Hindawi
Wireless Communications and Mobile Computing
Volume 2020, Article ID 8866996, 12 pages
https://doi.org/10.1155/2020/8866996

WILEY | Hindawi

Research Article

SLR-SELinux: Enhancing the Security Footstone of SEAndroid with Security Label Randomization

Yan Ding, Pan Dong, Zhipeng Li, Yusong Tan, Chenlin Huang, Lifeng Wei, and Yudan Zuo

School of Computer Science, National University of Defence Technology, Changsha 410073, China

Correspondence should be addressed to Pan Dong; pandong@nudt.edu.cn

Received 14 July 2020; Revised 3 September 2020; Accepted 4 October 2020; Published 26 October 2020

Academic Editor: Ashok Kumar Das

The root privilege escalation attack is extremely destructive to the security of the Android system. SEAndroid implements mandatory access control to the system through the SELinux security policy at the kernel mode, making the general root privilege escalation attacks unenforceable. However, malicious attackers can exploit the Linux kernel vulnerability of privilege escalation to modify the SELinux security labels of the process arbitrarily to obtain the desired permissions and undermine system security. Therefore, investigating the protection method of the security labels in the SELinux kernel is urgent. And the impact on the existing security configuration of the system must also be reduced. This paper proposes an optimization scheme of the SELinux mechanism based on security label randomization to solve the aforementioned problem. At the system runtime, the system randomizes the mapping of the security labels inside and outside the kernel to protect the privileged security labels of the system from illegal obtainment and tampering by attackers. This method is transparent to users; therefore, users do not need to modify the existing system security configuration. A tamper-proof detection method of SELinux security label is also proposed to further improve the security of the method. It detects and corrects the malicious tampering behaviors of the security label in the critical process of the system timely. The above methods are implemented in the Linux system, and the effectiveness of security defense is proven through theoretical analysis and experimental verification. Numerous experiments show that the effect of this method on system performance is less than 1%, and the success probability of root privilege escalation attack is less than 10^{-9}.

1. Introduction

With the widespread application of the Android system, an increasing amount of sensitive information is processed by the system, and additional attention is provided to the system security [1, 2]. Numerous forms of attacks against the Android system exist; among which, the root privilege escalation attack enables the attacker to have "supreme" permission in the system and arbitrarily processes the system resource, causing remarkable damage to the system [3]. SEAndroid mechanism based on SELinux can effectively prevent attackers from gaining root privilege. Although multiple levels of security measures are currently implemented in Android, including app permission and middleware MAC (Mandatory Access Control), SEAndroid achieves the strongest defense effect of access control on the kernel level. It divides the privileges of the system into different "types" and specifies the security permissions to the legitimate processes. Thus, even if the attacker modifies the owner of a process as root, the process still cannot bypass the security checks of SELinux, by which the root privilege escalation is effectively prevented.

However, through the buffer overflow vulnerability-based attack method proposed in this paper, the security label of the targeted process could be maliciously modified into arbitrary value. The security label is one of the key factors of the SELinux mechanism, and all security decisions are made on the basis of the security labels of subjects (processes) and objects (files). If the privileged security labels have been achieved, the permission checks are successfully bypassed and then the root privilege is also escalated. Therefore, protecting the confidentiality and integrity of the security labels of SELinux is a key problem in the effective protection of the system resources and upper-level applications.

Solving this problem faces several challenges. First, since the configuration policy of Linux is open to all users, the specified values of privileged security labels must be protected from illegal acquisition and use by attackers. And the integrity of security labels must be timely detected and the right values must be recovered as soon as possible when the attack succeeds. Second, SELinux and SEAndroid have large-scale security configuration rules [4], which are all configured on the basis of security labels of the system subjects and objects. The protection of security labels of SELinux should not affect the existing security policy configuration, that is to say, the protection should be transparent to users. Moreover, the performance must be considered while improving security. The MAC detection of SELinux, which is implemented in the LSM framework, checks every system call and other system operations. Thus, implementing the lightweight protection mechanism is necessary.

To address these challenges, this paper proposes a dynamic security policy named SLR-SELinux to achieve the confidentiality and integrity protection of security labels. This method divides each SELinux security label into two parts: out-of-kernel and in-kernel ones. The out-of-kernel label is used in the configuration of access control rules, which is consistent with the existing system security policy configuration. The in-kernel label participates in the access control decisions at the kernel level. The corresponding relationship between the two labels is a random mapping, which makes attackers hardly obtain the specified target labels. A tamper-resistant detection mechanism of security labels at the kernel level is also proposed to improve the recoverability of security policy. The integrity check of the process security labels is deployed on the key execution path of the system. Therefore, the illegal modification of the security labels can be timely detected and recovered.

The major contributions of this paper are summarized as follows.

(1) An attack method of tampering SELinux security label is proposed based on the Linux kernel privilege escalation vulnerability. Experiments have proven the effectiveness of this method, and the privileges of the root are successfully obtained

(2) A SLR-SELinux security policy model is proposed based on the security label randomization mapping between the labels inside and outside the kernel. The framework is designed at the Linux kernel. And a fined-grained randomization strategy named full-randomization strategy is proposed, in which the random seed is achieved based on SRAM PUF (Physical Unclonable Function), and the random allocation of security labels is accelerated by the Bloom filter technique

(3) A tamper-proof checking method is proposed for the integrity protection of security labels in the kernel. The integrity detection is deployed on the key access path of the system, and the tampered labels could be recovered as soon as possible

(4) The above technologies are evaluated on the prototype system, and the effects are proved through theoretical proof and numerous experiments

The paper is organized as follows. Section 2 presents a review of related literature. Section 3 discusses the root exploitation method for tampering the security labels of SELinux. Section 4 introduces a system model of enhanced SELinux with randomized security labels. Section 5 indicates the tamper-proof checking method on security labels in the kernel. Section 6 presents the theoretical analysis of the security effect of the current research. Section 7 introduces the experimental evaluation. Section 8 provides the conclusion and suggestions for future studies.

2. Related Work

The system security problem in Android has received considerable attention in both academic and industrial fields due to its open-source feature and wide application. The architecture of the Android software stack can be divided into the Linux kernel, the Android middleware, and the application levels from the bottom-up. Security researches at the middleware level mainly focus on the security issues introduced by the Android local library, the operating environment, and the application architecture [5]. For various applications of the Android system, the permission of the applications is mainly implemented through the permission system [6], complying with the "least privilege principle" authorization management. The system permissions are divided into three different kinds: owner, root, and application. However, the security of the middleware level only solves the security problems of a certain level of the Android system, and the permission system has problems of coarse granularity of security management [7, 8] and overprivileged [9]. These security mechanisms mainly improve the system security through the development of the Android middleware level, and any security control implemented through middleware ultimately depends on the control of the kernel level. If an attacker directly attacks the system kernel, then the upper-level security mechanism can be bypassed.

Researchers proposed to introduce SELinux into the Android to solve this problem; SELinux strengthened the security of the underlying Android operating system [10, 11]. SELinux, a Linux security enhancement module proposed by NSA, provides the Linux system with MAC based on the type enforcement security policy. This policy is known for its fine-grained access control and strong security policy. The SEAndroid security module has been introduced since Android 4.3. With the advancement of SEAndroid security policy research, an increasing number of Android functions are protected by SEAndroid. Therefore, SEAndroid security research has also received considerable attention. Currently, the study on SEAndroid security can be divided into the analysis [12, 13], generation, and refinement [14, 15] of the security policy.

Many security problems are not unique to the Android system but are inherited from the underlying Linux system because the Android system is an extension of the Linux

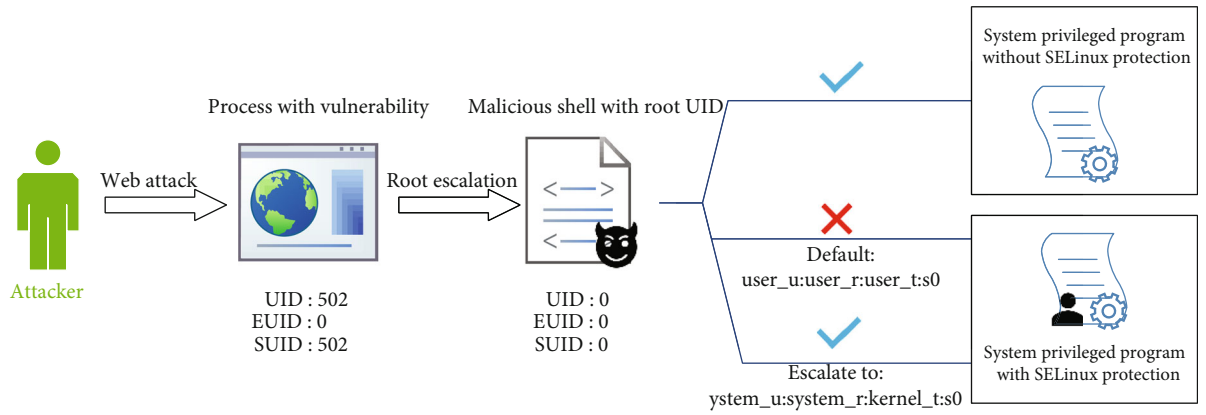

Figure 1: The root privilege escalation attack on SELinux.

system. Therefore, the system security of the underlying SELinux mechanism is crucial to the security of the entire system. SELinux has been researched for years. Most of the studies focused on the policy configuration security of SELinux, such as SELinux policy analysis and verification [16–19], policy comparison [20], policy visualization [21], and policy information flow integrity measurement [22, 23]. However, we found that the attackers could use the privilege escalation vulnerability of the system to bypass the SELinux mechanism. Therefore, this paper focuses on the security enhancement of the SELinux mechanism. Through the randomization and integrity checking of security labels, the security permissions of a process cannot be maliciously tampered, and meanwhile, there is no influence on the existing configuration of system security policy.

3. Threat Model and Attack Method

3.1. Threat Model. The typical procedure of penetration attack to the computer system could be divided into three steps: (1) remotely web attack to achieve the permission as an ordinary user, (2) root escalation attack to achieve the permission as the root, and (3) accessing and destruction on the system resources. With the protection of SELinux, even if the attacker succeeds in step 2, the promotion of continued attacks in step 3 will be prevented.

As shown in Figure 1, for a process of which the *uid* is 502, if the attacker only modifies the user ID and group ID of the process, the ultimate privileged control over the system cannot be obtained (e.g., modifying the password of the root), even though the user identification of the process has also been elevated to the root. Thus, the security label is the key point in SELinux. However, if the security label of the process is modified to the privileged one, the corresponding permission over the system can only be obtained by the attacker and then the password of the root can be modified.

This paper is focused on defensing the root privilege escalation attack on SELinux in the above threat model. An empirical attack evidence is implemented firstly, providing the basis of the follow-up research.

3.2. Root Privilege Escalation Attack on SELinux. SELinux is a MAC module built on the LSM framework [24]. The Linux kernel queries SELinux before each system call to determine whether the process is authorized to perform the requested operation. With SELinux, the management of privileges is completely different from that of the standard Linux system. The privileges of a process depend on its security context instead of the user labels. Therefore, the privileges are confined even if the attacker escalates the user identity to the root user. Thus, the SELinux can reduce the threats of privilege escalation attacks.

The security labels of a process are saved in the process credentials in the Linux kernel. The structure of process credentials is named as *cred*. The main information concerned with the process permissions in *cred* includes user/group ID and the set of capabilities. If SELinux is enabled, then the structure also includes the security label, which represents the security attributes in the process.

Figure 2 shows that the total kernel space size of a process is 8 KB, and the structure of the thread descriptor, which is named as *thread_info*, shares the same memory region with the kernel stack of the process. *thread_info* is stored at the bottom of the shared memory region. A pointer *task*, which indicates the process descriptor *task_struct*, is found in *thread_info*. Moreover, *task_struct* includes pointers *cred* and *real_cred*, indicating the *cred* structure. All the user and group IDs are saved in the *cred* structure. If SELinux is enabled, then the pointer *security* indicates the structure *task_security_struct*, which includes sids associated with the process.

One of the typical methods used to escalate the privileges of the process is modifying the user/group IDs to 0 (uid of root) saved in the *cred*. The procedure comprises the following three steps.

Step 1. Obtain the memory address of *cred.*

The base address of the shared memory region of kernel stack and *thread_info* is 8 KB aligned. Therefore, we can obtain the address of *thread_info* by resetting the lower 13 binary bits of the address of any variable in the kernel stack. Then, the address of *task_struct* with the pointer *task* can also be acquired.

We also obtain the address of *cred* based on the features of *task_struct*. In the *task_struct*, the pointer *real_cred* is

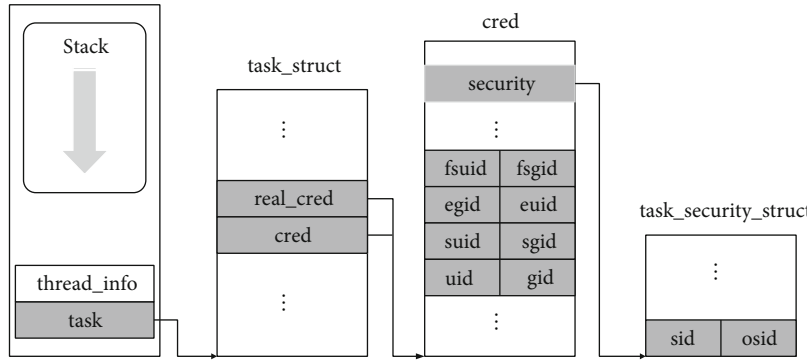

Figure 2: The structure of process credentials.

similar to *cred*. According to this feature, we can locate the address of *cred* and *real_cred* by searching two similar pointers in *task_struct*. After finding two similar 64 binary bits in *task_struct* and the value of the identified 64 binary bits is in the range of kernel space addresses, then we can regard these bits as the correct address of *cred*.

Step 2. Obtain the copy of *cred* and modify the data on process privileges.

First, we must create a data structure with a similar layout to *cred*. Then, we copy the data in *cred* into the new created data structure and modify the data in it, including the user and group IDs.

We must also modify *sid* and *osid* in the *task_security_struct* for SELinux. By changing the values of *sid* and *osid* to 1, we can modify the security context of the process to *system_u:system_r:kernel_t:s0*, which is unconfined in SELinux.

Step 3. Cover the original data in *cred* with the modified data in the copy of *cred.*

After modification, the values of all user and group IDs in the copy are 0, and the *osid* and *sid* are 1 in the *task_security_struct*. The user identification of the process is elevated from normal user to the root when the original data in *cred* are covered with those in the copy, and the security context of the process is also modified.

As shown in Figure 1, a user whose uid is 502 finally obtains the privileged label *system_u:system_r:kernel_t:s0* and performs the system management successfully in our experiment.

4. Security Label Randomization of SELinux

In the kernel space, the traditional allocation of in-kernel security label (sid) is sequential starting at 1, and the mapping between out-of-kernel security labels (security contexts) and in-kernel security labels (sids) is fixed in all SELinux distributions. Thus, the attackers can easily predict the sid for the necessary security context. We propose a randomized allocation of sids to solve this problem and enhance the uncertainty of relations between sids and security contexts. Therefore, the attackers cannot accurately predict the sid of the specific security context, which increases the difficulty of kernel privilege escalation attacks.

4.1. Definitions of SLR-SELinux Policy. A SELinux policy comprises two parts. The first part is label mapping, which assigns security labels to concrete subjects (or objects) in the operating system. Traditionally, subject and object labels are, respectively, called *domain* and *type*. The second part involves a set of rules that define which domain of subjects can access which class and type of objects with a set of permissions. The definition of the SELinux policy is defined as follows:

Definition 1. (SELinux policy). A policy is $P = (L_s, L_o, M, S, O, R)$, where L_s and L_o are the set of security labels of subjects and objects, respectively; $M : L_S \cup L_O \rightarrow S \cup O$ is a mapping that assigns security labels to concrete subjects S and objects O; and $R = \{r \mid <l_s, l_o> \rightarrow \{\text{allowed operations}\}\}$ is the set of policy allowed rules.

In SLR-SELinux, a random mapping of the security label is introduced into the policy. This mapping divides a security label into two parts according to its usage space: *out-of-kernel* and *in-kernel* labels. The out-of-kernel security label, which has a fixed representation and is saved in the file system, is used for the policy configuration in the application level. By contrast, the in-kernel label, which has a dynamically generated representation on the boot time, is used for the access control decision in the kernel space. The random mapping function is defined as follows:

Definition 2. (random mapping of security labels). A mapping is $F : L_S \cup L_O \rightarrow L_S{}' \cup L_O{}'$, where L_s, L_o are the set of out-of-kernel labels of subjects and objects, respectively; $L_s{}'$, $L_o{}'$ are the set of in-kernel security labels of subjects and objects, respectively. The mapping assigns a random in-kernel label to each out-of-kernel label arbitrarily.

The definition of SLR-SELinux is as follows:

Definition 3. (SLR-SELinux policy). A policy is $P^{'} = (L_s, L_o, L_s^{'}, L_o^{'}, M, M^{'}, S, O, R, R^{'}, F)$, where L_s, L_o are the set of out-of-kernel labels of subjects and objects; $M : L_S \cup L_O \rightarrow S \cup O$

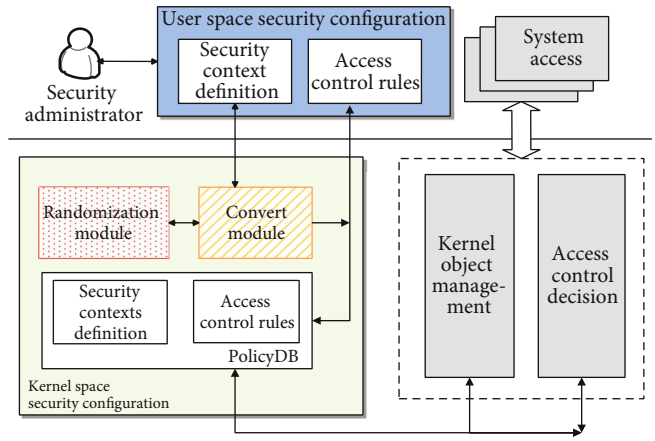

Figure 3: Framework design of SLR-SELinux.

is a mapping that assigns out-of-kernel security labels to concrete subjects S and objects O, $R = \{r \mid <l_s, l_o> \rightarrow \{\text{allowed operations}\}\}$ is the set of allowed policy rules defined by out-of-kernel security labels; F is the random mapping between the out-of-kernel and in-kernel security labels; M' : $L_S{}' \cup L_O{}' \rightarrow S \cup O$ is the mapping that assigns in-kernel security labels to concrete subjects S and objects O; $R^{\cdot} = \{r^{\cdot} \mid <l_s{}', l_o{}' > \rightarrow \{\text{allowed operations}\}\}$ is the set of policies used by access control decision in kernel space.

The in-kernel labels corresponding to one out-of-kernel label will be different in every system booting because the mapping between the two types of labels is a random function. This difference complicates the speculation of the right representation of the security label inside the kernel by the attacker.

4.2. Framework Design. Figure 3 shows the SLR-SELinux framework with the randomized allocation on security labels.

Security configuration in user space includes the definition of security contexts and access control rules. The configuration is loaded into *policydb* in kernel space during the system booting process. The subjects and objects in kernel must be labeled with the specified sids according to the security configuration before they are accessed or used for the first time. Therefore, a module named as *convert module* is added into SLR-SELinux, to allocate random sid for the security contexts.

When a kernel object requests a security context, SLR-SELinux first determines the security context according to the *security context definition* and checks the *sidtab* (sid table containing registered security contexts indexed by the allocated sids) to determine whether the security context has been registered. If the security context exists in the sidtab, then the sid can be directly obtained. Otherwise, the *convert module* allocates a random sid for the security context via *randomization module*.

The *randomization module* is responsible for generating a random value and returning it to the *convert module*. The *convert module* then checks whether the sid to be allocated conflicts with all the already allocated sids. If conflicting, then another random value will be required until there is no conflict.

A function, *generate_random_sid*(), is designed in the randomization module to generate a random sid. Since the sid is descripted as an integer in the Linux kernel, the maximum possible value is 2^{32}. Therefore, Mersenne Twister (MT19937-32) [25] is used in this function to generate a random number. As a kind of pseudorandom number generator, MT19937-32 is well-known for a remarkably long cycle period of 2^{19937}-1. MT19937 has the characteristics of 623 distributed to 32-bit accuracy. The performance of MT19937 on K-distributed to v-bit accuracy reached the theoretical maximum of the evaluation standard considering that $\lfloor 19937/32 \rfloor = 623$. Moreover, the speed of MT19937-32 in generating random numbers is generally faster than that of other pseudorandom generating algorithms because its primary operations are *bit or*, *bit and*, and *shift*.

The key factor affecting the random sequence is the random seed. The same random seed will create the same random sequence, so the random seed must ensure the randomness and confidentiality [26]. The random seed is obtained based on SRAM PUF. SRAM PUF is a technology in which SRAM is evaluated by a stimulus (challenge), which provides a noisy response based on the manufacturing process variations of the SRAM. The noisy response can only be obtained during the normal operation of SRAM. Thus, the noisy response can be turned into stable data, which can serve as random seeds with high confidentiality, by using fuzzy extractors. However, the extracted seed will always be the same one. To solve this problem, the system time of the first calling in the *randomization module* is obtained and made the *xor* operation with PUF data to act as the random seed.

4.3. Full-Randomization Strategy. Different randomization grains of security labels will affect the security and performance of the operating system differently. To achieve the greatest defense effect, the full-randomization strategy is proposed. Each sid is allocated randomly by separately calling the function *generate_random_sid*(). The main problem to be solved in this strategy is the conflict of the sid to be

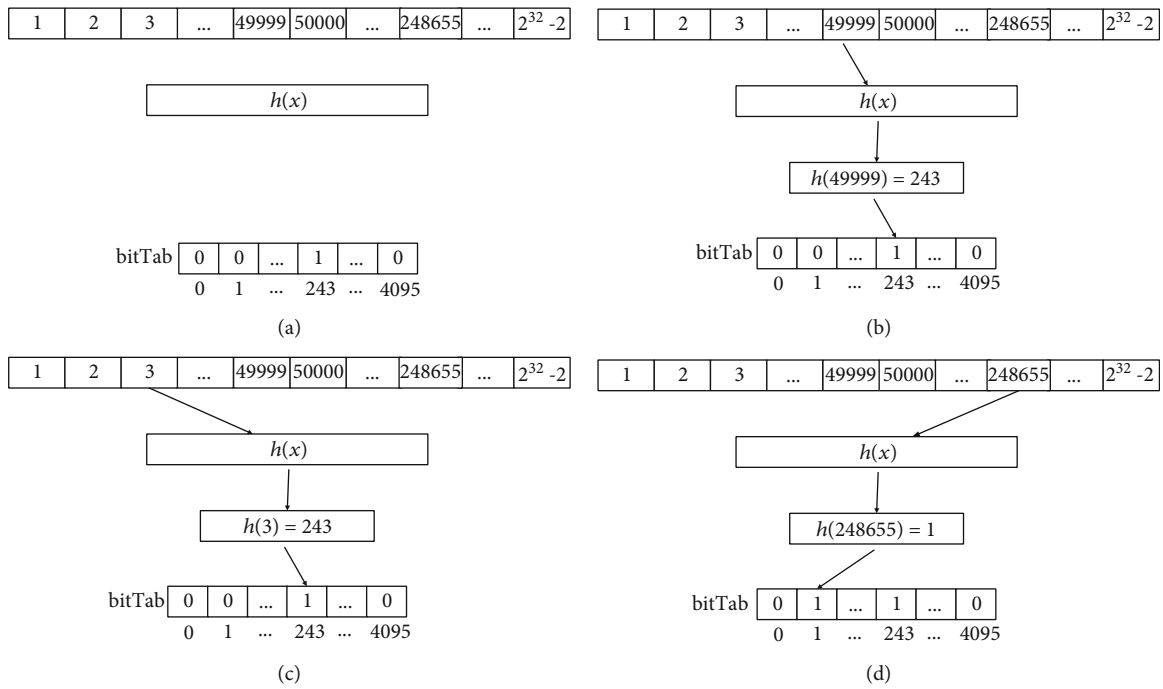

Figure 4: Bloom filter principle.

allocated with the already allocated ones. Conflict checks must be conducted for every sid to be allocated.

SLR-SELinux first checks whether the security context is registered into the sidtab before sid allocation. If not registered, then a new sid must be allocated. The *convert module* needs to check whether the random value generated by *generate_random_sid*() is conflicted with the already allocated sids. If conflicted, then another new sid needs to be allocated.

We use *Bloom filter* [27] to check the conflict and facilitate an efficient insertion. As shown in Figure 4, the Bloom filter comprises three parts: the original space (in the random number generator space, size 2^{32}-1), the hash function $h(x)$, and an all-zero bit array (taking the 4096 size as an example). First, every bit in the bitTab is set to 0 (Figure 4(a)). Then, a random value of 49999 is obtained and $h(49999) = 243$. Because bitTab[243] is 0, the module will set bitTab[243] to 1 and return 49999 as a sid (Figure 4(b)). Another process requires the allocation of a sid (assuming it is 3, $h(3) = 243$). But bitTab[243] is 1 (Figure 4(c)). Thus, the module will refuse the random value 3 and require another random value (e.g., 248655). Finally, a random value of "248655" is generated. Since $h(248655) = 1$ and bitTab [1] is 0, the random value "248655" is returned as a new sid and bitTab [1] is set to 1(Figure 4(d)).

The random sid generation increases the uncertainty in the corresponding relationship between the sid and security context. However, the sid may be an arbitrary value between 1 and 2^{32}-2; that is, the probability of successfully guessing the sid is only $1/(2^{32}-2)$. It is proven that SELinux will firstly examine whether the sid is already defined before the access control rules check. If the sid is undefined, the process with this sid will be crashed. However, only crashing the user's process is not enough to defense the brute force attack. So, we add an alert mechanism into the system to notify the administrator about this situation. Moreover, not only the undefined sids will trigger the alert. If the process's sid found that its owner should be the object of the system, such as file or socket, the alert is also triggered.

5. Tamper-Proof Checking on Security Label

5.1. Definition of Method. The randomization of security labels mainly protects the confidentiality of the privileged security labels so that the attackers could not obtain the desired targeted security attributes. However, once the security label has already been modified by attackers, it is urgent to detect the tamper behavior and recover the sid to the legal one as soon as possible. A method of tamper-proof checking on the security label is proposed in this paper.

In the method, a mapping table called *pid_sid_table* and a set of checking hooks are defined in the operating system kernel, as shown in Figure 5.

The *pid_sid_table* records the valid security label of each process running in the system. The table entry is saved in the form of <*pid*, *sid*>, indicating that *sid* is the valid security label of the process with *pid*.

The checking hooks are inserted in the kernel on the key procedure of the process management and accessing behaviors. When the process is created, the item of <*pid*, *sid*> is inserted into the mapping table as a new node; when the process is revoked, the node is deleted; when the security label of the process is changed through legal operations, the node is updated.

When the process accesses the resource of the system, the validity of the security label of this parent process is checked. If the label is inconsistent with the one in the mapping table,

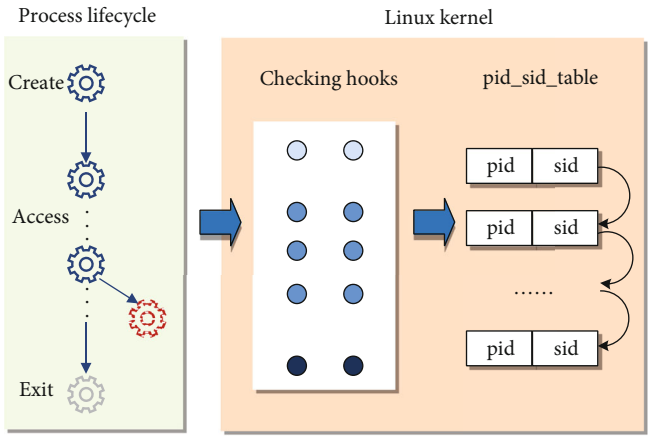

Figure 5: The method of tamper-proof checking on security label.

the security label of the process will be recovered to the valid value. When the *execve* operation is performed, the label of the parent process will be checked firstly and then the parent's valid sid will be set as the default label of the child process. For the possible performance overhead, the calling of checking hooks should be carefully chosen according to the real scenario.

5.2. Implementation in Linux. Based on the analysis of the process lifecycle in Linux kernel, the management functions of *pid_sid_table* are added at the following important time points.

(1) *selinux_pst_insert*(). Insert a node into the *pid_sid_table* table. All processes in the Linux system are created by the function *do_fork*(). And when a process executes a new program, the permission credentials *cred* of the current process will be modified through the function *commit*_creds(). Hence, we choose to call the function *selinux_pst_insert*() during the processing of these two functions.

(2) *selinux_pst_remove*(). Delete a node from the *pid_sid_table* table. Process revocation is conducted through the function *do_exit* (). Thus, the function *selinux_pst_remove* () is called during the processing of this function.

(3) *selinux_pst_check*(). Check whether the *sid* of the current process has been illegally modified, by detecting whether the security label of the process is consistent with the *sid* stored in the *pid_sid_table*. If illegally modified, the security label will be recovered. Tamper-proof detection must be performed before the system executes the new program. Hence, the detection is deployed when the process commits the changes to the *cred* of the current process. This procedure is also completed in the function *commit_creds*().

The calling relationships of corresponding functions at different kernel levels are shown in Figure 6.

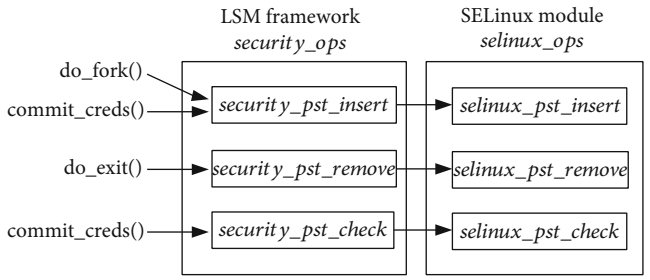

Figure 6: Modification of kernel functions.

6. Security Analysis

6.1. Security Proof

Theorem 4. *(equivalence with SELinux). The configuration of access control rules in SLR-SELinux is equivalent with the rules in SELinux, so the defense effect of SELinux could be also achieved in SLR-SELinux.*

Proof. Because of the one-to-one mapping feature of the random mapping function $F, \forall l \in L_S \cup L_O$ in P_{SELinux}, $\exists l' \in L_S{}' \cup L_O{}'$ in $P_{\text{SLR-SELinux}}$, then $\forall r = <l_s, l_o> \rightarrow \{\text{allowed operations}\} \in R$ in P_{SELinux}, $\exists r' = <l_s{}', l_o{}'> \rightarrow \{\text{allowed operations}\} \in R'$ in $P_{\text{SLR-SELinux}}$ and vice versa. Therefore, $P_{SELinux} \Longleftrightarrow P_{SLR-SELinux}$. SLR-SELinux could achieve the same effect on access control as SELinux.

Theorem 5. *(recoverability of policy). The security policy could be recovered to the valid status after being maliciously modified by the attacker.*

Proof. For a process p_i, there is a table entry $<p_i, l_i>$ in the *pid_sid_table*, where l_i is the valid value of the security label owned by p_i. When the attack succeeds, the security label of p_i will be maliciously modified to the invalid value of $l_i{}'$. Once an operation $o_j \in O_c$ is made by p_i, where O_c is the set of checked operations, the checking hook function of o_j will be called. Then, it will be found that the current security label $l_i' \neq l_i$, the tampering is discovered, and the security label will be recovered to l_i. The policy is returned to the valid status.

Therefore, SLR-SELinux could complete the function of mandatory access control and separation among security domains as same as SELinux, and thus, the attack about authority escalation, such as that malicious application accesses unauthorized data, could be defended. For example, even if an attack on the web service is completed successfully, the victim process can only access the system resource permitted by SLR-SELinux and the destruction effect will be limited to the minimum range. Not only that, SLR-SELinux's random allocation on security labels could defend the root privilege escalation based on buffer overflow vulnerability and the defense effect will be analyzed in the next section. Even if the label is modified to the targeted value by coincidence, the tamper-proof checking scheme will discover and recover it as soon as possible.

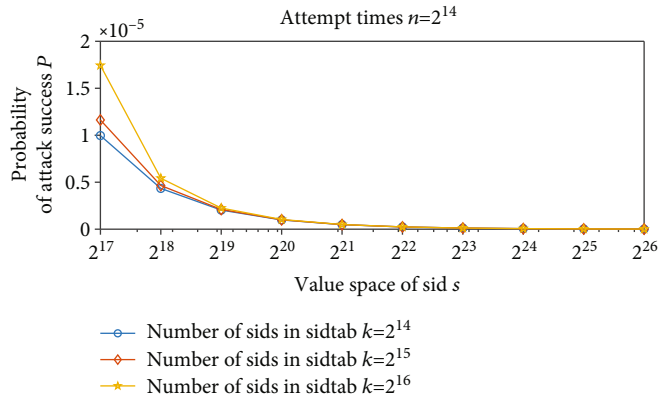

Figure 7: Probability of attack success vs. value space of sid.

6.2. Defence Effect Analysis. Since the in-kernel security label (sid) is randomly allocated in the proposed scheme, the root privilege escalation attack succeeds only if the correct sid of the targeted security label is guessed out. To achieve this goal, the attacker could exploit the brute force attack, that is to say, the attacker guesses a different sid value one time and then tampers the victim process with that value, trying to pass the permission check of SELinux.

To defend the guessing attack, an alert mechanism is added to the system. In the implementation of permission check hooks, it is examined firstly whether the sid has been registered in the sidtab of SELinux. If the sid is not registered in the sidtab, which means it is an invalid value, then the alert will be triggered and the system will be restarted. Once the system restarts, all sids will be reallocated and the mapping between in-kernel and out-of-kernel labels will be changed. If the guessed sid has been registered in the sidtab by coincidence, then the attacker can repeat this attack behavior. If the attacker identifies the targeted sid without triggering the system alert, the attack successes.

To evaluate the defense effect, the selected evaluation index is the probability of attack success P. P is defined as the probability that the attacker exploits the brute force attack successfully to obtain the right sid without triggering the system alert. With the full-randomization strategy, P is mainly related to the following three factors: (1) the value space of sid s, (2) the number of registered sids k, and (3) the attempt times of the attacker n. s is the range of possible values of sid. k is the number of legally allocated sids in the sidtab. n is the times the attacker has tried without triggering the alert. Apparently, $n \le k$. Otherwise, the alert must be triggered.

Therefore, the probability of attack success in full-randomization strategy is shown as follows.

$$P = \frac{1}{s} + \frac{k-1}{s} \times \frac{1}{s-1} + \frac{k-1}{s} \times \frac{k-2}{s-1} \times \frac{1}{s-2} + \cdots + \frac{k-1}{s} \times \frac{k-2}{s-1} \times \cdots \times \frac{k-(n-1)}{s-(n-2)} \times \frac{1}{s-(n-1)}, \tag{1}$$

$$P < \frac{1}{s} + \frac{k}{s} \times \frac{1}{s-1} + \frac{k}{s} \times \frac{k}{s} \times \frac{1}{s-2} + \cdots + \frac{k}{s} \times \frac{k}{s} \times \cdots \times \frac{k}{s} \times \frac{1}{s-(n-1)}, \tag{2}$$

$$P < \frac{1}{s-n} \times \left(1 + \frac{k}{s} + \left(\frac{k}{s}\right)^2 + \cdots + \left(\frac{k}{s}\right)^{n-1}\right) \approx \frac{s}{(s-k)(s-n)}. \tag{3}$$

In Figure 7, n is fixed to 2^{14}, and P rapidly declines when s ascends, because it is more difficult to guess the sid in a larger value space. And with the same s, P increases slightly when the number of sids k is getting larger, because it is more difficult to trigger alert when there are more valid sids in the sidtab.

In Figure 8, s is fixed to 2^{16}, and P increases slowly with the attempt times n ascends. It is shown that P almost maintains the same order of magnitude, indicating the fine defense effect.

As shown in Figure 9, P increases as the value of k gets closer to s. The reason is that when the number of sids in the sidtab is close to the value space of sid, the attacker has a greater chance to suppress the alert and could try more values about the targeted sid. Therefore, the proportion of k in s should be as small as possible. Then, the attacker has little chance to attack successfully. Fortunately, P declines quickly with the less proportion of k in s. When the proportion is smaller than 97%, P will be less than 10^{-9}.

In reality, the number of security types in SELinux-Policy(2:2.20140421-9) is about 2^{12} (4096) and the value space of sid is 2^{32}. Assuming that the factor k is 4096 (i.e., 4096 sids have been allocated) and the factor s is 2^{32}, the security of the full-randomization strategy is shown in Table 1.

Clearly, the maximum of the attempt times for the attacker is 4096 because the alert must be triggered if the attacker tries more times. The results show the probability of attack success is low and stable. The value of P is under 10^{-9}, indicating the system is safe enough.

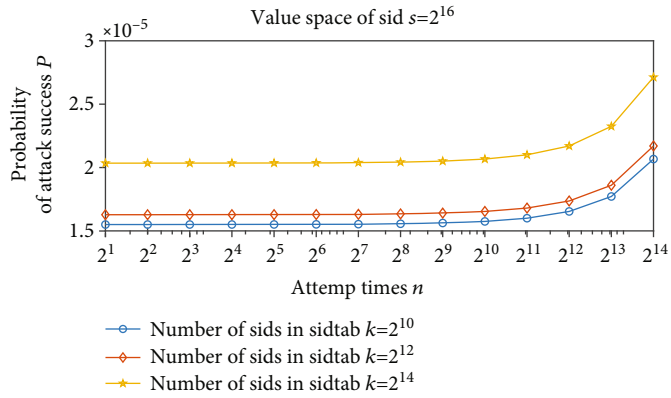

Figure 8: Probability of attack success vs. attempt times.

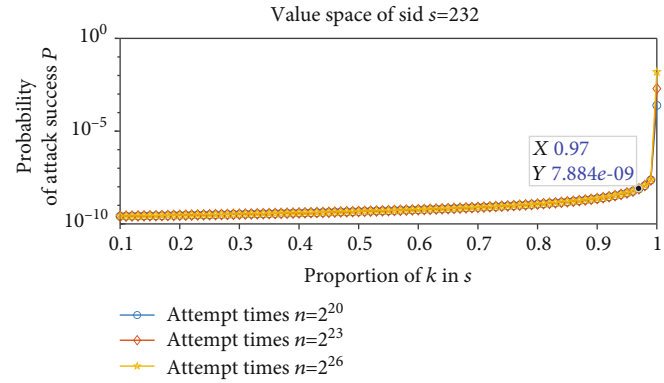

Figure 9: Probability of attack success vs. the proportion of k in s.

Table 1: Probability of attack success in the current SELinux policy.

Attempt times	P in full-randomization strategy (x10^{-9})
1	0.2328
2	0.2328
4	0.2328
8	0.2328
16	0.2328
32	0.2328
64	0.2328
128	0.2328
256	0.2328
512	0.2328
1024	0.2328
2048	0.2328
4096	0.2328

6.3. Comparison with Others. The defense methods of root privilege escalation attack could be divided into three categories, the separation of privilege and the memory protection in user space and in kernel space. The separation of privilege scheme, such as SELinux, is based on the fine-grained control on the root privilege. SLR-SELinux is also designed in this manner.

Table 2: Boot time test.

	Original SELinux	SLR-SELinux
Average time (seconds)	23.66	24.20
Proportion	100%	101.61%

The memory protection methods are based on preventing the execution control flow of the process from jumping into the malicious code injected in the user space. The typical schemes of memory protection in user mode includes compiling protection (StackGuard [28], StackShield [29]), data execution protection (NX [30], ExecShield [31]), and Address Space Layout Randomization (ASLR) [32]. These schemes could only prevent the hijacking of execution flow in user mode and have little defense effect on the exploit of buffer overflow vulnerability in kernel mode. The hardware-based protection methods, including SMAP and SMEP [33] of Intel CPU, prevents the process in kernel-mode from executing the section of data and code in the user space. But attackers could also inject the malicious data into the kernel space. The KASLR, which deploys ASLR in the kernel space, is implemented by the GRSecurity project. But it cannot defense the attack method proposed in this paper, for the relative address is used in our attack. Other academic achievements, such as kRazor [34] and randomization of structures in kernel [35], are limited to large-scale promotion

Table 3: System performance test for Linux (UnixBench).

No.	Test items	Origin SELinux	SLR-SELinux with full-randomization	SLR-SELinux with tamper-proof checking
1	Dhrystone 2 using register variables	8377.5	8378.2	8377.9
2	Double-precision whetstone	2745.1	2745.5	2746.1
3	Excel throughput	4042.7	4004.2	3553.2
4	File Copy 1024 bufsize 2000 maxblocks	2755.4	2794.4	2800.5
5	File copy 256 bufsize 500 maxblocks	1655.6	1664.7	1680.7
6	File Copy 4096 bufsize 8000 maxblocks	5755.8	5856.3	5833.3
7	Pipe throughput	3437.0	3409.5	3429.9
8	Pipe-based context switching	3076.5	3055.1	3065.6
9	Process creation	4574.8	4577.5	3899.3
10	Shell scripts (1 concurrent)	4549.5	4510.8	3944.8
11	Shell scripts (8 concurrent)	4341.2	4320.1	3951.9
12	System call overhead	4366.5	4350.3	4336.1
	System benchmark index score	*3837.0*	*3835.7*	*3535.4*

application for the compatibility with the commercial distribution of Linux.

7. Experiment and Evaluation

We implemented SLR-SELinux based on CentOS 6.2 and performed tests for security protection effect and system performance. The experiments are conducted on 64 bits of CentOS 6.2 (kernel version 2.6.32, processor model of Intel(R) Core(TM) i3-4130 CPU @ 3.40 GHz). The SELinux security policy used is the *targeted* policy.

7.1. Defense Effect Test. We use the vulnerability CVE 2013-2094 on the CentOS 6.2 to test the defense effect. The vulnerability CVE-2013-2094 [32] is in the function "*perf_swevent_init*" from the file *kernel/events/core.c*. The vulnerability comes from the incorrect usage of integer data, which can be utilized to gain root authority by local attackers.

When the system runs without the security label randomization method, the security context of the attack process is *user_u:user_r:user_t:s0*, which has a lower permission in the system. Then, root privilege escalation attacks can be performed on this low-privileged security context by editing security labels to "1," which is the sid of the security context *system_u:system_r:kernel_t:s0*. However, with the security label randomization method, the attack process crashes. The reason is that when the attacker edits *osid* and *sid* to "1," this security label has no corresponding valid security context in the sidtab. Thus, the process cannot return to the user space from kernel space normally, thus leading to a crash.

After improving the tamper-proof detection in SELinux, when the attack process maliciously modifies its security label using root privilege escalation attacks, the tampered security label of the process is detected and recovered. Thus, the attackers cannot break through the security protection of SELinux for the system through kernel privilege escalation attacks.

7.2. System Performance Tests. The system performance tests include boot time and runtime performance tests. The influences of randomized strategies on the system performance are within acceptable limits because SLR-SELinux allocates sids for the security labels only when they are used for the first time.

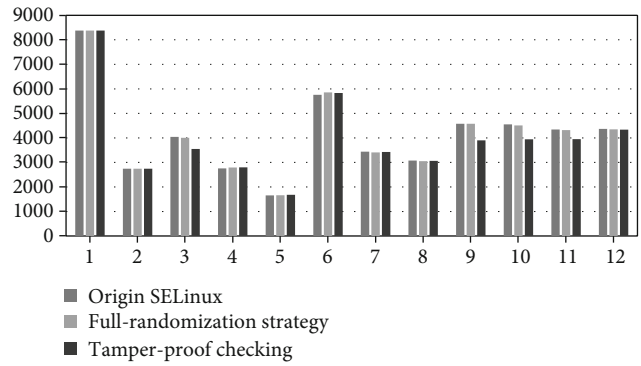

Figure 10: System performance test for Linux (UnixBench).

7.2.1. Boot Time Test. We measured the boot time of the implemented original SELinux and SLR-SELinux. Each item was measured 100 times, and then, the average boot time was obtained.

As shown in Table 2, in contrast with the original SELinux, the boot times after implementation of SLR-SELinux only increase by 1.61%. This finding indicates that the randomized strategy in SLR-SELinux has limited influence on the boot time.

7.2.2. Runtime Performance Test. We tested the runtime performance of the system with UnixBench 5.1.3. Table 3 shows that the results of each test item of UnixBench are near to each other for the three situations: original SELinux, SLR-SELinux with full-randomization, and SLR-SELinux with tamper-proof checking.

Figure 10 intuitively shows the result of the system performance test by UnixBench 5.1.3. The tests were repeated 100 times for every item in the table. The system

benchmark scores provided by UnixBench indicate that full-randomization strategy only have minimal impact on the overall system performance within 1%. For test items of Excel throughput, process creation, and shell scripts, the system performance with tamper-proof checking scheme is diminished due to the frequent creating and canceling processes. Therefore, the user can consider whether the method is used to achieve strong enough protection according to the real requirement.

8. Conclusions

In this paper, a random allocation method of security labels named SLR-SELinux is proposed to enhance the defense capability of SELinux against root privilege escalation attacks. With the randomized strategies, the values of security labels are different after each system reboot. Therefore, the attackers cannot predict the sid for the specific security context accurately, thus increasing the difficulty of root privilege escalation attacks. A tamper-proof detection method of security label is also proposed to further improve the security protection, with which the integrity of the security label is measured in the critical execution path of the system, and the malicious tampering behaviors are detected and corrected timely. The theoretical analysis and experiments show that the method can achieve good defense effect and system performance. We will focus on improving the performance of the tamper-proof detection mechanism in future research.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

We declare that we have no financial and personal relationships with other people or organizations that can inappropriately influence our work; there is no professional or other personal interest of any nature or kind in any product, service, and/or company that could be construed as influencing the position presented in the manuscript entitled.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (grant numbers U19A2060, 61502510) and the National Key Technologies Research and Development Program (China) (grant number 2018YFB0803501).

References

[1] StatCounter, *Android challenges Windows as worlds most popular operating system in terms of internet usage*, 2017, http://gs.statcounter.com/press/android-challenges-windowsas-worlds-most-popular-operating-system.

[2] H. Lockheimer, "Android and security," *Google Mobile Blog*, 2012, http://googlemobile.blogspot.com/2012/02/androidand-security.html.

[3] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: scalable and accurate zero-day android malware detection," *Procceedings of MobiSys*, 2012.

[4] R. Wang, W. Enck, D. Reeves et al., "EASEAndroid: automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning," *USENIX Security*, vol. 15, pp. 351–366, 2015.

[5] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security & Privacy. Magazine*, vol. 7, no. 1, pp. 50–57, 2009.

[6] Y. Zhauniarovich and O. Gadyatskaya, "Small changes, big changes: an updated view on the android permission system," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, Springer, Cham, 2016.

[7] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," *Procceedings of the IEEE Annual Computer Seeurity Applications Conference*, pp. 340–349, 2009.

[8] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in Android," *Procceedings of the ACM Asia Conference on Computer and Communications Security*, vol. 7, pp. 71-72, 2012.

[9] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and Communications Security*, vol. 18, pp. 627–638, New York, NY, USA, 2011.

[10] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing android–powered mobile devices using SELinux," *IEEE Security and Privacy*, vol. 8, no. 3, pp. 36–44, 2010.

[11] S. Smalley and R. Craig, "Security enhanced (SE) Android: bringing flexible MAC to Android," *Proceedings of the Network and Distributed System Security Symp*, vol. 20, pp. 20–38, 2013.

[12] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang, "Analysis of seandroid policies: combining MAC and DAC in android," *Proceedings of the 33rd Annual Computer Security Applications Conference*, pp. 553–565, 2017.

[13] E. Reshetova, F. Bonazzi, T. Nyman, R. Borgaonkar, and N. Asokan, "Characterizing SE Android policies in the wild," *CoRR abs*, vol. 1510, article 05497, 2015.

[14] R. Wang, W. Enck, D. Reeves et al., "EASE android: automatic policy analysis and refinement for security enhanced Android via large-scale semi-supervised learning," *Proceedings of the USENIX Conference on Security Symposium, USENIX Association*, vol. 24, no. 15, pp. 351–366, 2015.

[15] R. Wang, A. M. Azab, W. Enck et al., "SPOKE: scalable knowledge collection and attack surface analysis of access control policy for security enhanced Android," *Proceedings of the 2017 ACM on Asia Conference on Computerand Communications Security*, no. 17, pp. 612–624, 2017.

[16] M. Alam, J.-P. Seifert, Q. Li, and X. Zhang, "Usage control platformization via trustworthy SELinux," *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, vol. 8, pp. 245–248, 2008.

[17] B. Hicks, S. Rueda, and L. S. Clair, "A logical specification and analysis for SELinux MLS policy," *ACM Transactions on Information and System Security*, vol. 13, no. 3, pp. 1–31, 2010.

[18] T. Jaeger, R. Sailer, and X. Zhang, "Resolvingconstraint conflicts," in *SACMAT*, vol. 4, pp. 105–114, ACM Press, New York, USA, 2004.

[19] A. Sasturkar, S. D. Stoller, C. R. Ramakrishnan, C. Science, and S. Brook, "Policy analysis for administrative role based access control," *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006.

[20] H. Chen, N. Li, and Z. Mao, "Analyzing and comparing the protection quality of security enhanced operating systems," *NDSS*, vol. 9, 2009.

[21] W. Xu, M. Shehab, and G.-J. J. Ahn, "Visualizationbased policy analysis: case study in SELinux," *Proceedings of the ACM Symposium on Accesscontrol models and technologies*, vol. 13, pp. 165–174, 2008.

[22] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA:policy-reduced integrity measurement architecture," *SACMAT*, vol. 6, pp. 19–28, 2006.

[23] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger, "Integrity walls: finding attack surfaces from mandatory access control policies," *ASIACCS*, vol. 12, pp. 75-76, 2012.

[24] G. Vinod, J. Trent, and J. Somesh, "Automatic placement of authorization hooks in the Linux security modules framework," *Proceedings the ACM conference on Computer and Communications Security*, vol. 12, no. 5, pp. 330–339, 2005.

[25] M. Matsumoto and T. Nishimura, "Mersenne twister," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[26] S. J. Zhao, Q. Y. Zhang, G. Y. Hu, Y. Qin, and D. G. Feng, "Providing root of trust for ARM TrustZone using on-chip SRAM," *Proceedings of the 4th Int'l Workshop on Trustworthy Embedded Devices*, 2014.

[27] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[28] P. D. Varma and V. Radha, "Prevention of buffer overflow attacks using advanced stackguard," *Proceedings of the 2010 International Conference on Advances in Communication, Network, and Computing, IEEE Computer Society*, pp. 357–359, 2010.

[29] J. Wilander and M. A. Kamkar, "Comparison of publicly available tools for dynamic buffer overflow prevention," *NDSS*, vol. 3, pp. 149–162, 2003.

[30] H. M. Gisbert and I. Ripoll, "On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows," *IEEE International Symposium on Network Computing and Applications*, vol. 13, pp. 145–152, 2014.

[31] K. Limbandit and Y. Teng-Amnuay, "Misuse for security hardening assessment in application software deployment," *International Journal of Future Computer and Communication*, vol. 1, no. 2, pp. 147–150, 2012.

[32] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE Symposium on Security and Privacy*, pp. 191–205, Berkeley, CA, USA, 2013.

[33] "Related intel security features & technologies," https://software.intel.com/security-software-guidance/best-practices/related-intel-security-features-technologies.

[34] A. Kurmus, S. Dech, and B. Tu, "Quantifiable run-time kernel attack surface reduction," in *Lecture Notes in Computer Science*, pp. 212–234, 2014.

[35] X. Zhi, C. Hui-yu, and H. Hao, "Kernel rootket defense based on automatic data structure randomization," *Chinese Journal of Computers*, vol. 5, pp. 1100–1110, 2014.