Towards Continuous Integrity Attestation and Its Challenges in Practice: A Case Study of Keylime

Margie Ruffin[1], Chenkai Wang[1], Gheorghe Almasi[2], Abdulhamid Adebayo[2], Hubertus Franke[2], Gang Wang[1]

[1]University of Illinois Urbana-Champaign, Urbana, IL 61801, USA
[2]IBM Research, Yorktown Heights, NY 10598, USA

***Abstract*—Continuous integrity attestation is vital for cloud providers to ensure the integrity of remote systems in a continuous manner. Current solutions, such as Keylime, rely on Trusted Platform Module (TPM) and Linux's Integrity Measurement Architecture (IMA) but often struggle to balance minimizing false alerts and maintaining effective threat detection. In this paper, we examine common causes of attestation failures in Keylime through active experiments. We find that false positives often stem from unscheduled OS updates, and we propose a dynamic policy generation scheme as a solution (validated over 66 days of experiments). Our false negative experiments reveal vulnerabilities in existing designs, including five common issues across Keylime and IMA. Exploiting these issues, attackers could evade detection in all tested scenarios. Our findings offer insights into common failures of continuous integrity attestation, and our proposed solution is being integrated into Keylime with community support, enhancing cloud security. Our code will be available at https://github.com/mruffin/Dynamic-Policy-Generator.**

I. Introduction

Remote attestation [1], [2], [3] is increasingly important to cloud providers to ensure the integrity of large fleets of remote systems. The Trusted Platform Module (TPM) is among the most widely adopted solutions for proving that remote attestation data is genuine [2]. TPM is a crypto-processor that implements the concept of hardware root-of-trust [4]: it can prove its own authenticity and initial state by providing a certificate signed by its manufacturer. In the past decade, most TPM-based solutions have been focused on *measured boot attestation*, which ensures the integrity of machine booting [5], [6], [7], [8], [9].

However, attestation is needed beyond the boot time. In recent years, continuously attesting a machine *long after boot* has drawn significant interest from major industrial stakeholders such as Amazon AWS [10], Intel [11], Microsoft Azure [12], IBM [13], and Redhat [14], [15]. This is often called "runtime" or "continuous" integrity attestation. Prior research on runtime attestation has been focused on verifying *specific software properties* and their code execution (control flow) [1], [16], [17], [18], [19]. However, cloud providers have a more emergent need to perform attestation for the entire file system [8], [20], [21], periodically building a trustworthy state snapshot for the machine based on the signatures of system executables. We call this *continuous integrity attestation*.[1] While industry players often develop their solutions in-house [11], [10], there is a growing open-source community working towards a general and standard framework for continuous integrity attestation [22].

Despite the different approaches, there are common research questions rooted in the fundamental trade-off for continuous integrity monitoring – minimizing false alerts while keeping attestation useful. Intuitively, permissive policies cause low sensitivity (i.e., missing true attacks), while strict policies tend to fire false alerts. This problem is especially challenging in today's cloud environment, which is subject to software diversity in both space (different types of machines) and time (quick update strategies), making the establishment of a reasonable attestation policy difficult.

Our Work. In this paper, we seek to empirically examine the potential failures from the *continuous* integrity attestation framework and understand their root causes. We specifically focus on Keylime's integrity attestation framework [23] for a case study. We choose Keylime because it is the most widely used open-source solution in this domain. Keylime is hosted by the Cloud Native Computing Foundation (CNCF) and is backed up by many industrial partners and a growing open-source community. Recently, cloud providers have started to integrate Keylime for remote attestation [24], and Red Hat Enterprise Linux also has built-in support for Keylime [15].

Research Questions. This paper was motivated by a set of questions to which we found no satisfactory answers in current practice or literature. (1) What are the common causes of false positives in Keylime's *continuous* integrity attestation? (2) To what extent can Keylime's continuous attestation detect different malicious attacks as opposed to policy failures caused by neglect or configuration errors? (3) How to systematically reduce (or eliminate) false positives and false negatives?

Evaluation of False Positives. To understand false positive errors, we take the integrity monitoring policy developed by IBM Research as the testing target (the policy is not used in production). We run the system for a week with *normal*

[1]Industrial practitioners (e.g., Keylime developers) have used the term "Runtime Integrity Attestation" to describe this type of system. However, we believe "Continuous Integrity Attestation" is a more accurate term, to differentiate it from solutions that focus on software execution/runtime integrity. We will explain further in Section II.

operations only. Normal operations include navigating the filesystem, opening and closing files, launching scripts to perform tasks, and performing system updates. We find that false positive errors are mainly caused by OS updates, which lead to discrepancies between the predefined Keylime policy and the measurement results.

To address this problem, we propose a *dynamic policy generation* for Keylime coupled with a data-center controlled update schedule for software updates. Dynamic policy generation involves a local "mirror" for Linux OS distributions for policy generation and system updates. To evaluate the effectiveness of the idea, we implement the prototype using Canonical's Ubuntu OS distribution for a proof-of-concept. We run a prolonged experiment for 66 days (31 days for a "daily update" experiment and 35 days for "weekly updates"). We demonstrate that Keylime can continuously perform attestation without triggering attestation failures throughout the experiment time period (36 system updates in total). For daily updates, it takes, on average, 2.36 minutes to update the policy, and the policy update overhead is small (1,271 lines, 0.16 MB on average).

Evaluation of False Negatives. To trigger false negatives, we empirically test three categories of attacks commonly faced by cloud providers that involve modifying system files: Ransomware, Rootkit, and Botnet Command and Control (C&C). We execute 8 real-world attacks and find that Keyline's continuous integrity attestation can detect all the attacks *if the attackers are unaware of the presence of Keylime*. However, during the experiments, we have identified five problems (P1–P5) across Keylime and Linux IMA (Integrity Measurement Architecture) that can enable *adaptive attacks*. Some of the problems are related to unmonitored directories by Keylime (P1) or unmonitored file systems by IMA (P3), while others are caused by deeper design issues, such as IMA's file evaluation logic (P4) and its inability to properly handle script interpreters (P5). Crucially, by exploiting one or a combination of these problems, adaptive attackers can *execute all 8 attacks without being detected by Keylime*.

Based on our results, we recommend four mitigation fixes for the discovered problems. We also use the results to reflect on the proper use case of Keylime and discuss the fitness of Keylime for general attack detection (see Section V).

We *responsibly disclosed* our findings to the Keylime open-source community, some of which have been acknowledged.

Summary of Our Contributions.

- Measure: we present the first empirical study of the attestation failures of Keylime's *continuous* integrity attestation framework, including both false positives and negatives.
- Analyze: we identify common causes of false positive/negative failure across Keylime and IMA.
- Fix: we propose a dynamic policy generation method to address the false positive problems. We provide recommended fixes for the false negatives. We will share the developed tool and testing suite with the community.

II. Background and Related Work

Remote Attestation and Trusted Platform Module. Remote attestation aims to establish the "trustworthiness" of one or more remote systems by comparing their actual state against a policy defined by the data center operator. The verifier (the program running attestation) is considered trusted; the attested systems (or the provers) are completely untrusted. The Trusted Platform Module (TPM) [2] is a device commonly used to establish that information from the attested prover system is genuine. The TPM is a chip that conforms to a secure cryptoprocessor standard, a dedicated microcontroller designed to secure hardware through integrated cryptographic keys [25], [4]. The TPM can prove its own identity (using certificates signed by its manufacturer) and the identity of the system it's installed on, and provides a "root of trust". The TPM also helps build "chains of trust" by allowing software measurements to be recorded in a nonrepudiable fashion. *Measured Boot Attestation* proves this chain of trust from hardware to the booted kernel, ensuring the integrity of the *boot process* [5]. *Runtime/continuous integrity attestation* picks up where the measured boot left off and focuses on software integrity once the kernel is booted.

Although measured boot attestation is well-studied [5], [6], [7], [8], [9], runtime/continuous integrity attestation and related techniques are less mature [20].

Integrity Attestation using IMA. Most runtime/continuous integrity attestation frameworks rely on Linux's Integrity Measurement Architecture (IMA). In its most basic mode, IMA collects and records hashes of files when they are opened before their content is accessed for reading or execution [26]. File hashes are recorded in a cumulative log and in a single TPM PCR (Platform Configuration Register).

We use Fig. 1 to describe the attestation process using IMA. Through attestation, the verifier (trusted machine) seeks to verify the state of the prover (untrusted machine). ❶ The verifier (trusted machine) issues a challenge by requesting the measurement list (IMA Log) as well as the TPM-signed PCR values, which represent an aggregate of measurements from the attested machine (prover). The aggregate is retrieved through a TPM Quote process[2], which cryptographically ensures its integrity. ❷ The verifier validates the IMA log against the aggregate. This is done to preclude the possibility of tampering with the log, either on the attested machine (prover) or en route. ❸ A runtime policy[3] is supplied by the verifier and used to validate the file measurements found in the IMA log. ❹ Finally, the verifier can reason about the trustworthiness of the prover's runtime integrity and issue a verdict.

Certain industrial practitioners have used the term "Runtime Integrity Attestation" to describe such IMA-based attestation frameworks [28], [11]. However, we believe that "Continuous Integrity Attestation" is a more accurate term because IMA

[2]TPM Quote refers to retrieving the values from the TPM's PCR using a cryptographic process [27].

[3]The policy is a collection of measurements of authorized files for a particular machine.

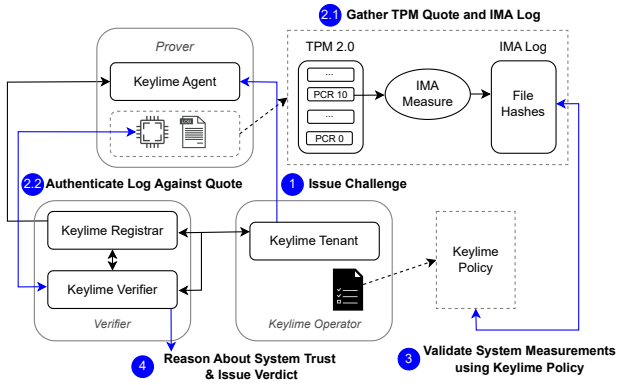

Fig. 1: Relationship between Keylime components, TPM 2.0, and Linux Integrity Measurement Architecture (IMA).

only measures file hashes rather than monitoring the detailed file execution at runtime. The term helps to distinguish itself from existing solutions that focus on software runtime execution integrity [29], [30], [17], [18]. For the rest of the paper, we will use "Continuous Integrity Attestation."

Keylime. We select Keylime [23] to study the practical challenges faced by continuous integrity attestation and measurement. Keylime is an open-source project hosted by the Cloud Native Computing Foundation (CNCF) [22], [23] and is the *only open-source solution* in this domain. We select Keylime for two reasons. First, it is widely recognized as the "go-to" software solution for remote attestation and continuous integrity measurement in modern distributed infrastructures [31]. It is used as the basis for various research endeavors [32], [33] and is growing in its adoption in the open-source community [34]. Alternatives to Keylime have been proposed [35], [36], [37]; however, none offer the stable contributions of a large open-source community or are adaptable enough to scale to large production environments. Second, Keylime has received significant industry attention from practitioners. Stakeholders such as IBM Cloud are actively integrating Keylime for remote attestation [24]. Keylime is also already a supported project for Red Hat Enterprise Linux [15].

As shown in Fig. 1, Keylime consists of 4 components: an *agent*, a *verifier*, a *registrar*, and a command line tool called the *tenant*. These components work together to realize the attestation process described earlier in this section. Here, we briefly describe the responsibilities of these components. Further details can be found in Keylime's documentation [28]. The "agent" is the only component that runs on the untrusted machine to be attested (i.e., *the prover*). Its main job is to collect IMA measurements and gather TPM quotes from the prover machine for attestation. The *verifier* machine is trusted, and it hosts two components: (a) The "Keylime verifier" is in charge of implementing the attestation process, and (b) the "Keylime registrar" supports the verifier by managing initial communication with the agent and guarding against spoofed or compromised TPM devices. Lastly, the "tenant" is a command-line management tool used to manage groups of attested nodes. Keylime can act as an alert system that notifies its operator by raising a flag if the continuous integrity monitoring fails (e.g., a system executable file has been altered without approval).

Related Work and Our Differences. Most prior work in this area focused on *code execution of specific software* (e.g., control-flow correctness) [29], [30], [17], [38], [18]; others focused on verifying dynamic system properties [39], or service integrity within a Trusted Execution Environment (TEE) [19]. Our paper is different as we focus on large-scale file system monitoring for cloud providers.

Two related works [40], [41] also focus on the cloud environment but have a different threat model from ours: they seek to ensure integrity for *users/tenants* who use *untrusted public cloud*. Other related works focus on different/orthogonal aspects such as the attestation of embedded systems [17], [16], [35], privacy issues of integrity measurements [42], and providing attestation support for legacy systems/clients [43]. A recent work uses Keylime to build a *virtual* trusted platform module (vTPM) that virtualizes the hardware root of trust for virtual machines' remote attestation [44].

III. False Positive (FP) Evaluation

We start with our false positive experiments. False positives (FP) represent benign activities that incorrectly cause Keylime to fire an alert on failed attestation.

A. False Positive Experiment

To trigger false positives, we set up a virtual environment (Canonical Ubuntu 22.04) and initiate the continuous integrity monitoring using an initial Keylime policy created by IBM Research as our target (the policy is not used in production). We let the system run for a week and only conduct benign operations with no attacks. As such, any alerts fired by Keylime for failed attestation are considered false positives.

The initial policy aims to capture the state of *executable files* in the target machine to perform attestation (executable files are also what IMA measures). The policy is constructed as the following: a bash script recursively goes into each directory in the root directory "/" until it reaches the bottom-most file, takes the SHA256 hash for *executable* files, and writes it to the policy. It focuses on files with the "executable" bit set to include files such as dynamic libraries and kernel modules. Note that the initial policy is designed to be permissive but also with false positives in mind. For instance, it excludes files specifically used for the containers and customization scripts that did not originate from Canonical. While not comprehensive, it provides a just-in-time snapshot of a machine that, in testing, undergoes little to no changes. This policy is designed for cloud nodes with a stable set of executables.

B. Causes of False Positives

During the one-week testing, we observed alerts/errors that stopped Keylime from attesting due to two main reasons.

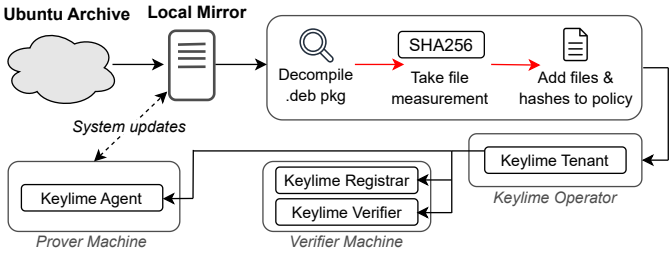

Fig. 2: The dynamic policy generation framework and its interaction with Keylime components.

System Updates. Ubuntu performs OS updates automatically if not otherwise configured [45], which was the main case of attestation errors. (1) The first type of error is "hash mismatch", which means both the IMA log and the policy contain the file, but the hashes do not match (i.e., modified file or file name). (2) The second type of error is "missing file in the policy", which means there exists a file in the IMA Log, but it is not present in the policy (i.e., newly added files).

SNAPs. SNAPs are applications containerized with their entire set of dependencies. The purpose of SNAPs is to avoid situations in which competing applications on the same host require mutually incompatible sets of dependencies [46], [47]. Each Ubuntu SNAP runs in an isolated sandbox, in the "`/snap`" directory, and appear on the policy as "`/snap/core20/version_num/file_name`". However, since IMA is unaware of the containerized nature of SNAPs, measurements of SNAP binaries appear truncated, without their SNAP prefixes, e.g., as "`/file_name`". Keylime raised an error because it failed to match the measured (truncated) file name. This problem is not specific to SNAPs but would occur to any containerized execution, or files executed under chroot environment.

C. Solution: Dynamic Policy Generation

We explore solutions to address the identified problems. For SNAP files, there are two ways to address them. (a) We can force a consistent file naming format between the target policy and the IMA log. This can be done simply by post-processing the target policy and scrubbing the prefix `/snap/core20/version_num/` and other variations of the file names. (b) Another option is to simply disable SNAP on the machine. Since SNAP is not part of the base OS, by default, we can disable SNAP to prioritize the efficiency of the attestation.

Second, the system update problem is more challenging, and thus, we focus on this problem for the rest of this section. To maintain the validity of the policy while keeping the system current with the latest updates, we propose a dynamic policy generation scheme. The idea is that every time before a major update is scheduled to be implemented, we create a new policy to proactively account for the dynamic changes a machine might encounter. While the idea may appear simple, this is the first solution that allows Keylime to run continuously without false positive errors (see experiments later).

Dynamic Policy Generation Framework. We implement a dynamic policy generation prototype using Ubuntu 22.04 LTS Jammy Jellyfish [48]. The idea is illustrated in Fig. 2. The cloud operator can use any stable distribution of their chosen operating system. Our framework follows these steps: (1) identify updates in advance, (2) generate policies based on updates, and (3) preempt system updates to provide Keylime with the policy for attestation.

Implementing a Mirror of OS Distribution. To control updates, operators disable "Unattended Upgrades" and create a local mirror of the Ubuntu distribution with sufficient storage (e.g., 1TB). This mirror includes the "Main," "Security" and "Updates" repositories, essential for a base OS. Other repositories such as "Universe" or "Multiverse" are not measured because they are not needed to run a base OS. This allows controlled updates and prevents overloading the distribution's web server with requests to pull down packages.

Dynamic Policy Generator. The generator refreshes the mirror, measures executable files from the "Main," "Security," and "Updates" repositories, and filters duplicates. To measure the files, we download and uncompress the package from the mirror and extract its executable files and filenames. We iterate through the files using their filenames and take their SHA256 measurements. Finally, the measurements, along with the filenames, are written into the policy. This process is repeated for every package in the three sub-repositories. We only consider executables because IMA only considers them when it measures files inside the challenged party. Operators can update policies efficiently by incorporating only modified or new executables. The updated policy is issued to the Keylime agent, which compares real-time measurements from the kernel against the policy. A key advantage of dynamic policy generation is that we can account for specific package updates and append new hashes to the existing policy, which is more efficient than regenerating the policy entirely.

Handling Kernel Modules. Executing the update in practice faces practical challenges from kernel modules. Kernel Modules can be loaded and unloaded into the kernel on demand. They can extend the kernel's functionality without the need to reboot [49]. There are two issues to consider. First, a machine can have many kernels (including outdated ones), and the policy should only consider one kernel that the machine currently uses and disallow outdated kernel modules. Second, during a system update, a new kernel can be installed, but it will not run before rebooting the system. As such, when a machine performs an update without rebooting, the policy can tentatively ignore the new kernels. This also means the policy will need to be updated for new kernels before the reboot.

Handling Policy-File Consistency During Update. During updates, Keylime continues attestation using the newly issued policy. To avoid errors from mismatches between old files and the new policy during the brief update window (e.g., 5

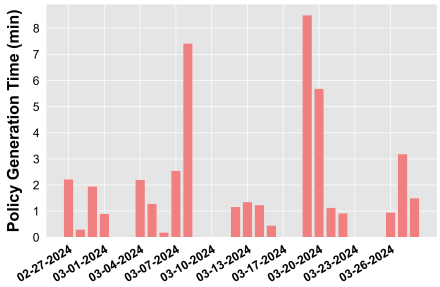

Fig. 3: The time it takes to update an existing Keylime policy (daily update).

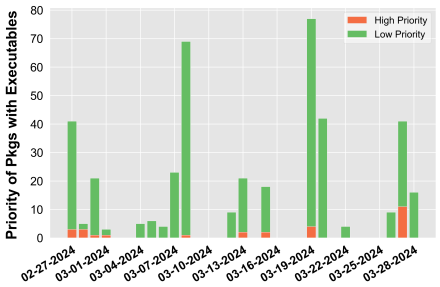

Fig. 4: The number of newly added and changed packages that contain executables per update (daily update).

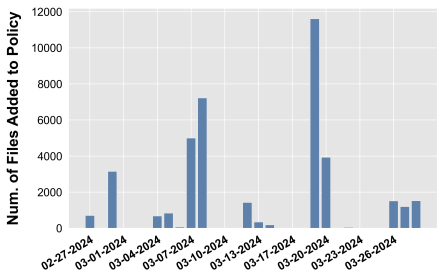

Fig. 5: Number of added and changed file entries to policy for each policy update (daily update).

minutes), the updated policy retains existing entries and adds only new or modified files. This ensures the system remains in policy throughout the update. Deduplication to remove outdated hashes can occur post-update.

D. Evaluation

We evaluate our dynamic policy generator's *effectiveness* and *overhead* with two experiments. In the first experiment, the operator set the system to update every day (similar to the default updating frequency of "Unattended Upgrades") and ran the system for 31 days (from February 26 to March 28, 2024). In the second experiment, we hypothesize an operator who decides to update less frequently (i.e., weekly) to see if there is an extra benefit or cost (running for 35 days, from May 6 to June 3, 2024). Due to space limitations, we present the first experiment in detail and put the second experiment result in supplementary materials [50].

Effectiveness of Reducing False Positives. Across the two experiments (66 days, 36 updates in total), Keylime *did not fire any false positive alerts* for the vast majority of the system updates (with one exception, which will be explained below). Zero false positives were fired during the normal operation periods after the system update. This confirms that our proposed method is effective in reducing false alerts.

During the experiment, the only time when Keylime stopped attestation was due to a particular configuration of our experiment setup. This error was due to our incorrect setup: the Ubuntu mirror was synced at 5:00 AM daily. On March 27, 2024, the release was issued *after* our mirror was synced and there was a misconfiguration from the operator (human error) who installed the updates using the official distribution instead of the local mirror, which fired the false positive. To prevent this error from occurring, the operator should have updated the agent machine from the local mirror.

Time Overhead to Update the Policy. Fig. 3 shows the time overhead of updating the policy each day. The result shows that updating the Keylime policy daily is a feasible approach; it can be done rapidly. For most of the days, the policy update only takes less than 10 minutes. On average, it takes 2.36 minutes with a standard deviation of 5.26. This is because policy update only measures hashes for new/updated files.

Experiment	# Low-P Pkgs	# Hig-P Pkgs	# of Files Updated	Time (mins)
Daily Update	15.6	0.9	1,271	2.36
Weekly Update	76.4	2.6	5,513	7.50

TABLE I: **Result Summary**. The average number of packages (high and low priority), number of files, and time it takes to generate a new policy for the daily- and weekly-update experiments.

Number of Involved Packages Per Update. We examine the number of packages included in each system update. Here, we only consider the packages that contain *executables*. As shown in Fig. 4, the majority of updates have less than 30 package updates that involve executable files. The average number of updates is 16.5, with a standard deviation of 26.8. If we further examine the *high-priority* package updates, the number is even smaller. We extract the package priority labels ranging from high-priority (including "Essential", "Required", "Important", and "Standard",) to low-priority (including "Optional", and "Extra"). The daily update only has 0.9 high-priority package updates (with executables) with a standard deviation of 2.2.

Policy Size Changes Per Update. Finally, we examine the number of entries written to the Keylime policy for each update. Fig. 5 shows the number of file entries. The number is higher than the number of packages since each package can contain multiple executable files. On average, each update only adds 1,271 lines to the policy file, which corresponds to a file size change of 0.16 MB. This is small compared to the original policy size (323,734 lines, 46 MB on day 1). Overall, our result suggests that the policy updating overhead is small and supporting daily updates is feasible.

Impact of Reduced Update Frequency. Our second experiment aims to explore if less frequent updates (i.e., weekly) would significantly increase the update overhead. The result is summarized in Table I. Overall, the result suggests that weekly updates have a comparable (and a slightly higher) cost per update compared with daily updates. We don't recommend this option given the potential negative consequences of delayed

important updates, as it may leave the system vulnerable.

IV. False Negative (FN) Evaluation

False negatives represent malicious activities that fail to be captured by Keylime's continuous integrity monitoring. In this section, we present the false negative evaluation to understand to what extent Keylime can be used to detect malicious attacks that involve modifying system files.

A. Experiment Setup and Attack Selection

For this experiment, we use the *new policy* derived from the false positive experiment (instead of the original policy) because the new policy can properly contain false positives. Before each attack, we make sure Keylime is set up correctly to perform attestation. Each experiment is conducted independently with the *same initial state* of the machine. We use an Ubuntu 22.04 LTS virtual machine with all software packages' versions aligned with the same mirror operated for the false positive evaluation.

We consider the scenario where the attacker has already gained access to the compromised system, with an intent to deploy attack payloads or to gain more capabilities. We test three categories of attacks that are commonly faced by cloud providers: Ransomware, Rootkit, and Botnet Command and Control (C&C), using a combination of open-source PoCs (i.e., proof-of-concept exploits) and real-world virus/malware samples. We understand that these categories are not an exhaustive list of threats. When selecting attack samples, we mainly consider the goals and capabilities of IMA and Keylime. More specifically, IMA and Keylime aim to attest that the loaded executables are trusted according to the policy. As such, we select attack samples that have the potential to alter, remove, or hide executable files from the running systems, which are within the scope of continuous integrity monitoring.

In total, we execute 8 attack samples, as shown in Table II. We have stopped testing new attack samples (after the 8 attacks) as the attack samples no longer trigger new *types* of false negatives to derive new insights. We provide further details about each attack in the supplementary materials [50].

B. Experiment Results

By running the aforementioned attack samples in the target environment, we reveal 5 common problems (P1–P5) of Keylime's continuous attestation. These problems have led to a successful execution of the attack payload which resides in the system temporarily or persistently without causing any failed attestation to trigger an alert. On a long-lived system, this could mean backdoors or APT (advanced persistent threats) reside in the running systems persistently for a long time due to the "false negative" attestation result..

Table II list the 5 problems. P1–P2 are problems of Keylime while P3–P5 are problems of IMA. The "*basic*" column shows the detection result if the attacker is unaware of the presence of Keylime while performing the basic attacks. The "*adaptive*" shows the scenario where attackers take advantage of the problems of Keylime/IMA (P1–P5) to evade detection. We find that all attacks can evade Keylime's detection if they exploit one or a combination of the discovered problems.

P1: Unmonitored Directories (Keylime). We discovered a vulnerability inherent in the Keylime policy which had a permissive setup that ignored some directories. We could avoid triggering any alerts if we executed attacks in these directories. One prominent example is the `/tmp` directory, where we could perform decompression, compilation, and download without detection. We discovered this issue when we first ran `Mortem-qBot-Botnet_Src`, which had a deployment script that utilized `/tmp` as its working directory. Then, we found this problem applied to all other attacks. We confirmed that other attacks could be deployed and executed in `/tmp` and remained undetected until a reboot. We further inspected the Keylime policy and found it excluded the `/tmp` directory altogether. This is inherent from the original policy (the exclusion was initially to improve attestation efficiency and reduce false positives).

P2: Incomplete Attestation Log (Keylime). By default, Keylime would stop polling upon detecting discrepancies between the IMA log and the policy. This will lead to an "incomplete" attestation log and can be further exploited by attackers. We discovered this problem during an attack when we accidentally triggered Keylime's attestation failure. With this exploit, the attacker can first trigger a false positive (e.g., by adding benign executable files that are not in the policy) to cause Keylime to stop polling and produce an incomplete attestation log. After that, the attacker can execute the attack (e.g., by running an executable). Note that the attack will not be written to the attestation log because the false positive has paused Keylime's attestation. When Keylime restarts attestation, the unresolved false positive will pause it again to produce another incomplete log. Operators can manually resolve the FP, but it will take time, and this provides a time window for the attacker to execute attacks.

P3: Unmonitored File Systems (IMA). During the attack experiments, we discovered that the IMA policy (derived from Keylime's official documentation) had elected to ignore a range of file systems, such as `tmpfs`, `procfs`, `debugfs`, `ramfs`, `securityfs`, `overlayfs`. Note that `tmpfs` and `procfs` are commonly writable directories. This means attackers may take advantage by executing the attack directly from `/proc`, for instance. Since IMA ignores `/proc` during its measurements, Keylime is thus blind to the corresponding activities. This would allow the attack to persist until a reboot and remain undetectable throughout the compromise.

P4: A Lack of Re-Evaluation (IMA). We discovered a design issue in IMA that *IMA measurement is done only once per executable*. This might be a reasonable design choice for IMA, but it can lead to problems when used for attack detection. More specifically, when testing *Diamorphine* and *Vlany* in the `/usr` directory, we expected Keylime to detect them. However, we observed that the execution of the malware

Name	Detected? Basic	Adaptive	Problems Exploitable P1	P2	P3	P4	P5	Mitigat.
Ransomware:								
AvosLocker	✓	✗	●	●	●	●		✓*
Rootkit:								
Diamorphine	✓	✗	●	●	●	●	●	✓*
Reptile	✓	✗	●	●	●	●	●	✓*
Vlany	✓	✗	●	●	●	●	●	✓
Botnet C&C:								
Mirai	✓	✗	●	●	●	●	●	✓*
BASHLITE	✓	✗	●	●	●	●	●	✓*
Mortem-qBot	✓	✗	●	●	●	●	●	✓*
Aoyama	✓	✗	●	●	●	●	●	✗

TABLE II: **Attacks tested against Keylime**. Legend: ✓—The sample execution is detectable by Keylime; ✓* The sample execution is detectable upon reboot / fresh attestation; ✗—The sample execution is not detectable by Keylime; ●—This attack sample may take advantage of the discovered problem. The column "basic" means basic attacks where adversaries are unaware of the presence of Keylime and execute the attacks normally. The column "adaptive" means adaptive attacks where adversaries take advantage of the discovered problems (P1–P5) to evade Keylime.

was never reported in the attestation log (before a reboot). Upon investigation, we discovered that it was because the installed malware was first put to "`/tmp`" and then moved to its final destination "`/usr`" for execution. This malware was not detected by Keylime when it was first added to "`/tmp`" (the reason is explained in P1). However, IMA still evaluated the malware since IMA did not ignore the "`/tmp`" folder. After it was moved to `/usr` directory, it also did not appear in the IMA attestation log even though it was executed in the destination directory. The reason was that IMA, by design, does not *re-evaluate* an identical file with a different path within the same file system. This gives attackers more flexibility and amplifies the attack's impact.

To exploit this, the attackers would first put their attack payload in a directory that IMA evaluates but Keylime ignores. Then, the attacker can move the payload to their destination directories for execution. Fundamentally, this is possible because IMA and Keylime have different mechanisms for their exclusion (i.e., IMA may ignore a file system while Keylime ignores specific directories). Since IMA does not re-evaluate a file with a moved path under the same filesystem, Keylime would never know the path/directory change.

P5: Scripts and Interpreters (IMA). This was observed within attack examples that either come with a Makefile or scripts (e.g., Python and Bash) for compilation and deployment. We found that IMA treated scripts' execution drastically differently based on how it was invoked: if it was executable directly invoked and relied on shebang to load (i.e., `./script.py`), the script file would be attested. However, if it was using an interpreter to load the script (i.e., `python ./script.py`), the interpreter would be attested (i.e., `Python`) instead of the script itself. In this case, attesting the interpreter (i.e., `Python`) is not useful for detecting the attacks. This challenges a fundamental design choice of IMA when many services and critical system components are implemented using a scripting language. From the attackers' perspective, they could circumvent attestation by simply implementing the attack or invoking the attack via interpreters. As shown in Table II, this problem applies to all 8 attack samples except for AvosLocker, which only has a binary.

C. Mitigation Solutions

Based on our analysis, we provide recommendations for mitigating the discovered problems. These recommendations aim to improve the capability of Keylime and IMA to detect attacks that involve unauthorized changes to the system executables. However, we emphasize this does not necessarily make Keylime ready to handle all types of attacks. More specifically, we propose changes in (1) Keylime policy/configuration, (2) Keylime implementation, and (3) IMA implementation. We have run experiments with the recommended changes in (1) Keylime policy/configuration and obtained positive results. However, we unfortunately cannot implement the changes for (2) and (3) due to their complex nature and our resource constraints. The last column ("Mitigat.") in Table II lists the theoretically possible outcomes if the recommended changes were implemented: 7 out of 8 attacks can be detected by Keylime upon reboot or fresh attestation, except for "Aoyama." The reason is that Aoyama is fully implemented in Python. It can exploit the script interpreter problem (P5) since P5 is difficult to fully mitigate (see reasons below).

Enriching Keylime/IMA Policies. Keylime or IMA ignores certain directories or filesystems (P1, P3), which allows adversaries to execute their attacks. A straightforward fix is to enrich the Keylime policy by adding the missing entries of executables from the ignored directories or filesystems. While this may slightly increase the overhead of attestation, the impact is manageable. Through a quick test, we believe the additional coverage on filesystems and directories is feasible given systems under normal operations do not have executables routinely appearing in those places (e.g., "`/tmp`").

Improving Keylime's Attestation Process. To counter P2, a potential solution is to change the design of Keylime such that it *does not stop polling upon errors* during attestation. Even though we agree that "failed attestation results are not trustworthy," we still recommend that Keylime should not return incomplete attestation results but instead should always complete the full attestation process. Even in the presence of false positives, Keylime should not undermine the ability/possibility of discovering real discrepancies caused by attacks in the system.

Improving IMA Design: Re-Evaluation. IMA's design problem is more complicated to address. For P4, IMA does not re-evaluate the same executable after relocation to a new directory within the same filesystem—to the operating system, it is identical. However, with Keylime's directory-based tracking and filtering, this becomes a problem. There are two possible solutions: (1) IMA can provide the directory information in the IMA log and re-evaluate the file upon directory changes,

or (2) Keylime needs a new mechanism to do filtering and file tracking *without depending on paths or directory information* (e.g., by relying on checksums themselves).

Improving IMA Design: Script Invocations. Fixing P5 is more challenging as it involves multiple parties: Keylime, IMA, and script interpreters. One possible direction is to have Keylime implement separate components to support the evaluation of script invocation. With a dedicated kernel module monitoring file operations of known interpreters, Keylime would no longer be limited by the design of IMA. A key challenge to realize this idea is to effectively distinguish code from data through interpreters.

Alternatively, a recent advancement in the Linux kernel community may offer a promising solution. More specifically, a patch set named "script execution control" allows interpreters that open executable scripts to set a special flag to inform the kernel that the file opened is executable in the interpreter [51]. If the common/popular interpreters (e.g., Python, and Bash) opt-in to use this new feature, it would eventually be possible to mitigate the issue in common/popular script interpreters. Keylime operators may even set the policy to only allow/-support interpreters that support "script execution control" to reside in the attested systems.

V. Discussion and Conclusion

In this paper, we empirically examine the failures from Keylime's continuous integrity monitoring framework and understand their root causes. In the following, we discuss key findings and implications.

Dynamic Policy Generation. To address the false positive errors (introduced by system updates), we develop a dynamic policy generator and confirm its effectiveness. In addition, we show that the policy update is highly efficient for practical deployment (it takes 2.36 minutes on average for a daily update). However, there is still room for improvement. For example, the current method requires individual operators to build file hashes themselves for the packages. This can be substantially improved if file hashes in packages are generated and then signed by the package maintainers (similar to `ostree` [52]). This would allow operators to know that what they are running is indeed trusted. Keylime is currently compatible with Linux-based operating systems (e.g., microOS, Ubuntu, and Red Hat Enterprise Linux). Our paper has shown that dynamic policy generation works for Ubuntu. As long as other Linux-based OSes have a similar structure for package release, our method can be potentially adapted to work with them. Keylime is not currently compatible with Windows, which has its own remote attestation architecture [53]. As such, our framework does not apply to Windows.

Proper Use Cases of Keylime. We again emphasize that Keylime and IMA are *not* designed to capture all types of attacks. However, malicious activities that involve modifying system executables should be within scope. Focusing on these types of attacks, we provide valuable lessons to Keylime operators and developers regarding the potential vulnerabilities and directions of improvements in Keylime policies and implementation. The result suggests that any rules that elect to skip attestation should be cautiously used—this is especially true when they are *wildcards* of directories or filesystems. In addition, we argue that Keylime (or similar remote attestation solutions) should not be used as an Intrusion Detection System (IDS). Instead, it should be only used to verify that *a known list of executables and static files* are intact. In other words, Keylime should be positioned to ensure existing essential system components are not maliciously alerted (i.e., in compliance). Keylime cannot make reliable decisions on unknown executables or unmonitored directories.

Limitations and Future Work. Our current experiments are limited to Ubuntu Linux (Jammy Jellyfish 22.04). Future work could include other operating systems for the test like Red Hat Enterprise Linux (RHEL). In addition, our paper is focused on Keylime since it is the only open-source solution that is suitable for a production cloud environment (also considering its real-world integration with Red Hat Enterprise Linux, MicroOS, and cloud providers). Future work is needed to explore whether the problems are generalizable to other attestation frameworks for runtime/continuous integrity monitoring. Finally, our false negative experiments have a limited number of attack categories/samples. However, these attacks are within the scope of the attestation workflow and have exposed problems that are universally applicable to similar attacks (see Table II). Future work can test more diverse attack types and attack samples to generalize the findings.

Responsible Disclosure and Result Sharing. We have shared our findings with Keylime, the Linux kernel community, and RedHat. First, we reported the false positive and false negative results to Keylime and received acknowledgments. Second, we wrote a detailed report regarding the issues associated with IMA and shared it with Linux kernel developers. The community was receptive to the findings and verified that IMA behaved as we observed in our experiments. The community was welcoming to patches to address the problems and had begun to discuss ways to develop them. For example, they pointed out a recent effort on "script execution control", which offers a possible path to addressing the script innovation problem in Keylime (a discussion has been added to Section IV-C). Third, we shared our idea of dynamic policy generation with RedHat. We are currently discussing the feasibility of extending the scheme to RedHat and also improving the file hash generation process for better security.

VI. Acknowledgments

This work was supported by the IBM-Illinois Discovery Accelerator Institute and the NSF Graduate Research Fellowship Program under Grant No. 21-46756.

References

[1] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn, "Attestation-based policy enforcement for remote access," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. Washington DC, USA: ACM, 2004, p. 308–317.

[2] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen, "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63–81, Jun. 2011.

[3] G. Chen, Y. Zhang, and T.-H. Lai, "Opera: Open remote attestation for intel's secure enclaves," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. London, United Kingdom: ACM, 2019, p. 2317–2331.

[4] T. T. C. Group, "TPM Main Part 1 Design Principles," https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf, March 2011.

[5] J. D. Osborn and D. C. Challener, "Trusted Platform Module Evolution," *Johns Hopkins APL Technical Digest*, vol. 32, no. 2, pp. 536–543, 2013.

[6] R. Haas and M. Pirker, "The state of boot integrity on linux - a brief review," in *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES)*. Vienna, Austria: ACM, 2024.

[7] A. Raghuramu, L. Cao, P. Sharma, M. Sánchez, J.-M. Kang, C.-N. Chuah, D. Lee, and V. Saxena, "Metered boot: Trusted framework for application usage rights management in virtualized ecosystems," *IEEE Transactions on Network and Service Management*, vol. 19, no. 3, pp. 2238–2250, 2022.

[8] S. Hosseinzadeh, B. Sequeiros, P. R. M. Inácio, and V. Leppänen, "Recent trends in applying tpm to cloud computing," *Security and Privacy*, vol. 3, no. 1, p. e93, 2020.

[9] R. Wang and Y. Yan, "A survey of secure boot schemes for embedded devices," in *2022 24th International Conference on Advanced Communication Technology (ICACT)*. Pyeongchang, South Korea: IEEE, 2022, pp. 224–227.

[10] AWS, "The security design of the aws nitro system," https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf, 2024.

[11] Intel, "Runtime integrity measurement and attestation in a trust domain," https://www.intel.com/content/www/us/en/developer/articles/community/runtime-integrity-measure-and-attest-trust-domain.html, 2024.

[12] Microsoft, "TPM attestation overview for azure," https://learn.microsoft.com/en-us/azure/attestation/tpm-attestation-concepts, 2024.

[13] W. Ozga, P. Sagmeister, T. Visegrády, and S. Dragone, "Scalable attestation of virtualized execution environments in hybrid- and multi-cloud," *arXiv 2304.00382*, 2023.

[14] M. A. Silva, G. Almasi, J. Bottomley, L. Sturmann, and M. Peters, "Keylime's durable attestation makes security auditable," https://next.redhat.com/2023/04/25/keylimes-durable-attestation-makes-security-auditable/, 2023.

[15] RedHat, "Red hat enterprise linux 9: Deploying keylime for runtime monitoring," https://docs.redhat.com/fr/documentation/red_hat_enterprise_linux/9/html/security_hardening/proc_deploying-keylime-for-runtime-monitoring_assembly_ensuring-system-integrity-with-keylime#proc_deploying-keylime-for-runtime-monitoring_assembly_ensuring-system-integrity-with-keylime, 2024.

[16] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-flat: Control-flow attestation for embedded systems software," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Vienna, Austria: ACM, 2016, p. 743–754.

[17] M. Geden and K. Rasmussen, "Hardware-assisted Remote Runtime Attestation for Critical Embedded Systems," in *2019 17th International Conference on Privacy, Security and Trust (PST)*. Fredericton, NB, Canada: IEEE, Aug. 2019, pp. 1–10.

[18] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti, "ScaRR: Scalable runtime remote attestation for complex systems," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 121–134.

[19] M. Morbitzer, B. Kopf, and P. Zieris, "GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments," http://arxiv.org/abs/2202.07380, May 2022.

[20] W. Ozga, D. Le Quoc, and C. Fetzer, "TRIGLAV: Remote Attestation of the Virtual Machine's Runtime Integrity in Public Clouds," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. Chicago, IL, USA: IEEE, Sep. 2021, pp. 1–12.

[21] W. Ozga, D. L. Quoc, and C. Fetzer, "Weles: Policy-driven runtime integrity enforcement of virtual machines," *CoRR*, vol. abs/2104.14862, 2021.

[22] CNCF, "Keylime," https://keylime.dev/, 2024.

[23] ——, "Keylime Development — Gitub Repository," https://github.com/keylime/keylime, 2024.

[24] K. Foy, "Keylime security software is deployed to ibm cloud," https://news.mit.edu/2021/keylime-security-software-deployed-ibm-cloud-0727, 2021.

[25] T. T. C. Group, "TPM 2.0 Library," https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2004.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *13th USENIX Security Symposium (USENIX Security)*. San Diego, CA, USA: USENIX Association, Aug. 2004, pp. 223–238.

[27] Linux, "TPM Quote Tools," https://linux.die.net/man/8/tpm_quote_tools, 2024.

[28] K. Developers, "Overview of Keylime — Keylime Documentation 7.11.0 documentation," https://keylime.readthedocs.io/en/latest/design/overview.html, 2024.

[29] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks," in *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing (STC)*. Chicago, IL, USA: ACM, 2009, p. 49–54.

[30] E. Shi, A. Perrig, and L. Van Doorn, "BIND: A Fine-Grained Attestation Service for Secure Distributed Systems," in *2005 IEEE Symposium on Security and Privacy (S&P)*. Oakland, CA, USA: IEEE, 2005, pp. 154–168.

[31] I. Pedone, D. Canavese, and A. Lioy, *Chapter 26 Trusted Computing Technology and Proposals for Resolving Cloud Computing Security Problems*. Sebastopol, CA: CRC Press, November 2020.

[32] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud," in *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*. Los Angeles, CA, USA: ACM, Dec. 2016, pp. 65–77.

[33] D. G. Berbecaru and S. Sisinni, "Counteracting software integrity attacks on IoT devices with remote attestation: a prototype," in *2022 26th International Conference on System Theory, Control and Computing (ICSTCC)*. Sinaia, Romania: IEEE, Oct. 2022, pp. 380–385.

[34] Kylie Foy, "Lincoln Laboratory's cloud-security software is released into Red Hat Enterprise Linux | MIT Lincoln Laboratory," https://www.ll.mit.edu/news/lincoln-laboratorys-cloud-security-software-released-red-hat-enterprise-linux, Dec. 2022.

[35] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified Hardware/Software Co-Design for remote attestation," in *28th USENIX Security Symposium (USENIX Security)*. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 1429–1446.

[36] E. Bravi, D. G. Berbecaru, and A. Lioy, "A Flexible Trust Manager for Remote Attestation in Heterogeneous Critical Infrastructures," in *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Napoli, Italy: IEEE, Dec. 2023, pp. 91–98.

[37] J. Pecholt and S. Wessel, "CoCoTPM: Trusted Platform Modules for Virtual Machines in Confidential Computing Environments," in *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*. Austin, TX, USA: ACM, Dec. 2022, pp. 989–998.

[38] A. K. Simpson, N. Schear, and T. Moyer, "Runtime integrity measurement and enforcement with automated whitelist generation," in *Proc. of ACSAC (Poster)*. Melbourne, Australia: IEEE, 2014.

[39] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. Lisbon, Portugal: IEEE, 2009, pp. 115–124.

[40] N. Santos, K. P. Gummadi, R. Rodrigues *et al.*, "Towards trusted cloud computing." *HotCloud*, vol. 9, no. 9, p. 3, 2009.

[41] N. Paladi, C. Gehrmann, and A. Michalas, "Providing user security guarantees in public infrastructure clouds," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 405–419, 2017.

[42] M. Eckel, D. R. George, B. Grohmann, and C. Krauß, "Remote attestation with constrained disclosure," in *Proceedings of the 39th Annual Computer Security Applications Conference (ACSAC)*. Austin, TX, USA: ACM, 2023, p. 718–731.
[43] J. G. Beekman, J. L. Manferdelli, and D. Wagner, "Attestation transparency: Building secure internet services for legacy clients," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*. Xi'an, China: ACM, 2016, p. 687–698.
[44] V. Narayanan, C. Carvalho, A. Ruocco, G. Almasi, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev, "Remote attestation of confidential vms using ephemeral vtpms," in *Proceedings of the 39th Annual Computer Security Applications Conference (ACSAC)*. Austin, TX, USA: ACM, 2023, p. 732–743.
[45] A. Whitehouse, "Ubuntu Explained: How to ensure security and stability in cloud instances—part 2," https://ubuntu.com/blog/ubuntu-updates-best-practices-for-updating-your-instance, November 2023.
[46] SNAP, "Introduction to Snaps," https://ubuntu.com/core/services/guide/snaps-intro, 2024.
[47] ——, "Installing snap on Ubuntu | Snapcraft documentation," https://snapcraft.io/docs/installing-snap-on-ubuntu, 2024.
[48] J. Jellyfish, "Ubuntu 22.04.4 LTS (Jammy Jellyfish) — Index of /ubuntu/dists/jammy," http://archive.ubuntu.com/ubuntu/dists/jammy/, 2022.
[49] C. Ubuntu, "Ubuntu kernels from Canonical | Ubuntu documentation," https://ubuntu.com/kernel, 2024.
[50] M. Ruffin, C. Wang, G. Almasi, A. A. Adeyabo, H. Franke, and G. Wang, "Supplementary Materials," https://tinyurl.com/3u54yypa, 2025.
[51] "Script execution control (was O_MAYEXEC) [LWN.net]," Dec. 2024, [Online; accessed 19. Feb. 2025]. [Online]. Available: https://lwn.net/Articles/1001173
[52] OSTree, "Operating systems and distributions using ostree," https://ostreedev.github.io/ostree/, 2024.
[53] msmbaldwin, "Azure Attestation overview," https://learn.microsoft.com/en-us/azure/attestation/overview, Aug. 2024.