applied sciences

MDPI

Article

XFilter: An Extension of the Integrity Measurement Architecture Based on Fine-Grained Policies

Alan Litchfield [1,*] and Weihua Du [2]

[1] Service and Cloud Computing Research Lab, Auckland University of Technology, Auckland 1010, New Zealand
[2] Datacom, Wellington 6011, New Zealand
* Correspondence: alan.litchfield@aut.ac.nz

Abstract: The Integrity Measurement Architecture subsystem on the Linux platform is a critical security component in the kernel to ensure the integrity of the running system. However, the default Integrity Measurement Architecture policy mechanisms based on options such as file owner and FSMAGIC cannot achieve a file-level configuration. Although Integrity Measurement Architecture supports the Linux Security Module policy rules to be close to the goal of fine-grained configuration, it is not easy to be managed because the Linux Security Module was not originally designed for integrity measurement. Moreover, the Linux Security Module-based policy does not apply in some use cases considering the type of Mandatory Access Control tools chosen by users. This paper presents a new policy configuration option, named XFilter, that achieves a fine-grained policy configuration method. The XFilter includes two policy matching mechanisms, XLabel and XList, which share the same policy token created for XFilter exclusively. XLabel marks the files for measurement using a label in the file's extended attribute (xattr). By contrast, XList stores the measurement information in a list of file paths. To simplify the deployment, an automatic configuration process is implemented for integrating into the package management system. The evaluation results suggest that both mechanisms satisfy the requirements of file-level IMA policy control and create a performance burden for system operation in the acceptable range. They also reveal a positive correlation between the increment of the system latency and the growth of the length of file paths list for the XList mechanism.

Keywords: X-filter; integrity measurement architecture; linux security module; mandatory access control; XLabel; XList

Citation: Litchfield, A.; Du, W. XFilter: An Extension of the Integrity Measurement Architecture Based on Fine-Grained Policies. *Appl. Sci.* **2023**, *13*, 6046. https://doi.org/10.3390/app13106046

Academic Editors: Leandros Maglaras, Helge Janicke and Mohamed Amine Ferrag

Received: 24 November 2022
Revised: 1 May 2023
Accepted: 1 May 2023
Published: 15 May 2023

1. Introduction

Critical to protecting a system from malicious software and remote exploitation is ensuring the integrity of the operating system. To meet threats to system integrity, we consider the vulnerability of system security components where hardware with cryptographic processing capabilities are deployed increasingly. To meet this trend, the Trusted Computing Group (TCG) provides the Trusted Platform Module (TPM) as an open industry standard for hardware-based trust mechanism [1].

The technology delivered by the TPM specification is an enabler of remote attestation and is integrated into many network authorisation solutions. Remote attestation is a technique for verifying the status of a remote client through a trust mechanism over a network [2,3]. On the Linux platform, the Integrity Measurement Architecture (IMA) measurement module was developed in the kernel to work with the TPM devices, thereby achieving the system integrity verification. As a subsystem in the Linux kernel, IMA proves to both remote and local systems that the files in a running system are not altered intentionally or accidentally [4]. As the focus of this study, the IMA policy mechanism identifies specific files that should be assessed by the measurement functions.

By default, the IMA subsystem in the Linux kernel carries a built-in policy to determine what files are to be measured. IMA covers a large range of file operations in the system,

such as file execution, library mmap, and file opening by the root user for reading [4]. The default policy settings rely on factors such as FSMAGIC and file owners, which are overly coarse-grained and may not be suitable for every use case. Usually, according to the threat models and application scenarios, only a few sensitive files need to be measured in many remote attestation solutions. The strategy of measuring a large number of files creates a nondeterminism problem [5], such that even though all file measurement operations are acceptable, the system status may not always be maintained or consistent.

The IMA policy supports the policy system of some Linux Security Modules (LSM), such as SELinux and Simple Mandatory Access Control Kernel (SMACK), as a mechanism for refining measurement levels. However, this method is impractical because SELinux and SMACK are Mandatory Access Control (MAC) tools used within the system and not available in all use cases. Linux distributions such as SUSE Linux Enterprise are shipped with AppArmor as the default MAC application, whose support on SELinux is at a fairly early stage [6]. SMACK may also be absent from the support list. Additionally, due to their lower impact on the file loading overhead, many system installations choose AppArmor or TOMOYO instead of SELinux [7,8]. Additionally, SELinux implements MAC policies by means of contexts composed of users, roles, and types [9]. It means SELinux cannot apply a policy to individual files and is not sufficiently fine-grained. However, increasing the level of granularity will increase complexity and make security maintenance more difficult. SMACK supports a labelling mechanism for fine-grained control, but it requires that a specific pseudo-filesystem be mounted for SMACK use only [10], thereby increasing complexity and maintenance costs.

Therefore, the implementation of an LSM-independent fine-grained policy, especially at the file level, can bring multiple benefits. Such an approach can help lower Central Processing Unit (CPU) time spent on hash processing. Additionally, limiting the number of binaries to be measured can reduce the size of the Measurement List (ML), which is critical for constrained devices [11]. Moreover, this approach addresses the nondeterminism problem, enabling the verification of the integrity of critical executables without the need to obtain an ML. Thus, by avoiding the transmission and evaluation of each ML [12], the computational cost of record processing is lowered. Furthermore, independence from the other LSM subsystems improves flexibility and efficiency for system maintenance because more relations between IMA and MAC tools in the security service deployment are possible.

To achieve a fine-grained level of measurement control without relying on LSM objects, this paper presents the implementation of a built-in filtering system called XFilter. The system provides two mechanisms, XLabel and XList, to implement file-level measurement control. XLabel marks the files covered by the IMA measurement as labels in its extended attribute (`xattr`). XList records the file paths in a list loaded by the kernel during IMA initialisation.

2. Prior Research

IMA is built on Trusted Computing Group (TCG) standards and TPM hardware [12,13]. Similar solutions based on TCG/TPM have been proposed, such as the Next Generation Secure Computing Base (NGSCB) framework [14]. NGSCB provides the measurement function by dividing a disk into trusted and untrusted partitions, each running independent operating systems. However, due to a lack of flexibility and the incomplete coverage of measurement, NGSCB is limited in its use. Another trust computing architecture that implements integrity measurement in partition blocks on a hard disk is Terra [15]. However, Terra requires a large amount of measurement data to be stored and the consequential difficulty in processing the stored data. By implementing measurement functions at the kernel level, IMA overcomes these issues with a simpler approach.

However, identifying IMA function security vulnerabilities reveals a potential attack surface where a block device with malicious firmware can subvert the protection mechanism [16]. By manipulating a disk cache at the firmware level, an attacker can deceive the

IMA such that a measured file is incorrectly reported as not modified when it has been. The exploitation surface is limited to the firmware and so the means of exploiting a vulnerability may be limited to vendor channels or the supply chain.

To provide a high-integrity Android operating system for mobile phones, a small set of SELinux policies and the PRIMA mechanism are combined [17,18]. Depending on the IMA framework and the application of the class-level attestation, the measurement is implemented in both low-level system files and the binaries on top of the Android's Dalvik Virtual Machine (VM). This design may satisfy the requirement of fine-grained attestation. However, the absence of a root-of-trust device such as TPM may limit how the technique is applied in production.

It has been claimed that a weakness is exposed in IMA if a large number of files need to be checked, resulting in the ML consuming too much storage space [19]. However, the assertion about storing ML in TPM does not recognise that it is only the aggregated value of ML that is stored in TPM, which takes very little space. However, the issue may exist for storage- and processing-constrained devices that may store ML, such as those on the Internet of Things (IoT).

The IMA framework is often deployed as a stage in remote attestation. For example, Trusted Platform Service (TPS) is a policy-driven architecture that is a part of the general authorisation mechanism. TPS utilises IMA-based remote attestation to provide verification for system-wide integrity by combining IMA, TPM, Trusted Boot, and TPS [20]. To meet integrity requirements for VM environments such as Virtual Network Functions (VNF) and the underlying hypervisor, an IMA-based remote attestation solution may prove that the environment is trustworthy [21], but while a demonstration shows this may work, the approach has yet to be deployed in common hypervisors such as Xen and KVM. Similarly, a concern related to malware infiltration in a multi-tenanted cloud environment that puts the whole infrastructure at risk may be addressed through the application of a Tenant-Attested Trusted Cloud Service (Ta-TCS) [22]. Ta-TCS applies the IMA approach to address the needs of the trust mechanism for service consumers and providers in a way that provides minimal performance loss.

Targeting performance, the scalability of remote attestation that relies on the IMA framework is influenced by time cost, recognised file coverage for executable code-diversity and configuration files, and unrecognised files, because they cannot be addressed by the file measurement database found on the remote party [23]. The influence of code diversity can be managed effectively, but system processes such as bootstrapping provide a significant time cost impact. Additionally, while bootstrapping, configuration files may not be recognised.

In addition to the security concerns expressed in conventional hypervisor-based virtualisation, container-based virtualisation techniques also provide reason for concern. For example, the potential for data leakage from containers and between tenants needs to be controlled. A possibility for controlling this is to apply the IMA technique to the field of the container [24]. For example, applying a container IMA and container-PCR (cPCR) to Docker adapts the mean of the ML partition to prevent data leakage from a container.

Extending the IMA approach to other architectures, network security consistently provides areas of concern and the IMA approach offers opportunities to strengthen trust, for example, as a way to collect proofs in the Network Endpoint Assessment (NEA) protocol supported by strongSwan VPN and strengthen the overall trust mechanism of the host machine [25]. The IMA technique is also involved in security strategies of embedded systems deployed in the IoT networks. As an example, we can mention an architecture that applies IMA as its critical subsystem to assure the security of device authentication in the smart grid [26], where authentication of trusted programs is guaranteed via Public Key Infrastructure (PKI). In manufacturing and power generation plants, the Supervisory Control and Data Acquisition (SCADA) system includes constrained devices that may see negative performance impacts from a verification mechanism. Application of IMA-based remote attestation provides trust with less performance impact [11,27], but IMA

measurement lists are large and storage in constrained prover devices can be an issue. The potential solution is the incremental update approach in measurement maintenance.

Over time, any security tool will demonstrate weaknesses and the likely issues that affect the feasibility of the IMA application are performance impact in the system where it is deployed and nondeterminism. Any definition of nondeterminism is related to its context. For instance, in the modern computer system, nondeterminism may refer to the phenomenon that the running result of a program is affected by external physical influences, such as energy fluctuations [28], or from the uncertainty of input generated by external programs. It is this latter scenario where IMA measurement is related such that in a complex operating system environment, the system files to be measured in one round of operation may not be measured in another round. This can lead to inconsistency in the aggregated value produced by TPM between each run of the system. Thus, it is infeasible to check integrity by simply comparing aggregated values between the prover and remote verifier. In addition, measurement list transmission and processing introduces computation and network overhead and reaching storage limits [19]. Furthermore, depending on the hash method used, the extra processing of hash functions may result in a decline in system performance after IMA is enabled [12].

Performance and nondeterminism can be addressed through the application of batch extend and core measurement, where in batch extend, hashes of multiple files are aggregated into a single hash entry before they are pushed into PCR-10. The core measurement method evaluates the integrity of critical system files in a task [5]. Both methods reduce computational cost on cryptographic processing but the batch extend function is limited to the system booting process, whereas core measurement is limited to the evaluation of the contents of `/etc`, `/bin`, `/usr/bin`, `/sbin`, `/usr/sbin`, and `/lib/modules` and lacks sufficient granularity. However, there ought to be a limit to the number of systems files to be measured, and this can be achieved by enhancing existing policy functions through adaptation of the rules set.

In the Unix universe, most application issues can be divided into two categories, the mechanism and policy. Mechanism refers to the capabilities delivered by a program, whereas policy is defined as the application of capabilities [29]. Policy is usually developed as a separate part of a program. An efficient or fine-grained policy offers improved operating system performance, for example the Control Flow Integrity (CFI) process in the MINIX and FreeBSD kernels [30] and the Linux kernel [31]. However, the latter study found that Linux kernel design has resulted in existing CFI policy functions that are excessively coarse-grained.

3. Method

The study applied the Design Science Research Methodology (DSRM) [32] to produce a practicable implementation as an artefact that may be a model, method, build, or instantiation. The framework provided by DSRM allows for the inclusion of relevant methods to define the problem, design a solution, and to construct an artefact. Thus, in this instance the artefact is a method that extends existing IMA functions to achieve a built-in fine-grained measurement policy. The DSRM process in this research is illustrated as Figure 1.

To provide an interference-free running platform, the `libvirt` virtual environment [33] was deployed, which relies on the Qemu emulator and Kernel Virtualization Module (KVM) hypervisor as the backend. A guest VM was configured with two CPU cores, 1024 MB memory, and other specifications matching the host machine. In addition, TPM emulation was enabled to simulate a complete IMA measurement function in the guest system.

The openSUSE Leap 15.2 [34] (July 2) Linux distribution, which includes core packages and security patches with SUSE Linux Enterprise (SLE15) [35], was selected as the operating system for development. The robustness [36] of regular-release distribution and the using of enterprise-targeting packages provides a stable environment for kernel development. Regarding kernel version, upstream Long-Term Support (LTS) release 5.4 [37] was used in

this research because it reflects the standard functions for general IMA features and its ease of migration to other platforms.

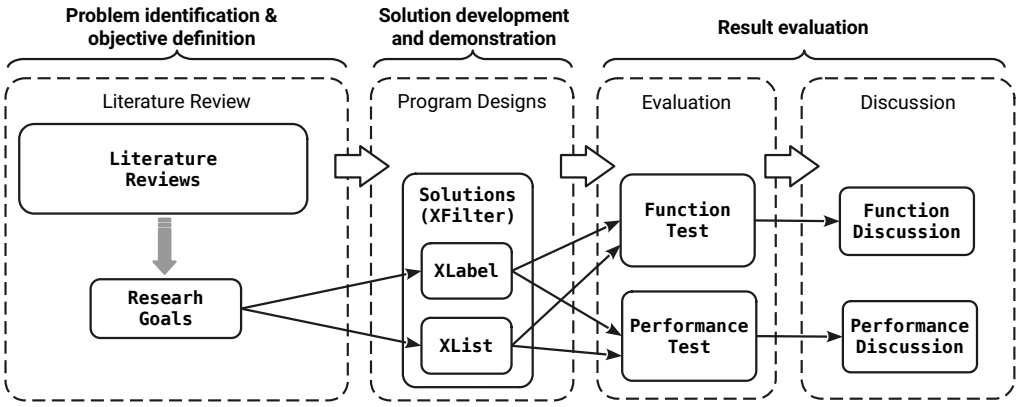


Figure 1. DSRM processing.

The code in this development was built with kernel-provided configurations, such as `makefile` and Kbuild [38], and the GNU Compiler Collection (GCC) (Version 7.5.0). The requirement of default settings in `Kconfig` means that the IMA subsystem is not built as a kernel module. Debugging was conducted using the kernel built-in `ftrace` function with `trace-cmd` as a front-end command-line tool. In addition, reliance was placed upon the ring buffer generated by the Linux kernel, which is called using the `dmesg` command.

The evaluation has two phases—the function test and performance evaluation. The function test targets workability and functionality and verifies the basic working process of new XFilter-based IMA policies. The performance evaluation collects performance data that include execution latency and memory consumption in the kernel space for statistical analysis. Evaluation data collection and processing is handled with `bash` shell scripts, which call multiple external tools. Rather than commonly used benchmarking applications, low-level tracers, `strace` and `ftrace`, are applied in the performance evaluation. These have been applied for I/O latency evaluation of the function call interfaces in specific file operations [39]. Apache2 (Version 2.4.43) was chosen as the target application because it is widely deployed [40] and involves a large number of dynamic libraries or configuration files.

The measurement target is introduced to simulate the workload in the evaluation. GNU `tar` (Version 1.30) is a suitable candidate as a target for the `time` command to capture observable latency values based on the compression operation, whereas the `ls` command (GNU `coreutils` version 8.29) is sufficient for `strace` for system call measurement from the userspace. Regarding the measurement of kernel function using `ftrace`, a minimal program developed specifically is involved as the target for controlling overall complicities.

Since the IMA functions are implemented in the kernel space, kernel memory allocation needs to be examined. In this project, overall memory consumption in the kernel space, reflected by `slabs` allocation counts, was captured inside the running kernel via `/proc/meminfo` [41].

For the data analysis, applications of IMA policy scenarios for the built-in policy in XLabel and XList were compared. Multiple measurement runs were conducted to provide a sufficient sample size, to derive a representative value or central tendency, and to balance efficiency against sample space size. The data collected present distributions of variables that range from extremely large to small. In this case, the basic mean is not suitable as a measurement approach. Outliers may affect the accuracy of the measurement result [42]. Therefore, a trimmed mean, in which a percentage of extreme variables are discarded as outliers [43], was applied in some scenarios as the method of central tendency measurement where the application of a trimmed mean was determined according to the distribution pattern.

4. XFilter Design

To achieve a fine-grained policy without the involvement of LSM, XFilter provides two approaches, XLabel and XList. XLabel labels a single file to be measured using the `xattr` attribute, where files with a specific attribute are measured when they are opened or executed. XList relies on a configuration file that contains a list of file paths, which are to be measured during file operations.

4.1. Features Review

This development introduces `xfltr` as a new policy token that provides user-configurable options for `XF_XLABEL` and `XF_XLIST` to compose policy rules. When the `xfltr` token appears in a rule, the rule is applied to files that meet the token requirements only. For example, the rule below reads as "measure the files running the `mmap` operation only if they are executable and labeled for XFilter".

```
measure func=MMAP_CHECK mask=MAY_EXEC xfltr=XF_XLABEL
```

The label name for XLabel in `xattr` is `security.xlabel`, which is written into IMA policy functions. `security.xlabel` is provided when the `setfattr` command is applied and while `security.xlabel` exists, and then a file shall be labeled by XFilter.

However, a potential vulnerability emerges when using `xattr` for labelling. While it is assumed that system administrators will not act maliciously, an attacker may exploit XLabel if they already have root permissions and may change attributes without needing to execute the file itself. The XList policy filtering method is not affected because the list with files to be measured is initialised during system bootstrapping and cannot be modified, even by root.

Regarding the `XF_XLIST` option used by XList, the interpretation of the policy rule is similar to XLabel. For example, the following rule is read by the IMA as "if files running the `bprm` check operation are executable and their paths are in the XList configuration file, then measure them".

```
measure func=BPRM_CHECK mask=MAY_EXEC xfltr=XF_XLIST
```

The XList configuration file is in `/etc/sysconfig/ima-xlist` and includes a list of file paths to be appended to the token `file=`. The configuration file can be edited by the user with a text editor. The following extract illustrates the file data structure.

```
...
file=/usr/bin/yes
file=/usr/bin/date
file=/usr/bin/zcat
...
```

During system bootstrapping, the XList configuration file list is parsed and written to a pseudo-file in `/sys/kernel/security/integrity/ima/xlist`. The extract below illustrates the file data structure.

```
...
/usr/bin/yes
/usr/bin/date
/usr/bin/zcat
...
```

The token `xfltr` can be integrated into any rule that belongs to the `measure` action and acts as a filter to reduce existing policy rules. Both XLabel and XList target the policy matching for a single file, allowing a user to limit the files to be measured at a fine-grained level.

4.2. The Implementation of XLabel

In order to measure labelled files, IMA functions need to be capable of both parsing policy rules that contain XLabel options and obtain the `xattr` value (Figure 2). The conditions determine if the target file should be measured, and if so, the measurement is processed by calling the related function.

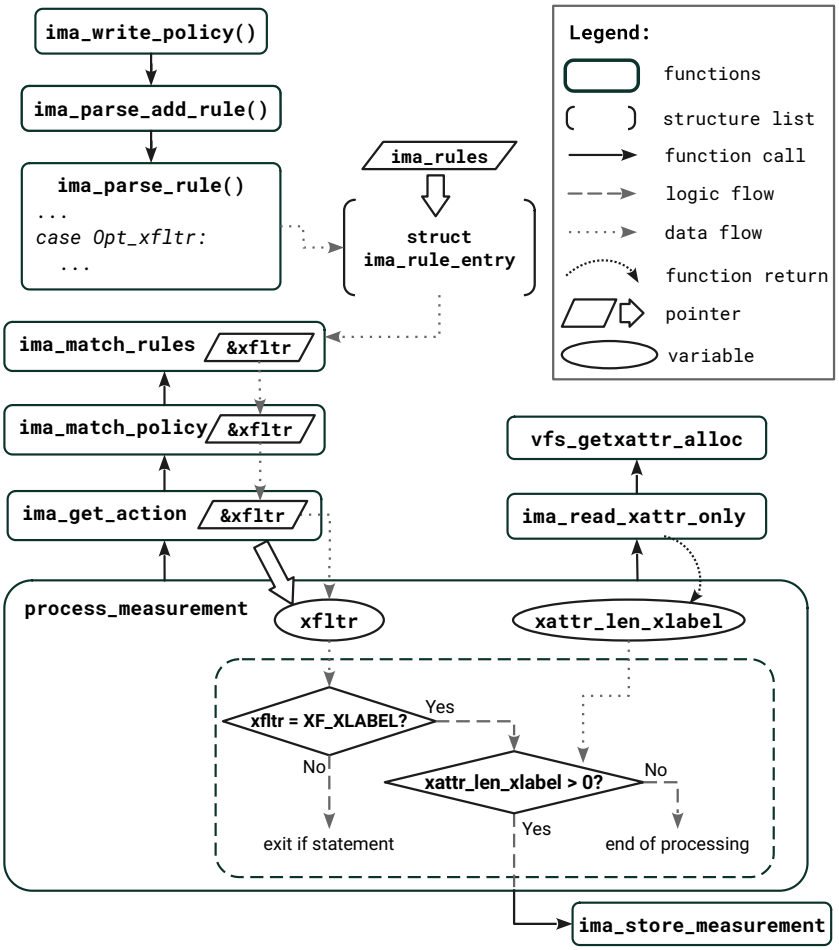


Figure 2. The general processing of XLabel.

The parsing process occurs during the initialisation phase when a new condition `Opt_xfltr` in `ima_parse_rule` is added to parse options related to XFilter. The result is stored as an aggregated `bitmask` in the entry of the structure list pointed to by `ima_rules`.

After initialisation, IMA examines whether XFilter is enabled and which XFilter methods should be applied in the policy. The XFilter method selects an alternative to XFilter from either XLabel or XList and relies on an enumerate type `xfltr` that is passed between functions, and the variable is matched with the structure list pointed to by `ima_rules`.

Subsequently, the XFilter method is kept in the variable `xfltr`, making it immediately available to `process_measurement`. Another input condition is the extended attribute `security.xlabel` of the target file and is obtained by retrieving the `xattr` value from the file's `dentry` structure. This process is performed by a `newl` function, `ima_read_xattr_only`. The length of the value returned by `newl` verifies the existence of `security.xlabel`. The completion of the measurement on the target file is determined with `security.xlabel` and the variable `xflter`.

4.3. The Implementation of XList

The XList filtering method relies on two conditions to establish the order of file measurement (Figure 3). Similar to XLabel, the first condition is met when XList options are

found by checking IMA policy rules. The policy parsing and processing for XList utilises the same process previously described for XLabel (Section 4.2). The second condition establishes whether a list of file paths already exists as the stored pseudo-file `/sys/kernel/security/integrity/ima/xlist`. If so, the file is examined and then used to decide if measurement is needed. As the file path matching interface, the introduction of the pseudo-file in `securityfs` provides security for integrity measurement. Once the XList pseudo-file is created, the list of file paths cannot be modified through the interface, and not even by root.

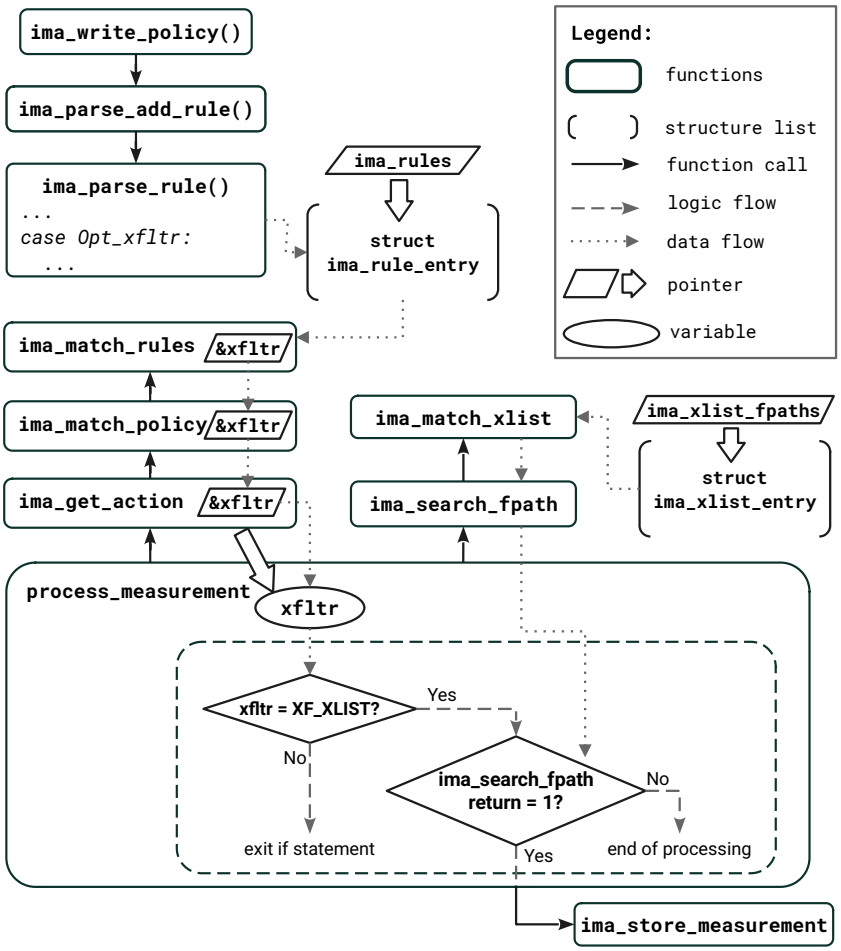


Figure 3. The general processing of XList.

As described earlier (Section 4.1), the policy is initialised during system bootstrapping in accordance with the XList configuration file in `/etc/sysconfig/ima-xlist`. Parsing `ima-xlist` takes the previously described process for handling the IMA policy file as a reference but due to the involvement of the pseudo-file operation, the steps illustrated in Figures 3 and 4 are more complicated than for XLabel (Figure 2).

For a new structure in `file_operations`, a prerequisite is defined, `ima_xlist_ops` (Figure 4). To call file operation functions, five function pointers are involved: `.open`, `.write`, `.read`, `.release`, and `.llseek`. Subsequently, the address of the structure is passed to other functions that create the pseudo-file in `securityfs`.

The new functions `ima_write_xlist`, `ima_release_xlist`, and `ima_open_xlist` are created as file operations. During system startup, a specially developed Dracut script runs through a process in which `ima_write_xlist` prepares the XList pseudo-file, retrieving input through the `write` operation in the `initiramfs` process. The goal of the function `ima_release_xlist` is to handle additional tasks after the XList pseudo-file is created, for example `dmesg` message printing. Additionally, to protect the file from further modification,

the process sets bit S_IWUSR for the i_mode object in the inode structure. The function ima_open_xlist defines user operations targeting the pseudo-file such as file reading. The operations rely on ima_xlist_seqops, which is a seq_operations-type structure that is subsequently passed to the seq_open function.

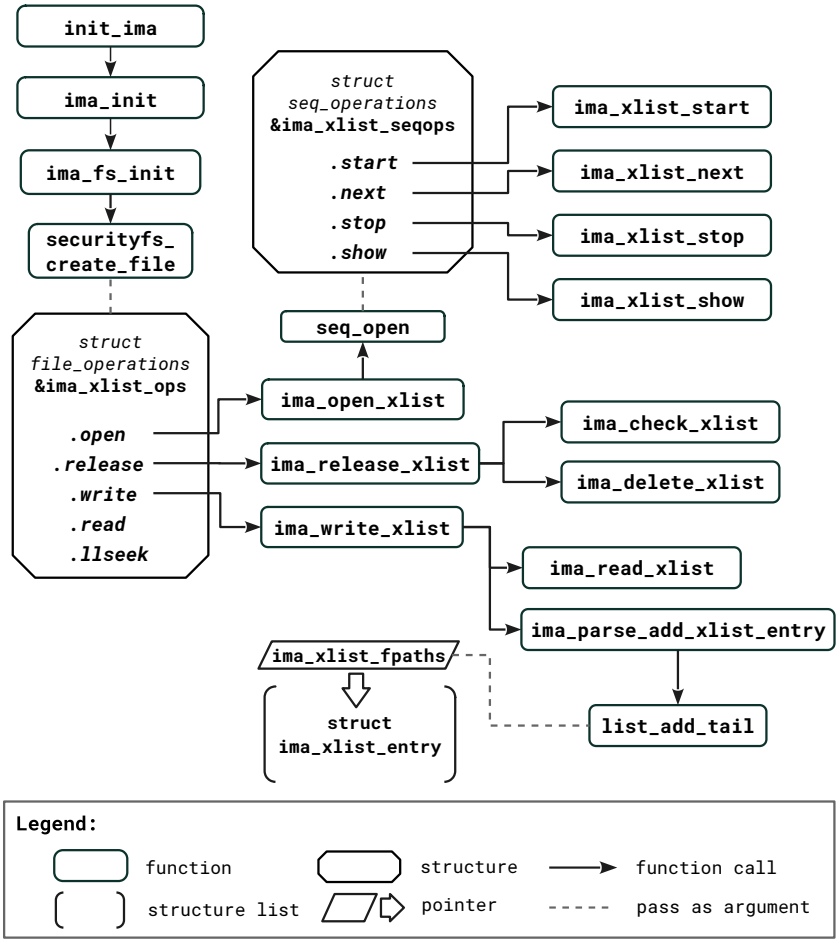


Figure 4. The initialisation of XList pseudo-file.

Finally, to handle the file path matching operation, the function ima_match_xlist is defined. The value returned is passed to process_measurement, where the measurement decision is determined according to the result of file matching and policy parsing.

4.4. Definition of Kconfig Options

The Kconfig options provide flexibility when system building by assisting kernel package maintainers and system administrators to select functions as they need. Three Kconfig options support XFilter function configuration during kernel compilation: IMA_POLICY_XFLTR, IMA_POLICY_XFLTR_XLABEL, and IMA_POLICY_XFLTR_XLIST.

All options are of the type bool and are disabled by default. Switching of the XFilter functions is controlled by IMA_POLICY_XFITR. The options IMA_POLICY_XFLTR_XLABEL and IMA_POLICY_XFLTR_XLIST depend on IMA_POLICY_XFITR and are responsible for enabling and disabling the XLabel and XList functions, respectively.

4.5. Package Management Integration

Putting system administration into practice, XLabel or XList maintenance tasks can be simplified by introducing package management integration. Software package operations provide automatic configuration of the file xattr for XLabel or file paths list for XList.

In this research, two bash shell helper programs, `xlabelling.sh` and `xlist_update.sh`, are provided. To demonstrate a simplified distribution of the solution, a customised Apache2 RPM package was built using Rpmbuild (Version 4.14.1) [44]. The inclusion of the helper programs facilitates IMA system management tasks. In addition, the potential for security risks from user misconfiguration is reduced by transferring the responsibility of measurement file identification from the user to the software maintainer. For each file listed in `flist`, `xlabelling.sh` sets attributes in `security.xlabel`, while `xlist_update.sh` reads the file paths from `flist` and adds them to the XList configuration file in `/etc/sysconfig/ima-xlist`.

5. Evaluation

Evaluation of the XFilter solution assesses effectiveness and efficiency by focussing on the functionality (Section 5.1) and performance (Sections 5.2–5.3) of the solution implementation.

5.1. Functional Evaluation

The functional evaluation for the XFilter solution applies black-box testing, in which the evaluation results are determined through observation and analysis of input and output data [45]. A bash shell test script that automates the assessment is provided, wherein bash was selected for its flexibility when calling external commands and tools. The scripts, `xlabelling.sh` and `xlist_update.sh`, provide options to specify the XFliter methods to be evaluated and the path of the RPM package.

The test script extracts the file `flist` that contains the path of all target files from the Apache2 package (Figure 5). When the Apache2 program is launched, the script checks if the files that appear in the runtime ML are to be measured. For XLabel, the script examines if the extend attribute `security.xlabel` exists for files listed in ML. In the case of XList, the script verifies that both the file paths in the `flist` file and in runtime ML are included in `/etc/sysconfig/ima-xlist`. If successful, the script prints the evaluation results, and otherwise if the evaluation fails then the output message prints the files not matched.

To confirm files to be measured, a verification routine is implemented from the software package to the XList configuration file, and then to the runtime ML. For both XFilter methods, the script also verifies that those files that should not be measured do not appear in the runtime ML.

Prior to each measurement method running, the IMA policy file is re-configured as listed in Listings 1 and 2 below. The tests are then performed during system environment bootstrapping. Due to the volume of output, only relevant lines are shown in Listings 3 and 4 and demonstrate that the functional tests for both the XLabel and XList methods are successful.

Listing 1. XLabel test configuration

```
measure func=MMAP_CHECK mask=MAY_EXEC xfltr=XF_XLABEL
measure func=BPRM_CHECK mask=MAY_EXEC xfltr=XF_XLABEL
measure func=FILE_CHECK mask=^MAY_READ xfltr=XF_XLABEL euid=0
measure func=FILE_CHECK mask=^MAY_READ xfltr=XF_XLABEL uid=0
```

Listing 2. XList test configuration

```
measure func=MMAP_CHECK mask=MAY_EXEC xfltr=XF_XLIST
measure func=BPRM_CHECK mask=MAY_EXEC xfltr=XF_XLIST
measure func=FILE_CHECK mask=^MAY_READ xfltr=XF_XLIST euid=0
measure func=FILE_CHECK mask=^MAY_READ xfltr=XF_XLIST uid=0
```

Listing 3. XLabel test output

```
# ./ima_xfilter_test.sh xlabel apache2-2.4.43-lp152.2.12.1.x86_64.rpm

Extracting xlist from the package ... done!
Extracted xlist file: /tmp/imatest/etc/ima_xfilter/apache2/xlist
Start testing ...
Launching apache2 ... done!
Examining the 'security.xlabel' xattr for the files have been
measured ...
Found xattr in /usr/sbin/start_apache2
Found xattr in /usr/share/apache2/script-helpers
... (omitted lines)
Checking the files not labeled ...
... (omitted lines)
Test Result: **Success**
```

Listing 4. XList test output

```
# ./ima_xfilter_test.sh xlist apache2-2.4.43-lp152.2.12.1.x86_64.rpm

Extracting xlist from the package ... done!
Extracted xlist file: /tmp/imatest/etc/ima_xfilter/apache2/xlist
Start testing ...
Lanunching apache2 ... done!
Testing extracted list against /etc/sysconfig/ima-xlist ...
Found: file=/usr/sbin/start_apache2
Found: file=/usr/share/apache2/script-helpers
... (omitted lines)
Testing runtime list against /etc/sysconfig/ima-xlist ...
Found: /usr/sbin/start_apache2
Found: /usr/share/apache2/script-helpers
... (omitted lines)
Checking the files not in xlist ...
Test Result: **Success**
```

Figure 5. The workflow of functional evaluation for XFilter.

5.2. Performance Evaluation for System Latency

Stress-testing a system during latency performance evaluation plays a critical role in nonfunctional software evaluation. A well-designed application should allow the user to use software functions without experiencing obvious delays [46].

In this research, any increase of latency from the inclusion of the IMA XFilter feature is assessed and recorded in an evaluation matrix (Table 1). The matrix includes the three layers of the user-space, syscall, and IMA kernel functions. The tools used for system latency data collection are `time`, `strace`, and `ftrace`. `strace` obtains latency data by measuring the syscalls `execve` and `mmap`, whereas `ftrace` evaluates the IMA hooks `ima_file_check`, `ima_bprm_check`, and `ima_file_mmap`. The three layers are closely linked such that the user-space program that is evaluated with `time` involves syscalls via `execve` and `mmap`, which are in turn evaluated with `strace`. During the processing of these syscalls, the IMA hook functions are called by `ftrace`.

Table 1. Performance evaluation matrix for system latency.

Layers	Tools	Functions	Scenarios				
			TCB	XLabel	XList (50)	XList (550)	XList (1050)
User-space	`time`	-					
Syscall	`strace`	`execve`					
		`mmap`					
Kernel functions	`ftrace`	`ima_file_check`					
		`ima_bprm_check`					
		`ima_file_mmap`					

In addition to the evaluation layers, five policy rule sets are applied, including IMA TCB, XLabel, and XList with 50, 550 and 1050 file paths. For XList, the processing time spent on file path traversal depends on the length of the file list, and therefore different file

path numbers are included. A typical number for a single application such as Apache2 is 50, while 550 and 1050 were selected to provide intervals of 500.

The MAC components, SELinux and SMACK, contribute considerable overhead to system operations. For example, the impact of SELinux on the overall performance may be as high as 7% [47]. Therefore, a comparison between the policies for LSM and XFilter is likely to be inaccurate and is not covered in this research. To ensure that the results accurately reflect any latency introduced by the new policy mechanism and given that the evaluation targets IMA policy processing, the IMA measurement actions for the XLabel and XList scenarios are temporarily disabled during testing. That is actioned by defining a new `Kconfig` option in the code, `IMA_POLICY_XFLTR_EVAL`, which is set before the evaluation.

The evaluation begins with environment preparation (Figure 6), such as detecting IMA policies and creating temporary files. The collection and processing of latency data is automated by bash scripts `ima_eval_time.sh`, `ima_eval_strace.sh`, and `ima_eval_ftrace.sh`, from where the commands `time`, `strace`, and `ftrace` are called. The test commands are executed 100 times while flushing system buffers and clearing memory caches between each round. A log file formatted as Comma-Separated Values (CSV) is created to record the time value in seconds or microseconds. The calculation results, skewness, and kurtosis are appended to the file.

One hundred latency values are obtained for each scenario in the matrix. The central tendencies for multiple sample sets are compared after skewness and kurtosis of the variable distributions are determined [48]. To establish normality, the mean is used for normal distribution [49] while the trimmed mean is used when there are outliers [42]. So that data present a normal distribution, skewness and kurtosis should be in the range of -2 ± 2 after trimming [50–52].

5.2.1. `Time` Command

In this evaluation, the bash command `time` is capable of millisecond measurement precision, compared with the centisecond level provided by GNU `time`. Only CPU time is collected, because it directly measures the time spent on kernel processing for a program.

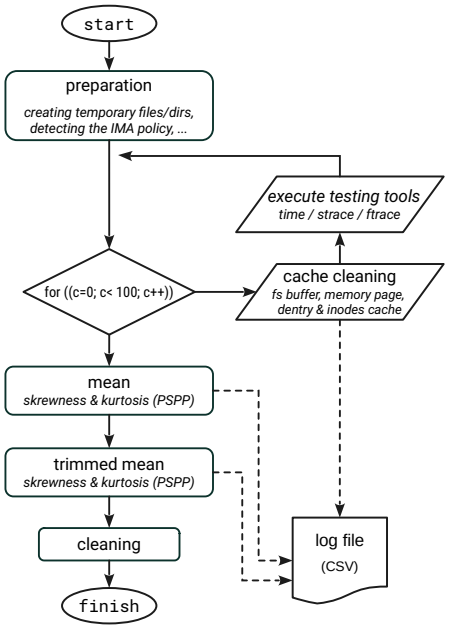


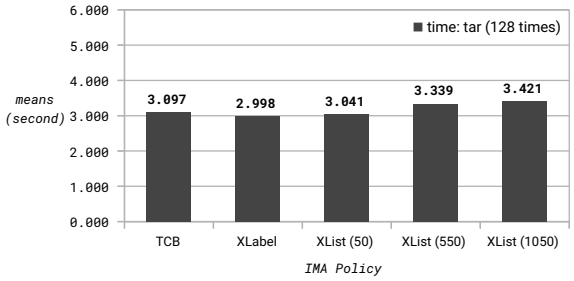
Figure 6. The workflow of performance evaluation for system latency.

As the target program in this measurement, GNU `tar` is executed in a loop of 100 iterations. In each round, nine text files are created prior to them being compressed by the `tar` command 128 times.

The results indicate that the skewness and kurtosis are within range and the mean was applied to represent the central location (tabular results in Figure 7a). Figure 7b shows that the means of the latency value in the policies of the TCB, XLabel, and XList with 50 records have little variation, while the XList file lists with 550 and 1050 records shows that latency is slightly higher. The XList with 1050 records has the highest latency value among all policy settings.

		TCB	**XLabel**	**XList (50)**	**XList (550)**	**XList (1050)**
`tar` (128 times)	**mean** (s)	3.097	2.998	3.041	3.339	3.421
	skewness	*−0.47*	*−0.72*	*−0.63*	*−0.86*	*−1.21*
	kurtosis	*−0.34*	*0.08*	*0.11*	*0.62*	*1.54*

(**a**) Tabular results from `time` command



(**b**) Means - `time` command

Figure 7. Evaluation results from `time` command (`tar` 128 times).

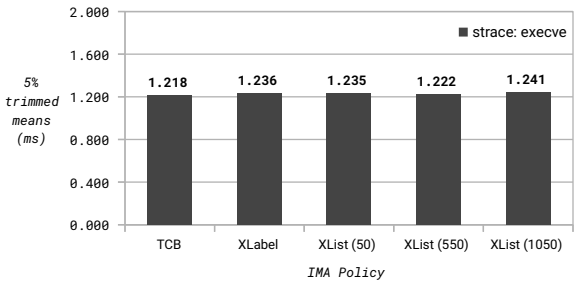
5.2.2. `strace`

`strace` (Version 5.3) evaluates the latency of policy processing from the userspace. For this purpose, `strace` measures the time consumed while processing two syscalls, `execve` and `mmap`, whose calling routines involve all three IMA hooks being evaluated, `ima_file_check`, `ima_bprm_check`, and `ima_file_mmap`. Since both `execve` and `mmap` are called by the `ls` command, the measurement can be conducted by applying `strace` to the `ls` action that lists files in a temporary directory.

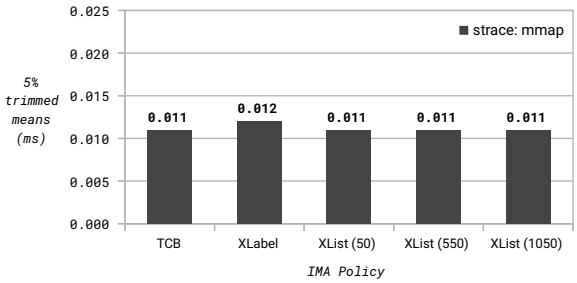
The evaluation results in the tabular results in Figure 8a are measured in milliseconds (ms). Examination of skewness and kurtosis of latency values indicated that some of them exceed the range of acceptance, and therefore a 5% trimmed mean was used in the analysis. Figure 8 shows that, for `execve` (Figure 8b) and `mmap` (Figure 8c), latency maintained a relatively constant level across all policies. The comparison for `execve` shows that the trimmed mean of TCB policy setting is slightly lower than others. However, its difference from the highest (XList with 1050 items) is only 0.023 ms.

		TCB	XLabel	XList (50)	XList (550)	XList (1050)
`execve`	**trimmed mean** (ms)	1.218	1.236	1.235	1.222	1.241
	skewness	*0.67*	*0.30*	*0.25*	*0.67*	*1.05*
	kurtosis	*0.16*	*−0.78*	*−0.89*	*−0.46*	*0.53*
`mmap`	**trimmed mean** (ms)	0.011	0.012	0.011	0.011	0.011
	skewness	*0.89*	*0.69*	*0.73*	*1.23*	*1.21*
	kurtosis	*0.08*	*−0.22*	*−0.01*	*1.30*	*0.70*

(**a**) Tabular results for `strace`



(**b**) Means - `strace` - `execve`



(**c**) Means - `strace` - `mmap`

Figure 8. Evaluation results for `strace`.

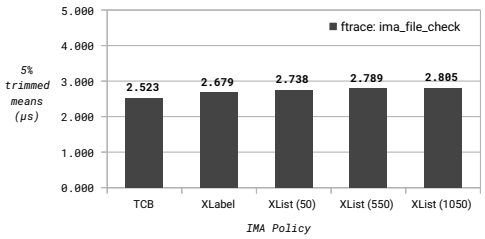
5.2.3. `ftrace`

The `ftrace` tool in kernel provides a complete interface for tracing kernel events and functions to evaluate the IMA hooks `ima_file_check`, `ima_bprm_check`, and `ima_file_mmap` (Table 1). In the evaluation process, a front-end command, namely `trace-cmd` (Version 2.6.1), is applied to call the `ftrace` interface. To reduce the number of repeat syscalls, a specific binary, `/usr/bin/hello`, is compiled as the target program. Kernel operations are simplified through the use of minimised logic, in which the `main` function contains only a `return` statement. The result of this serves to reduce the number of debug messages produced by the `trace-cmd` command. For each iteration, a report that contains the function graph for specific IMA hook functions is generated, from which the latency values measured in microseconds (µs) are filtered out immediately.

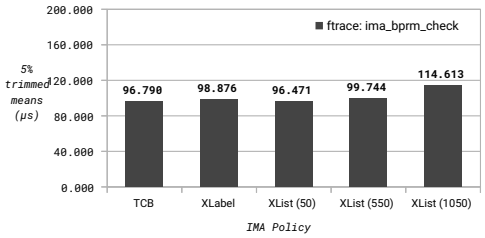
Due to high levels of skewness and kurtosis found in the data, a trimmed mean at 5% was applied. The tabular data in Figure 9a indicate that across all three hook functions, there is an upward trend in latency from the IMA TCB policy to the XList at 1050 items. The differences are minor for the `ima_file_check` function (Figure 9b) and significant for XList with 1050 items in `ima_bprm_check` (Figure 9c) and XList with 550 and 1050 items in `ima_file_mmap` (Figure 9d).

		TCB	XLabel	XList (50)	XList (550)	XList (1050)
`ima_file_check`	**trimmed mean** (µs)	2.523	2.679	2.738	2.789	2.805
	skewness	*0.79*	*0.20*	*0.40*	*0.65*	*0.41*
	kurtosis	*1.30*	*-0.07*	*-0.44*	*0.40*	*1.08*
`ima_bprm_check`	**trimmed mean** (µs)	96.790	98.876	96.471	99.744	114.613
	skewness	*1.17*	*0.81*	*0.70*	*0.77*	*0.30*
	kurtosis	*1.86*	*0.98*	*−0.20*	*−0.20*	*0.33*
`ima_file_mmap`	**trimmed mean** (µs)	3.492	3.865	4.376	10.149	16.527
	skewness	*0.62*	*0.62*	*0.72*	*0.67*	*0.35*
	kurtosis	*0.38*	*0.36*	*−0.39*	*0.04*	*−0.01*

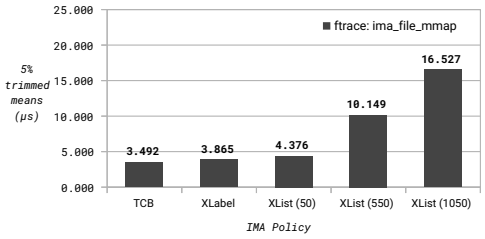
(**a**) Tabular results for `ftrace`



(**b**) Means - `ftrace` - `ima_file_check`



(**c**) Means - `ftrace` - `ima_bprm_check`



(**d**) Means - `ftrace` - `ima_file_mmap`

Figure 9. Evaluation results for `ftrace`.

5.3. Memory Use Performance Evaluation

Since the IMA XFilter functions are implemented at the kernel level, any measurement should focus on the memory usage during kernel processes. It is possible to measure the kernel memory usage of XFilter by identifying all allocated slabs utilised by kernel operations. The slab is comprised of sections of contiguous virtual memory divided into chunks of equal size [53]. This provides the allocation mechanism for kernel objects.

In this evaluation, the status of slab allocation is obtained from `/proc/meminfo`. Since IMA functions are not implemented as dynamic loading modules, only total slab usage is measured. Considering the time cost, 40 samples of slab information for each policy are collected. Each dataset is collected in a freshly bootstrapped system environment where the kernel and userspace applications run in a stable state.

The target system for the evaluation is deployed in the virtual machine environment. A bash shell script running on the host system monitors and controls the measurement via

a `ssh` connection (Figure 10), which restarts the guest machine after each round of data collection. When the rounds have completed, the data are processed to calculate mean and trimmed mean values.

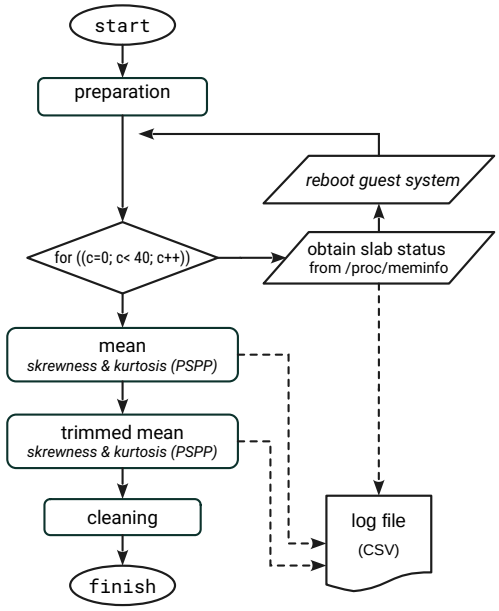
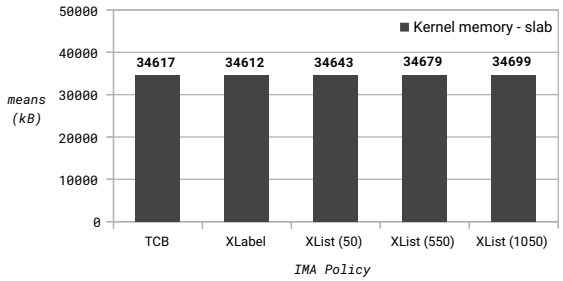


Figure 10. Memory usage performance evaluation workflow.

As shown in the tabular data in Figure 11a, skewness and kurtosis from the measurement results are within the acceptable range, and therefore the mean value is applied in the evaluation. A comparison across five policies (Figure 11b) shows that there is little variance between total slab allocation, while the largest differential value is 87 kB, which is insignificant when system memory provides more than 34,000 kB.

		TCB	XLabel	XList (50)	XList (550)	XList (1050)
slab size in `/proc/meminfo`	**mean** (kB)	34617	34612	34643	34679	34699
	skewness	*0.08*	*-0.01*	*-0.03*	*-0.08*	*0.01*
	kurtosis	*-0.80*	*-0.28*	*-0.75*	*-0.53*	*0.60*

(**a**) Tabular results for slab allocation



(**b**) Means - slab allocation

Figure 11. Evaluation results for slab allocation.

6. Discussion and Conclusions

This research proposes a flexible and fine-grained mechanism for policy-making in IMA measurement. Although two XFilter methods exist to address the same problem, the mechanisms behind their implementations and the user configuration processes are quite

different. For XLabel, the files to be measured are recorded as extended attributes in the filesystem layer, whereas XList introduces the provision for the creation and management of a pseudo-file in `securityfs`, which stores the file path of the target file. For both methods, token matching actions in rules parsing share the same processes and variables. Additionally, XLabel applies the command-line tools `getfattr` and `setfattr` to retrieve and set `xattr` values for the files to be measured. Meanwhile, a text editor is used to edit the XList configuration file that sets the file list to be written to the XList pseudo-file. Additionally, to simplify the system administration package management, integration is introduced.

The results of the functional evaluations suggest that both policy mechanisms work as expected and meet the requirement of IMA measurement. However, performance evaluations revealed advantages and shortcomings, which can be addressed in the development of a more effective and efficient solution.

Latency evaluations measure the time used for processing tasks and reflect how much CPU power is consumed. The evaluations measure latency for file operations through IMA hooks, syscalls, and user-space programs. The results reveal that XLabel policy latency is slightly higher than the default policy rule set, but with minimal difference in the trimmed mean. This small difference indicates that the `xattr` retrieval function has minimal impact on computational power. However, latencies for the XList policy rule sets show a greater difference, and the variance depends on the length of the file list. When a file list with 50 items was applied, the latency showed minimal variance with the XLabel policy but that variation increased markedly when the list increased to 550 and 1050 items. The variances can be explained by the combined impact of the length of the items list and how long it took for the list traversal period to complete, where the cumulative effect of more list items triggers an increasing number of search operations, which then leads to more context switching in the runtime. However, this does not explain the variation between the XList policies in which the hook `ima_file_mmap` is much higher than either `ima_bprm_check` and `ima_file_check`.

When compared to the above IMA hooks, measurement at the kernel syscall level showed a very different scenario. For both `execve` and `mmap` function calls, the latency values for all policies presented a stable trend. Even between the three XList policies, the differences in latency were insignificant. This result may be due to the number of function calls involved in the syscall interfaces, such that the overall measurement value is not sensitive to the latency values produced by the three IMA hooks, which account for a small number of function calls compared with other kernel functions.

The user-space evaluation performed by the `time` command provided a similar pattern to the IMA hooks. Nonetheless, the three XList policy measurements reveal a greater variance than that above. It is possible that the target program, `tar`, includes a large number of file operations with combined latencies aggregating as the length of the file list increases. This is different from the syscall level measurement, where the accumulation happens during function execution and is unrelated to IMA hooks.

The evaluation of memory consumption addressed overall kernel processing measurement rather than user-space programs because the IMA functions in the design are implemented at the kernel level. The evaluation results show that the memory consumption for the kernel as a whole was almost identical across all five policy mechanisms. Analysis of the XFilter code indicates two changes that may affect memory usage, with the first being the introduction of new variables. However, since only a few new variables are defined in this design, their impact is minimal. The other modification is the creation of the XList policy file list that is stored in the kernel memory space. With 1050 listed items in the evaluation, the amount of memory consumed is around 35 kB, which is a small proportion of the overall consumption rate of more than 34,000 kB. While Figure 11b illustrates that there is a tiny increase in mean memory consumption from 50 to 1050 items in `flist`, no evidence supports an increase in the number of file list items as a cause of this variance.

One possible shortcoming is that XLabel offers a greater attack surface than XList and is potentially more vulnerable. If an attacker achieves administrator privileges, then the attacker could manipulate the `xattr` adopted by XLabel. The vulnerability stems from the capacity `CAP_SYS_ADMIN`, originally requested by root, which has permissions for modification and deletion over `xattr` [54]. This may extend to the `xattr` used by LSM on SELinux and SMACK. The attack surface does not apply to XList because the bit-setting mechanism in the `inode` structure prevents any privileged users (including root) from modifying the file list through the pseudo-file interface once the IMA policy has been initialised.

A feasible approach to mitigate the potential XLabel vulnerability is to apply the Extended Verification Module (EVM), in which the system calculates a hash of `xattr` and signs it using a private key stored in the TPM device [55]. By validating the signature with the corresponding public key, attempted modifications can be detected early. However there is a time cost with the application of EVM due to the complexity of the solution.

Although this research covers most of what would be expected when developing new IMA policy mechanisms, some improvement is required. First, the performance evaluations are limited to specific policy scenarios. Expanding the range of policies may improve evaluation granularity, thereby highlighting the relationship between the length of the file list and function latency. In addition, including other LSM policies in the comparison can verify the suitability of XFilter as a replacement for the LSM-based IMA policy mechanism. Second, the cause of the increased latency observed in `ima_file_mmap` in contrast to `ima_bprm_check` and `ima_file_check` is still unknown. To explain this phenomenon, a deep investigation of the code may suggest why there are differences in the IMA hook implementation.

Generally, the design and evaluations suggest that the XFilter implementation in the IMA subsystem is feasible when more evaluations and validations have been undertaken. This research provides a reference for further in-depth development related to policy mechanisms in system integrity measurement.

Author Contributions: Conceptualisation, A.L. and W.D.; methodology, A.L.; software, W.D.; evaluation, A.L.; formal analysis, W.D.; writing—original draft preparation, W.D.; writing—review and editing, A.L.; supervision, A.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Trusted Computing Group. Trusted Computing. Available online: https://trustedcomputinggroup.org/trusted-computing (accessed on 1 May 2021).
2. Coker, G.; Guttman, J.; Loscocco, P.; Herzog, A.; Millen, J.; O'Hanlon, B.; Ramsdell, J.; Segall, A.; Sheehy, J.; Sniffen, B. Principles of remote attestation. *Int. J. Inf. Secur.* **2011**, *10*, 63–81. [CrossRef]
3. Francillon, A.; Nguyen, Q.; Rasmussen, K.B.; Tsudik, G. A minimalist approach to Remote Attestation. In Proceedings of the 2014 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 24–28 March 2014; pp. 1–6. [CrossRef]
4. Kasatkin, D.; Zohar, M. Integrity Measurement Architecture (IMA). Available online: https://sourceforge.net/p/linux-ima/wiki (accessed on 1 May 2021).
5. Son, J.; Koo, S.; Choi, J.; Choi, S.j.; Baek, S.; Jeon, G.; Park, J.H.; Kim, H. Quantitative analysis of measurement overhead for integrity verification. In Proceedings of the Symposium on Applied Computing, Marrakech, Morocco, 3–7 April 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 1528–1533. [CrossRef]
6. SUSE. Configuring SELinux—Security and Hardening Guide. Available online: https://documentation.suse.com/sles/15-SP1/html/SLES-all/cha-selinux.html#sec-selinux-support (accessed on 17 March 2021).

7. Schreuders, Z.C.; McGill, T.; Payne, C. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *ACM Trans. Inf. Syst. Secur.* **2011**, *14*, 2019604. [CrossRef]
8. Zhang, W.; Jaeger, T.; Liu, P. Analyzing the Overhead of Filesystem Protection Using Linux Security Modules. Available online: http://xxx.lanl.gov/abs/2101.11611 (accessed on 22 March 2023).
9. Radhika, B.S.; Kumar, N.V.N.; Shyamasundar, R.K.; Vyas, P. Consistency analysis and flow secure enforcement of SELinux policies. *Comput. Secur.* **2020**, *94*, 101816. [CrossRef]
10. The Linux Kernel Documentation. Smack-Linux Security Module Usage-The Linux Kernel User's and Administrator's Guide. Available online: https://www.kernel.org/doc/html/v5.4/admin-guide/LSM/Smack.html (accessed on 17 March 2021).
11. Rauter, T.; Holler, A.; Iber, J.; Krisper, M.; Kreiner, C. Integration of Integrity Enforcing Technologies into Embedded Control Devices: Experiences and Evaluation. In Proceedings of the 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), Christchurch, New Zealand, 22–25 January 2017 ; IEEE: Piscataway, NJ, USA, 2017; pp. 155–164. [CrossRef]
12. Sailer, R.; Zhang, X.; Jaeger, T.; van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In Proceedings of the 13th USENIX Security Symposium, San Diego, CA, USA, 9–13 August 2004; USENIX Association: Berkeley, CA, USA, 2004; Volume 13, pp. 223–238.
13. Sailer, R.; Jaeger, T.; Zhang, X.; van Doorn, L. Attestation-Based Policy Enforcement for Remote Access. In Proceedings of the 11th ACM Conference on Computer and Communications Security, Washington DC, USA, 25–29 October 2004; Association for Computing Machinery: New York, NY, USA, 2004; pp. 308–317. [CrossRef]
14. England, P.; Lampson, B.; Manferdelli, J.; Peinado, M.; Willman, B. A trusted open platform. *Computer* **2003**, *36*, 55–62. [CrossRef]
15. Garfinkel, T.; Pfaff, B.; Chow, J.; Rosenblum, M.; Boneh, D. Terra: A virtual machine-based platform for trusted computing. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, Bolton Landing, NY, USA, 19–22 October 2003; Association for Computing Machinery: New York, NY, USA, 2003; pp. 193–206. [CrossRef]
16. Bohling, F.; Mueller, T.; Eckel, M.; Lindemann, J. Subverting Linux' integrity measurement architecture. In Proceedings of the 15th International Conference on Availability, Reliability and Security, Virtual, 25–28 August 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 1–10. [CrossRef]
17. Muthukumaran, D.; Sawani, A.; Schiffman, J.; Jung, B.M.; Jaeger, T. Measuring integrity on mobile phone systems. In Proceedings of the 13th ACM Symposium on Access Control Models and Technologies, Estes Park, CO, USA, 11–13 June 2008; Association for Computing Machinery: New York, NY, USA, 2008; pp. 155–164. [CrossRef]
18. Nauman, M.; Khan, S.; Zhang, X.; Seifert, J.P. Beyond Kernel-Level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In Proceedings of the Trust and Trustworthy Computing, Berlin, Germany, 21–23 June 2010; Acquisti, A., Smith, S.W., Sadeghi, A.R., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 1–15. [CrossRef]
19. Stelte, B.; Koch, R.; Ullmann, M. Towards integrity measurement in virtualized environments—A hypervisor based sensory integrity measurement architecture (SIMA). In Proceedings of the 2010 IEEE International Conference on Technologies for Homeland Security (HST), Waltham, MA, USA, 8–10 November 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 106–112. [CrossRef]
20. Gopalan, A.; Gowadia, V.; Scalavino, E.; Lupu, E. Policy Driven Remote Attestation. In Proceedings of the Security and Privacy in Mobile Information and Communication Systems, Alborg, Denmark, 17–19 May 2011; Prasad, R., Farkas, K., Schmidt, A.U., Lioy, A., Russello, G., Luccio, F.L., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 148–159. [CrossRef]
21. Lauer, H.; Kuntze, N. Hypervisor-Based Attestation of Virtual Environments. In Proceedings of the 2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), Toulouse, France, 18–21 July 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 333–340. [CrossRef]
22. Ren, J.; Liu, L.; Zhang, D.; Zhang, Q.; Ba, H. Tenants Attested Trusted Cloud Service. In Proceedings of the 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, USA, 27 June–2 July 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 600–607. [CrossRef]
23. Cesena, E.; Ramunno, G.; Sassu, R.; Vernizzi, D.; Lioy, A. On Scalability of Remote Attestation. In Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing, Chicago, IL, USA, 17 October 2011; Association for Computing Machinery; pp. 25–30. [CrossRef]
24. Luo, W.; Shen, Q.; Xia, Y.; Wu, Z. Container-IMA: A privacy-preserving Integrity Measurement Architecture for Containers. In Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), Beijing, China, 23–25 September 2019; USENIX Association: Berkeley, CA, USA, 2019; pp. 487–500.
25. Steffen, A. The Linux Integrity Measurement Architecture and TPM-Based Network Endpoint Assessment. In Proceedings of the Linux Security Summit, San Diego, CA, USA, 30–31 August 2012.
26. Paverd, A.J.; Martin, A.P. Hardware Security for Device Authentication in the Smart Grid. In Proceedings of the Smart Grid Security, Halifax, NS, Canada, 21–23 August 2013; Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., et al., Eds.; Springer: Berlin/Heidelberg, Germany, 2013; pp. 72–84. [CrossRef]

27. Rauter, T.; Iber, J.; Krisper, M.; Kreiner, C. Supporting the Integration of New Security Features in Embedded Control Devices Through the Digitalization of Production. In Proceedings of the Systems, Software and Services Process Improvement, Ostrava, Czech Republic, 6–8 September 2017; Stolfa, J., Stolfa, S., O'Connor, R.V., Messnarz, R., Eds.; Springer International Publishing: Berlin/Heidelberg, Germany, 2017; pp. 360–371. [CrossRef]
28. Zeller, A. CHAPTER 4—Reproducing Problems. In *Why Programs Fail: A Guide to Systematic Debugging*, 2nd ed.; Morgan Kaufmann: Boston, MA, USA, 2009; pp. 75–103. [CrossRef]
29. Corbet, J.; Rubini, A.; Kroah-Hartman, G. An Introduction to Device Drivers. In *Linux Device Drivers*, 3rd ed.; O'Reilly Media: Sebastopol, CA, USA, 2005; Chapter 1.
30. Ge, X.; Talele, N.; Payer, M.; Jaeger, T. Fine-Grained Control-Flow Integrity for Kernel Software. In Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS&P), Saarbrucken, Germany, 21–24 March 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 179–194. [CrossRef]
31. Moreira, J.; Rigo, S.; Polychronakis, M.; Kemerlis, V.P. DROP THE ROP fine-grained control-flow integrity for the Linux kernel. In Proceedings of the Black Hat Asia 2017, Singapore, 28–31 March 2017.
32. Peffers, K.; Tuunanen, T.; Rothenberger, M.A.; Chatterjee, S. A design science research methodology for information systems research. *J. Manag. Inf. Syst.* **2007**, *24*, 45–77. [CrossRef]
33. Bolte, M.; Sievers, M.; Birkenheuer, G.; Niehorster, O.; Brinkmann, A. Non-intrusive virtualization management using libvirt. In Proceedings of the 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), Dresden, Germany, 8–12 March 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 574–579. [CrossRef]
34. DeMaio, D. openSUSE Leap 15.2 Release Brings Exciting New Artificial Intelligence (AI), Machine Learning, and Container Packages. Available online: https://news.opensuse.org/2020/07/02/opensuse-leap-15-2-release-brings-exciting-new-packages/ (accessed on 21 March 2021).
35. Rybczyńska, M. Bringing openSUSE Leap and SLE Closer. Available online: https://lwn.net/Articles/818382 (accessed on 21 March 2021).
36. Hoffman, C. Linux Distribution Basics: Rolling Releases vs. Standard Releases. Available online: https://www.howtogeek.com/192939 (accessed on 21 March 2021).
37. The Linux Kernel Organization. Active Kernel Releases. Available online: https://www.kernel.org/category/releases.html (accessed on 21 March 2021).
38. Chastain, M.E.; Germaschewski, K.; Ravnborg, S.; Engelhardt, J. Linux Kernel Makefiles. Available online: https://www.kernel.org/doc/html/latest/kbuild/makefiles.html (accessed on 21 March 2021).
39. Gregg, B. Ftrace: The Hidden Light Switch. Available online: https://lwn.net/Articles/608497 (accessed on 22 March 2021).
40. Netcraft. February 2021 Web Server Survey. Available online: https://news.netcraft.com/archives/2021/02/26/february-2021-web-server-survey.html (accessed on 23 March 2021).
41. Bowden, T.; Bauer, B.; Nerin, J.; Feng, S.; Seibold, S. The /proc Filesystem. Available online: https://www.kernel.org/doc/html/latest/filesystems/proc.html (accessed on 27 March 2021).
42. Wilcox, R.R. Trimmed Mean. In *Encyclopaedia of Research Design*; Salkind, N.J., Ed.; SAGE Publications, Inc.: Thousand Oaks, CA, USA, 2010.
43. DasGupta, A. The Trimmed Mean. In *Asymptotic Theory of Statistics and Probability*; Springer: New York, NY, USA, 2008; pp. 271–278. [CrossRef]
44. Chinthaguntla, K. Linux Package Management With YUM and RPM. Available online: https://www.redhat.com/sysadmin/how-manage-packages (accessed on 12 April 2021).
45. Howden, W.E. A functional approach to program testing and analysis. *IEEE Trans. Softw. Eng.* **1986**, *SE-12*, 997–1005. [CrossRef]
46. Molyneaux, I. Why Performance Test? In *The Art of Application Performance Testing: From Strategy to Tools*, 2nd ed.; "O'Reilly Media, Inc.": Sebastopol, CA, USA, 2014; pp. 1–10.
47. Vogel, B.; Steinke, B. Using SELinux Security Enforcement in Linux-Based Embedded Devices. In Proceedings of the 1st International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, Brussels, Belgium, 13–15 February 2008; ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
48. Kim, H.Y. Statistical notes for clinical researchers: Assessing normal distribution (2) using skewness and kurtosis. *Restor. Dent. Endod.* **2013**, *38*, 52–54. [CrossRef] [PubMed]
49. Wilcox, R.R. Trimming. In *The SAGE Encyclopedia of Social Science Research Methods*; Lewis-Beck, M.S., Bryman, A., Liao, T.F., Eds.; SAGE Publications, Inc.: Thousand Oaks, CA, USA, 2004. [CrossRef]
50. Kim, H.Y. Skewness. In *The SAGE Encyclopedia of Educational Research, Measurement, and Evaluation*; Frey, B.B., Ed.; SAGE Publications, Inc.: Thousand Oaks, CA, USA, 2018; Volume 4. [CrossRef]
51. George, D.; Mallery, P. Descriptive Statistics. In *IBM SPSS Statistics 26 Step by Step: A Simple Guide and Reference*, 16th ed.; Routledge: New York, NY, USA, 2020; pp. 112–120.
52. Cameron, A.C. Kurtosis. In *The SAGE Encyclopedia of Social Science Research Methods*; Lewis-Beck, M.S., Bryman, A., Liao, T.F., Eds.; SAGE Publications, Inc.: Thousand Oaks, CA, USA, 2004; Volume 0. [CrossRef]
53. Bonwick, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In Proceedings of the USENIX Summer 1994 Technical Conference, Boston, MA, USA, 6–10 June 1994.

54. The Linux Kernel Organization. xattr(7)—Linux Manual Page. Available online: https://www.usenix.org/legacy/publications/library/proceedings/bos94/bonwick.html (accessed on 7 May 2023).
55. Edge, J. The Return of EVM. Available online: https://lwn.net/Articles/394170 (accessed on 27 April 2021).