyap, J. Dave, M. Zubair, C. P. Wright, and E. Zadok. Using Berkeley Database in the Linux kernel. `www.fsl.cs.sunysb.edu/project-kbdb.html`, 2004.

[9] G. Kim and E. Spafford. Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. In *Proceedings of the Usenix System Administration, Networking and Security (SANS III)*, 1994.

[10] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer Commuications and Society (CCS)*, November 1994.

[11] S. King and P. Chen. Backtracking Intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003.

[12] LIDS Project. Linux intrusion detection system. `www.lids.org`.

[13] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004.

[14] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the Development of the Advanced Encryption Standard (AES). Technical report, Department of Commerce: National Institute of Standards and Technology, October 2000.

[15] Osiris. Osiris: Host Integrity Management Tool. `www.osiris.com`.

[16] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. `www.am-utils.org`.

[17] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th Annual USENIX Security Symposium*, August 2003.

[18] J. Reed. File Integrity Checking with AIDE, 2003. `www.ifokr.org/bri/presentations/aide_gslug-2003/`.

[19] R. L. Rivest. RFC 1321: The MD5 Message-Digest Algorithm. In *Internet Activities Board*. Internet Activities Board, April 1992.

[20] Samhain Labs. Samhain: File System Integrity Checker. `http://samhain.sourceforge.net`.

[21] M. Seltzer and O. Yigit. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference*, pages 173–84, Dallas, TX, January 1991. `www.sleepycat.com`.

[22] Tripwire Inc. Tripwire Software. `www.tripwire.com`.

[23] VERITAS Software. VERITAS File Server Edition Performance Brief: A PostMark 1.11 Benchmark Comparison. Technical report, Veritas Software Corporation, June 1999. `http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf`.

[24] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Pro-*

ceedings of the 11th USENIX Security Symposium, San Francisco, CA, August 2002.

[25] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003.

[26] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.

[27] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998. `www.cs.columbia.edu/~library`.

[28] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.