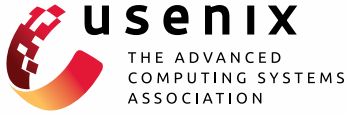


EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning

Ruowen Wang, *Samsung Research America and North Carolina State University;* William Enck and Douglas Reeves, *North Carolina State University;* Xinwen Zhang, *Samsung Research America;* Peng Ning, *Samsung Research America and North Carolina State University;* Dingbang Xu, Wu Zhou, and Ahmed M. Azab, *Samsung Research America*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-ruowen

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning

Ruowen Wang[1,2] William Enck[2] Douglas Reeves[2] Xinwen Zhang[1]
Peng Ning[1,2] Dingbang Xu[1] Wu Zhou[1] Ahmed M. Azab[1]

[1]*Samsung KNOX R&D, Samsung Research America*
{*peng.ning, xinwen.z1, dingbang.xu, wu1.zhou, a.azab*}*@samsung.com*
[2]*Department of Computer Science, NC State University*
{*rwang9, whenck, reeves, pning*}*@ncsu.edu*

Abstract

Mandatory protection systems such as SELinux and SEAndroid harden operating system integrity. Unfortunately, policy development is error prone and requires lengthy refinement using audit logs from deployed systems. While prior work has studied SELinux policy in detail, SEAndroid is relatively new and has received little attention. SEAndroid policy engineering differs significantly from SELinux: Android fundamentally differs from traditional Linux; the same policy is used on millions of devices for which new audit logs are continually available; and audit logs contain a mix of benign and malicious accesses. In this paper, we propose *EASEAndroid*, the first SEAndroid analytic platform for automatic policy analysis and refinement. Our key insight is that the policy refinement process can be modeled and automated using semi-supervised learning. Given an existing policy and a small set of known access patterns, EASEAndroid continually expands the knowledge base as new audit logs become available, producing suggestions for policy refinement. We evaluate EASEAndroid on 1.3 million audit logs from real-world devices. EASEAndroid successfully learns 2,518 new access patterns and generates 331 new policy rules. During this process, EASEAndroid discovers eight categories of attack access patterns in real devices, two of which are new attacks directly against the SEAndroid MAC mechanism.

1 Introduction

Operating system integrity relies on the correctness of 1) trusted computing base (TCB) code and 2) access control policy protecting the TCB code and OS resources. It is generally impractical to verify the correctness of OS code in commodity systems. Therefore, mandatory access control (MAC) policy is often used as a fallback when the security of the software inevitably fails [29].

SELinux [6] is the most notable MAC policy framework widely used in practice. Security Enhanced Android [38] (known simply as SEAndroid) is a recent port of SELinux to the Android platform. However, while Android is based on a Linux kernel, the runtime environment is vastly different than existing Linux distributions for commodity PCs. This difference resulted in a complete redesign of the MAC policy rules, with several new object classes (e.g., for Android's binder IPC).

As with SELinux, SEAndroid policy development is a challenging task, requiring many iterations of refinement to be ready for commercial deployment. For example, Google introduced a very permissive SEAndroid policy into Android version 4.3 and did not enable enforcement. Version 4.4 enabled enforcement, but the policy was still very permissive, containing only a few system daemons. Finally, Android version 5.0 provides a much more robust (but not perfect) version of the policy. Additionally, major smartphone vendors need to customize Google's base SEAndroid policy for their devices to add additional protections against known attacks.

SEAndroid policy refinement is currently a very manual process that typically involves analyzing audit logs to identify proposed changes. There are two general approaches to SEAndroid policy refinement. The first approach is to develop a least privilege [35] policy (also known as a "strict" policy in SELinux terminology) and monitor audit logs for access patterns that should be allowed. The second approach is to begin with a more permissive policy, refine the policy to prevent (or contain) privilege escalation attacks and use audit logs to verify the refinement. Each approach has disadvantages. If the policy is too strict, it will hurt the usability of deployed real-world devices. If the policy is too permissive, it will allow attacks. As a result, smartphone vendors use a combination of these two approaches.

The goal of our research is to significantly reduce the manual effort required to refine SEAndroid policy using audit logs. Audit log analysis is challenging for several reasons. First, audit logs are collected from millions of

real-world devices, and purely manual analysis is impractical. Second, the audit logs contain both benign access patterns and malicious access patterns. Existing SELinux tools such as `audit2allow` that blindly create rules to allow all access patterns in audit logs are error prone. Third, the functionality of both benign applications and malicious exploits is continually changing/upgrading, requiring frequent reassessment of the deny/allow boundary to refine the policy.

In this paper, we present Elastic Analytics for SEAndroid (EASEAndroid) as the first large-scale audit log and policy analytic platform for automatic policy analysis and refinement of SEAndroid-style MAC policy. Our key insight is that the policy refinement process can be modeled and automated using semi-supervised learning [18], a popular knowledge-base construction technique [16, 21]. We apply EASEAndroid to a database of 1.3 million audit logs from real-world Samsung devices running Android 4.3 over the entire year of 2014.[1] EASEAndroid correctly discovers 336 new benign access patterns and 2,182 new malicious access patterns, and automatically translates them into 331 policy rules. The generated rules are consistent with rules manually added by policy analysts. Among the malicious access patterns, EASEAndroid further discovers two new types of attacks in the wild directly targeting SEAndroid MAC mechanism itself.

This paper makes the following contributions:

- *We propose EASEAndroid, a semi-supervised learning approach for refining MAC policy at large scale.* Our approach scales to millions of audit logs that contain a mix of benign and malicious access patterns. While we focus on SEAndroid, the approach is more broadly applicable to type enforcement (TE) MAC policy.

- *We implement a prototype of EASEAndroid to help policy analysts analyze SEAndroid audit logs.* The implementation generates policy refinements, and discovers new Android attacks, providing new knowledge of both benign and malicious access patterns learned from audit logs for policy analysts.

- *We evaluate EASEAndroid on 1.3 million audit logs from real-world Samsung devices.* Using this dataset, EASEAndroid successfully learns 2,518 benign and malicious access patterns and generates 331 policy rules as a refinement. EASEAndroid also discovers two new types of attacks directly targeting SEAndroid. With the help of EASEAndroid, this is the first large-scale study on real-world malicious access patterns in Android devices.

[1] See Appendix A for more details about audit log collection.

The remainder of this paper proceeds as follows. Section 2 provides background on SEAndroid and semi-supervised learning. Section 3 defines the problem addressed in this paper. Section 4 describes the EASEAndroid design. Section 5 evaluates EASEAndroid against a large database of real-world audit logs. Section 6 discusses limitations. Section 7 overviews related work. Section 8 concludes.

2 Background

2.1 SELinux and SEAndroid

SEAndroid is a port of SELinux's type enforcement (TE) MAC policy to the Android platform [4]. As such, SEAndroid enforces mandatory policy on system-level operations between subjects and objects (e.g., system calls) [6]. In general, processes are regarded as subjects, whereas files, sockets, etc. are objects in different classes. A security context label is assigned to subjects (or objects) that share the same semantics. Traditionally, the subject label is called a *domain*, and the object label is called a *type* (nomenclature from DTE [12]). A policy rule defines which domain of subjects can operate which class and type of objects with a set of permissions, such as open, read, write [28]. For example,

```
allow app app_data_file:file {open read}
```

allows processes with the `app` domain to `open` and `read` `file` class objects assigned the `app_data_file` type. In addition to `allow` rules, SELinux provides `neverallow` rules to define policy invariants for malicious accesses that should never be allowed. These rules are enforced at policy compile-time and are necessary due to the complexity of the SELinux policy language.

SEAndroid extends SELinux's policy semantics to support Android-specific functionality, including mediation of Binder IPC and assigning security contexts based on application digital signatures. The goal of SEAndroid is to reduce the attack surface and limit the damage if any flaw or vulnerability is exploited causing privilege escalation [38]. This goal is accomplished by confining the capabilities of different privileged Android applications and system daemons.

The Android platform is vastly different than traditional Linux distributions, therefore the SEAndroid policy rules were created from scratch. While the regularity of Android's UNIX-level interactions results in a policy that is less complex than the example SELinux policy for PCs, the SEAndroid policy is still nontrivial and error prone. It requires careful understanding of subtle interactions between different privileged processes. In practice, policy development requires continual manual refinement based on audit logs.

```
type=1400 msg=audit(1399587808.122:14):
avc: denied { entrypoint } pid=285 comm="init"
scontext=u:r:init:s0
tcontext=u:object_r:system_file:s0 tclass=file

type=1300 msg=audit(1399587808.122:14):
syscall=11(execve) success=no exit=-13
items=1 ppid=1 pid=285 uid=0 gid=0
comm="init" exe="/init" subj=u:r:init:s0

type=1302 msg=audit(1399587808.122:14):
item=0 name="/system/etc/install-recovery.sh"
inode=3799 dev=b3:10 mode=0100755
ouid=0 ogid=0 obj=u:object_r:system_file:s0
```

Listing 1: A denied access event example recorded at the epoch time 1399587808.122 in an audit log. It consists of three entries: labels & permission (1400), syscall & process info (1300), object info (1302).

An audit log captures security labels and system calls of the operations that are not explicitly allowed by a rule. As shown in Listing 1, a denied operation generally has three entries with epoch timestamps. Log entries with `type=1400` record the denied permission (e.g., `entrypoint`), the security labels of the subject (source), called `scontext`, and the object (target), called `tcontext`, as well as the object's class, called `tclass` (e.g., `file`). Log entries with `type=1300` record the system call and the subject's process information, including the executable file path. Log entries with `type=1302` record the object information (e.g., the file name).

Traditionally, policy analysts develop and refine a policy by manually analyzing audit logs. Existing SELinux tools such as Tresys's setools [8] are used to analyze SEAndroid policies based on interactive user interface. Analysts also develop simple shell and Python scripts to parse audit logs. Unfortunately, such tools are not scalable to a large number of audit logs, and cannot distinguish benign or malicious access patterns in real-world audit logs. In addition, analysts often use a tool called `audit2allow` [9] that can create new `allow` rules by directly using the security labels captured in `type=1400` entries in audit logs. However, blindly using this tool may increase attack surface, because in some cases, existing labels are too coarse-grained or semantically inappropriate. Therefore, policy refinement usually consists of the creation and modification of both security labels and policy rules. Once the policy is refined by analysts, it is pushed to users' devices through a secure over-the-air (OTA) channel, similar to antivirus signature updates.

2.2 Semi-Supervised Learning

Semi-supervised learning is a type of machine learning that trains on both labeled[2] data (used by supervised learning) and unlabeled data (used by unsupervised learning) [18]. It is typically used when labeled data is insufficient and expensive to collect, and a large set of unlabeled data is available. By correlating the features in unlabeled data with labeled data, a semi-supervised learner infers the labels of the unlabeled instances with strong correlation. This labeling increases the size of labeled data set, which can be used to further re-train and improve the learning accuracy [44]. This iterative training process is commonly referred to as bootstrapping. Semi-supervised learning is popular for information extraction and knowledge base construction. Examples include NELL [16, 17], Google Knowledge Vault [21].

We hypothesize that the process of developing and refining SEAndroid policy is analogous to semi-supervised learning. Human analysts encode their knowledge about various access patterns into a policy. When analyzing audit logs, analysts find semantic correlations between known and unknown access patterns to infer whether the unknown ones are benign or malicious, such as a known malicious subject performing an unseen behavior (likely malicious), or a system daemon performing a new but similar functional operation (likely benign). These new patterns expand analysts' knowledge and help them refine the policy. However, when more and more logs are collected containing access patterns about new Android systems and new attacks, manual learning is time-consuming and likely to miss important knowledge. Our insight is that semi-supervised learning can automate this process to achieve scalability in policy refinement.

3 Problem

Refining SEAndroid policy is more challenging than refining SELinux policy. Existing SELinux tools such as `audit2allow` are severely limited in their ability to help policy analysts. This task has the following challenges.

C-1: *Consumer devices produce millions of audit logs.* Policy analysts cannot practically analyze audit log entries manually. A solution must automate or semi-automate the audit log analysis.

C-2: *Real-world audit logs contain a mixture of benign and malicious accesses.* Classifying log entries as benign or malicious is a central design challenge. It is often difficult to classify an access in isolation. Instead, the analysis must look at the broad context of the access, as well as the contexts of related known accesses.

C-3: *Target functionality is not static.* The set of benign and malicious applications continues to evolve as new software and malware is developed, requiring continuous audit log analysis and policy refinement.

[2] Here, labeled and unlabeled are machine learning terms, not related to security labels.

For example, benign software may access new resources, while malware may exploit new vulnerabilities to achieve privilege escalation.

We now define two terms to clarify the discussion in the remainder of this paper.

Definition 1 (Access Event)**.** *An access event is the access control event that causes the three audit log entries described in Section 2. These log entries may result from a policy denial, or an* `auditallow` *policy rule, which allows but logs the access.*

Note that this definition does not include allowed accesses that are not contained in the audit log.

Since audit logs are collected for millions of devices, the logs contain many duplicate access events. For the purposes of audit log analysis, it is useful to abstract the salient details of access events into an access pattern.

Definition 2 (Access Pattern)**.** *An access pattern is a 6-tuple (sbj, sbj_label, perm, tclass, obj, obj_label). Many access events may map to the same access pattern.*

Here *sbj* refers to a concrete subject such as an Android application or system binary. Binaries carried inside an application are generalized as the application. *obj* refers to concrete objects such as file paths and socket names. In some cases, we group over-specific files that share the same filesystem semantics as one *obj* (e.g., `/sdcard`). The values of *sbj* and *obj* are derived from the `comm`, `exe`, `pid`, and `name` values in the `type=1300,1302` log entries. *perm* and *tclass* are the same as the permission and the object's class in the `type=1400` log entries and policy rules. *sbj_label* and *obj_label* are derived from the `scontext` and `tcontext` values in `type=1400` log entries. For example, the access pattern for the access event in Listing 1 is ("`/init`", "`init`", "`entrypoint`", "`file`", "`/system/etc/install-recovery.sh`", "`system_file`").

Problem Statement: Given 1) a large dataset of new access patterns from audit logs, 2) a small set of known access patterns (e.g., known attacks), and 3) an SEAndroid policy, we seek to a) separate new benign access patterns from new malicious access patterns in the dataset, and b) suggest new rules and refined labels for the policy.

Threat Model and Assumptions: We assume that an audit log is collected from an Android device with a policy loaded in either enforcing or permissive mode. We assume the integrity of audit log contents. We therefore assume that the Linux kernel and its audit subsystem are not compromised. However, even if the SEAndroid policy properly confines Android applications and system daemons, they may be compromised by the adversary.

4 EASEAndroid

Elastic Analytics for SEAndroid (EASEAndroid) is a large-scale audit log and policy analytic platform for automated policy refinement. The novelty of EASEAndroid is that it models the policy refining process as a semi-supervised learning of new access patterns. At a high level, EASEAndroid starts with an initial knowledge base containing existing policy rules and a small set of (potentially manually) identified access patterns. It expands the knowledge base by correlating, classifying, and incorporating new access patterns captured by audit logs. Based on the new knowledge, EASEAndroid suggests policy changes (new rules and new domain and type labels in the context of SEAndroid). As more audit logs become available, EASEAndroid continuously expands the knowledge base and refines the policy.

Figure 1 shows the architecture of EASEAndroid. The architecture uses three machine learning algorithms that consider different perspectives of the knowledge base and audit logs. The output of these algorithms is fed into a combiner that combines and appends the new knowledge into the knowledge base. This learning process is iterated multiple times until no more new knowledge can be learned from the current audit log input. Finally, the policy generator suggests refinements.

Each machine learning algorithm analyzes a different perspective of the data. The goal of each algorithm is to find semantic correlations between unknown new access patterns and existing knowledge base, in order to classify each new access pattern as benign or malicious.

1. The *nearest-neighbors-based (NN) classifier* classifies new access patterns based on their relations to known access patterns in the knowledge base. It finds new access patterns that are related to known subjects/objects (e.g., known subjects are updated and perform new access patterns). By treating these known subjects/objects as neighbors of the new access patterns, it classifies the new access patterns based on the majority of their known neighbors.
2. The *pattern-to-rule distance measurer* calculates the distance between new access patterns and existing policy rules. If a new access pattern is closest to an `allow` rule, it is classified as benign. If it is closest to a `neverallow` rule, it is classified as malicious. If the access pattern is not close to either type of rule, it remains unclassified. The pattern-to-rule distance measurer also exposes potentially incomplete rules in existing policy for refinement.
3. The *co-occurrence learner* considers correlations across access patterns using statistical relations between new and known access patterns that frequently occur together in audit logs. Our intuition is

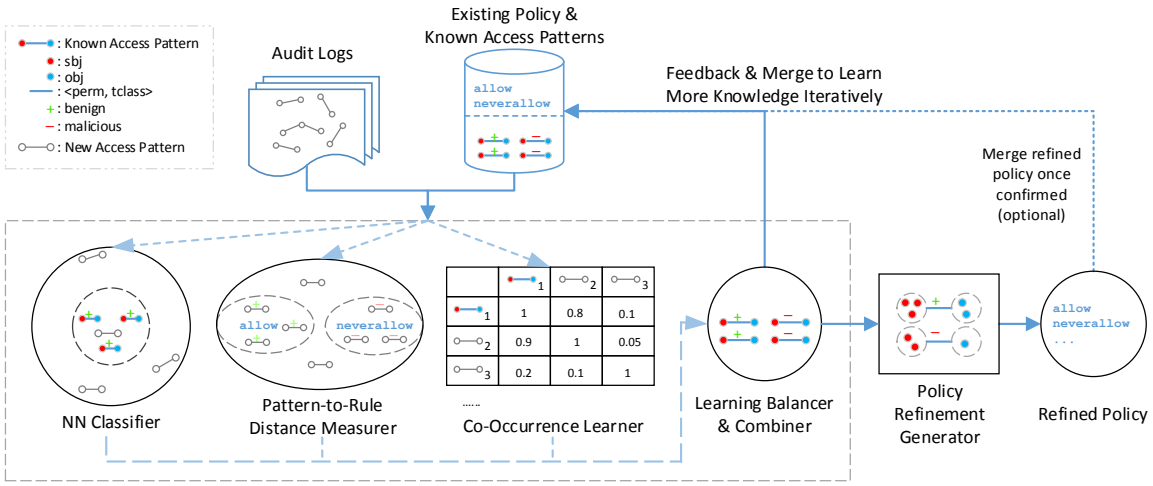


Figure 1: EASEAndroid consists of four learning components and a policy generator to iteratively learn new patterns and refine policy

that a benign functionality or a malicious attack often involves a series of access patterns that are captured together in an audit log. If known and new access patterns occur together, we can use the known ones to infer the classification of the new ones.

Each learner is configured with its own threshold to classify new access patterns independently. However, it is non-trivial to define proper thresholds because too relaxed thresholds could cause false classification, lowering the learning precision, while too strict thresholds could leave potential access pattern candidates unclassified, lowering the learning coverage.

The *learning balancer & combiner* manages the threshold of each learner, balances the precision and coverage, and combines the classification results from the three learners. It has two modes: (1) an *automated mode* that uses strict threshold in each learner to achieve high precision with the cost of less coverage; and (2) a *semi-automated mode* that relaxes each learner's threshold to achieve high coverage and relies on a majority vote from the three learners to increase the precision. In practice, the result of semi-automated mode requires policy analysts' verification to control error rate.

Finally, the *policy generator* takes newly classified access patterns as input from the combiner, to suggest policy refinements in the form of new rules and new security labels. It uses a clustering algorithm to group similar subjects and objects together. The clustering algorithm follows the principle of least privilege by inferring fine-grained labels that can cover and only cover the clustered concrete subjects and objects. In practice, the resulting refined policy can be confirmed by policy analysts and merge into the knowledge base to analyze new audit logs after the refined policy is deployed.

The remainder of this section describes each stage of the EASEAndroid architecture in detail.

4.1 Nearest-Neighbors-based Classifier

Nearest-neighbors-based (NN) learning is a common technique for classifying an unlabeled instance based on its nearest labeled neighbors within a defined distance [41]. Our intuition of using NN for access pattern classification is two-fold. First, known subjects often perform previously unseen access patterns in audit logs. This scenario often occurs when Android applications and system binaries are updated with new capabilities. Second, some known access patterns are also performed by new subjects. This scenario occurs when certain operations become popular and are copied by other new applications. In practice, some exploit kits and repackaged applications [43] have been found to share the same set of known malicious access patterns.

These two scenarios cause known subjects and patterns to be semantically connected with new subjects and patterns. EASEAndroid leverages this connectivity as the distance metric to design the NN classifier. When multiple known subjects (or patterns) connect to the same new pattern (or subject), the NN classifier can infer whether the new pattern (or subject) is benign or malicious, based on the majority of the connected known neighbors. Note that here the observation is with respect to concrete subjects and objects in access patterns. Hence, only a 4-tuple $(sbj, perm, tclass, obj)$ out of the original 6-tuple is required. For completeness, our implementation still includes sbj_label and obj_label in the dataset, but they are not used in this learner.

Algorithm 1 shows the procedure of the NN classifier. AP_k collects known 4-tuple access patterns, either benign or malicious. In practice, our AP_k is a small set contain-

Algorithm 1 NN-based Classification of Access Patterns

$AP_k \leftarrow \{(s_k,p_k,t_k,o_k)|s_k \in S_k,(p_k,t_k,o_k) \in P_k\}$
$AP_u \leftarrow \{(s_u,p_u,t_u,o_u)|s_u \in S_u,(p_u,t_u,o_u) \in P_u\}$
$AP_c \leftarrow \emptyset$
procedure NN_CLASSIFIER(AP_k,AP_u,AP_c)
  for each $(s,p,t,o) \in AP_u$ **do**
    if $s \in S_k \cap S_u$ and $(p,t,o) \in P_u - P_k$ **then**
      $S_{tmp} \leftarrow findAllSbjs((p,t,o),AP_u)$
      if $IsMajorityKnown(S_{tmp},S_k)$ **then**
        $AP_c \leftarrow AP_c \cup Classify((s,p,t,o))$
      end if
    else if $s \in S_u - S_k$ and $(p,t,o) \in P_k \cap P_u$ **then**
      $P_{tmp} \leftarrow findAllPatterns(s,AP_u)$
      if $IsMajorityKnown(P_{tmp},P_k)$ **then**
        $AP_c \leftarrow AP_c \cup Classify((s,p,t,o))$
      end if
    end if
  end for
end procedure
return AP_c

ing a few well-confirmed subjects and patterns, used as the initial seed. AP_u collects all unknown new access patterns from audit logs. To clearly describe the above two cases, we further divide the 4-tuple into S for all subjects, and P for the triples *(perm, tclass, obj)* as partial patterns shared by multiple subjects. AP_c is the result set of newly classified access patterns.

For each 4-tuple in AP_u, we check if it is a known subject with a new triple (partial pattern), or a new subject with a known triple. In the first case, besides the subject in this 4-tuple, $findAllSbjs$ collects all subjects S_{tmp} that perform (connect) the same new triple in AP_u, including both known and new subjects. Then $IsMajorityKnown$ checks if the majority of S_{tmp} is a set of known subjects from S_k with the same benign or malicious flag. If so, the new access pattern is classified as benign or malicious accordingly. The second case is done in the same way but using known triples to classify new subjects.

The function $IsMajorityKnown$ uses two empirically defined thresholds (m,σ). m determines the minimum required neighbors and σ is a percentage for how many known neighbors in S_{tmp} or P_{tmp} are required as a majority. Table 1 in the evaluation studies the effects of different threshold values.

From the perspective of machine learning, our NN-based classifier is a type of *radius-based near neighbors learning* [13], a variant of the common k-nearest-neighbors (kNN). The difference is that kNN is based on the top k neighbors while we find all neighbors within a radius as nearest neighbors (connectivity is the radius in our case).

Note that, it is possible that some access patterns are rarely connected with known ones. Besides, an access pattern could be evenly connected to both known benign and malicious ones. Both cases cause $IsMajorityKnown$ to return false. In this case, the NN classifier leaves the access patterns as unclassified and relies on the following learners to complement the learning process.

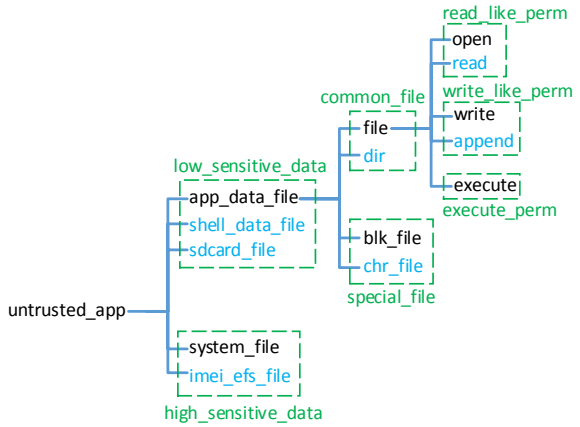


Figure 2: A decision tree example based on rules related to subject domain as `untrusted_app`. The black nodes are from existing rules and the blue nodes are semantic siblings.

4.2 Pattern-to-Rule Distance Measurer

EASEAndroid's second data perspective is the closeness of access patterns to policy rules. Since audit logs record denied accesses that cannot match with an `allow` rule, it is useful to know how far/close the denied access pattern is from an existing rule. In particular, because policy rules are developed incrementally, they may only cover a subset of permissions or access patterns and miss similar access patterns belonging to the same operation.

A common case of this is an imprecise list of permissions in an `allow` rule. For example, writing a file not only requires `write` permission, but also `append` and sometimes `create` (in case the file does not exist). Some malicious operations can also be performed using semantically equivalent, but different access patterns.

The pattern-to-rule distance measurer quantifies the difference between access patterns and existing rules. The purpose of this measurer is two-fold. First, pattern-to-rule distance indicates how likely a new access pattern is to be benign or malicious. Second, if an access pattern is very close to a policy rule, the policy refinement generator (Section 4.5) can update the rule rather than creating a new rule from scratch.

The distance measurer uses a metric based on the 1) subject label, i.e., *domain*, 2) object label, i.e., *type*, 3) tclass, e.g., `file`, and 4) permission, e.g., `write`. Note that all four of these elements are in both the SE-

Android policy `allow` rules, as well as the 6-tuple representing an access pattern in the audit log. Intuitively, an access pattern is very close to a rule if it shares the same labels and tclasses only with slightly different permissions (e.g., `write` vs `append`). The distance increases a little, but is still close, if a pattern and a rule operate on different but similar tclasses (e.g., `file` vs `dir`).

EASEAndroid systematically measures distance using decision trees based on existing policy rules. The distance is defined by the matching depth for a specific access pattern. Decision trees are built as follows.

Step 1: For every subject label, find all related rules and follow their semantic order to build a tree skeleton starting from the subject as the root, followed by object labels, tclasses and permissions as nodes in each layer.

Step 2: Extend each node with its semantically similar siblings.

Figure 2 shows an example decision tree. The black nodes indicate the tree skeleton, which uses rules such as `allow untrusted_app app_data_file:file {open}`, where `app_data_file` is the object node in the second layer and `file` is the tclass node in the third layer and so on. Then each node is extended with its semantic siblings, such as `sdcard_file` in the same group of low sensitive data as app data.

Given an access pattern and a decision tree, the distance measurer walks the decision tree and tries to match the access pattern's subject label, object label, tclass, and permission with each layer. The matching depth indicates how close a pattern is to existing rules. For the example in Figure 2, access pattern $ap_i =$ (`untrusted_app`, `sdcard_file`, `dir`, `read`) matches the fourth layer. The distance is computed as follows:

$$Dist(ap_i) = TotalLayerDepth - MatchedDepth(ap_i)$$

If we define $TotalLayerDepth = 4$, then $Dist(ap_i) = 0$, indicating the access pattern is very close to the rule. We create trees for both `allow` and `neverallow` rules to compute the distances from both sides.

In practice, the effectiveness of the distance metric depends on the correctness of semantic siblings. Fortunately, the SEAndroid policy development frequently uses semantic groups. A list of permissions, tclasses, and object types are already grouped together in policy source code using macros and attribute [7]. These groups form a ground truth for semantic siblings.

Additionally, recall that some existing subject and object labels are coarse-grained (e.g., labels assigned to various objects using wildcard in policy source code). If a pattern matches with a rule with a coarse-grained label, the distance measurer marks the distance as low confidence and relies on the learning balancer & combiner for additional verification (Section 4.4).

Finally, note that this technique can be further extended to measure access pattern to access pattern distance. Since the pattern-to-rule distance helps to infer both new patterns as well as identify incomplete rules for refinement, our design considers policy rules and leaves the distance between access patterns for future work.

4.3 Co-Occurrence Learner

When analyzing a large number of audit logs, some access patterns frequently occur together in many logs. This is because some high-level benign functionality or some popular multi-step attacks consist of a series of access patterns within a time period (typically minutes). The statistics of co-occurrence is a valuable means of correlating access patterns that have different subjects or objects, but share the same group semantics. When a group contains both known and new access patterns, the known access patterns can be used to infer the semantics of the new access patterns. In fact, co-occurrence is popular in natural language processing and knowledge extraction. For example, it is used for finding words that are frequently used together in a specialized domain [15].

The co-occurrence of access pattern can be represented using a $n \times n$ matrix for all n unique access patterns from the audit logs, as shown below. Each row stores one access pattern ap_i's co-occurrence percentage with every other access pattern, denoted in each column. The value c_{ij} is the percentage of the number of times that ap_i co-occurs with ap_j out of the total number of ap_i's occurrences throughout the logs.

$$CO_{AP} = \begin{array}{c} \\ ap_i \\ ap_j \\ \dots \end{array} \begin{array}{c} \begin{array}{ccc} ap_i & ap_j & \dots \end{array} \\ \begin{bmatrix} 1 & c_{ij} & \dots \\ c_{ji} & 1 & \dots \\ \dots & \dots & 1 \end{bmatrix} \end{array},$$

$$\text{where } c_{ij} = \frac{CoOccurNum(ap_i, ap_j)}{TotalOccurNum(ap_i)}$$

When counting the number of co-occurrences, it is important to avoid noise and duplicates. In practice, we use a time frame of 10 minutes to determine whether two access patterns are part of a co-occurrence set. Recall from Section 2 that each access pattern has an epoch timestamp. Additionally, when counting the occurrence at the granularity of logs, repeated pairs of co-occurred access patterns in one log are counted only once.

To use this co-occurrence matrix, the learner focuses on the rows with new access patterns. For each ap_i row, the learner sorts columns and selects the set of known access patterns in columns whose percentage is above a threshold. A majority vote of this known access pattern set determines the classification of the new ap_i (benign or malicious). On the other hand, the known access pattern rows may also have some highly co-occurred new access

pattern columns. However, one known access pattern is usually not enough to classify a new access pattern.

Note that the matrix is not symmetric. c_{ij} can be different from c_{ji} due to different total occurrence counts. For instance, some popular known malicious access patterns (e.g., `remount /system`) can co-occur with multiple less popular new access patterns, because multi-step attacks often use different steppingstones to achieve the final privilege escalation goal.

4.4 Learning Balancer & Combiner

Each learner is configured with its own threshold to classify new access patterns independently. However, it is non-trivial to define proper thresholds due to two reasons. On the one hand, if a threshold is too relaxed, it could cause false classification, lowering the learning precision and might further propagate the error to the next iteration of semi-supervised learning. On the other hand, if a threshold is too strict, it could miss potential access pattern candidates and leaves them as unclassified, lowering the learning coverage.

We design the learning balancer & combiner to manage the threshold setting of each learner, balance the precision and coverage, and combine the classification results from the three learners (also called ensemble or multi-view learning [17]). The final combined classification result is added to the knowledge base and sent to the policy refinement generator. Specifically, we propose two quantifiable methods to achieve the balancing:

Automated Mode: Since each learner specializes in one dimension, each learner with a strict threshold can directly contribute its classified access patterns with high precision. For example, we can set a minimum of 10 required known neighbors with a 90% bar for *IsMajorityKnown* in NN classifier; $Dist(ap_i) = 0$ with fine-grained rules in pattern-to-rule distance measurer; and $c_{ij} > 0.9$ with known access pattern set ≥ 10 in co-occurrence learner. The high precision of strict thresholds enables EASEAndroid to be used in an automated mode over multiple iterations of semi-supervised learning. However, with such strict thresholds, some access pattern candidates can be left as unclassified.

Semi-Automated Mode: This mode relaxes the thresholds to get more access pattern candidates. It uses a majority vote to choose the candidates shared by at least two learners with the same classification result, and the third learner must not have conflicting result.

Note that relaxed thresholds can increase the possibility of error propagation. However, if the analysis can tolerate a semi-automated configuration, relaxed thresholds can be used. Here a human analyst can investigate low-confident candidates and input external knowledge into EASEAndroid for better learning in future.

4.5 Policy Refinement Generator

Finally, the policy refinement generator translates newly classified access patterns[3] into the final policy form. A key part of the generator is to assign the concrete subjects (*sbjs*) and objects (*objs*) in the access pattern with appropriate security labels before generating policy rules.

According to the Android Open Source Project, Google provides a baseline definition of security labels for common subjects (e.g., system apps and binaries) and basic objects (e.g., basic files/dirs in Android file system structure). However, Google recommends that manufacturers replace the generic default labels with fine-grained labels to decrease the attack surface [4].

Recall that both the access pattern 6-tuple and the incomplete rules identified by the distance measurer include subject labels and object labels from the existing policy. While some of the labels are coarse-grained, they serve as a baseline to derive fine-grained labels. Specifically, the policy refinement generator takes all access patterns as input and clusters them into groups where each group shares the same 4-tuple *(sbj_label, perm, tclass, obj_label)*. Each group is further clustered by *sbjs* and *objs* to create subgroups that share the detailed semantics to derive fine-grained labels.

Our current generator prototype groups *sbjs* and derives fine-grained subject labels for built-in, vendor, and untrusted applications and binaries separately. The generator also groups file-like objects (e.g., `file`, `dir`, `blk_file`), which comprise the majority of tclasses. Group is performed using a *longest common prefix* search on file paths. This optimization helps to derive more fine-grained labels than provided by the general Android filesystem structure. Finally, the generator produces rules in the form of *(new_sbj_label, perm, tclass, new_obj_label)* as a policy refinement. If access patterns are matched with incomplete rules by the distance measurer, new rules merge with existing rules' permissions.

The generator handles benign and malicious patterns separately and generates `allow` and `neverallow` rules, respectively. Note that, it is possible that newly generated rules may conflict with existing rules due to incomplete or tightened access control. In such cases, policy analysts manually resolve conflicts (e.g., using `auditallow` to verify). Nevertheless, EASEAndroid exposes these conflicts with evidence collected through learning, therefore easing the policy refining process.

[3] In practice, the learning process can iterate multiple times with current audit logs. The generator caches all classified access patterns.

5 Evaluation

We implement a prototype of EASEAndroid and evaluate the learning capability and the security effectiveness of EASEAndroid from three perspectives:

1. We evaluate the coverage and precision of the classification result of EASEAndroid, and how they are affected by different threshold settings (Section 5.3).
2. We conduct a case study of the policy refinement generated by EASEAndroid, also comparing the generated rules with human-written rules (Section 5.4).
3. We further conduct a study on the new malicious access patterns classified by EASEAndroid and discuss several interesting new findings of attacks in the wild (Section 5.5).

5.1 Environment Setup

We build a prototype of EASEAndroid on an 8-node Hadoop cluster with each node having 8-core Xeon 2GHz, 32 GB memory. We use open source Cloudera Impala as the distributed SQL layer, with 10K SLOC Java as the learning layer. Parallelism is heavily employed for fast analytics. A data set of 1.3 million audit logs used in the following experiments are analyzed by EASEAndroid within 3 hours in a cold start.

5.2 Audit Log & Existing Knowledge

Audit Logs & Existing Policy We make use of 1.3 million audit logs over the entire 2014 from real-world devices running Android 4.3 (See Appendix A about audit log collection). All devices are loaded with an early version of Samsung SEAndroid policy (the policy remained unchanged) in enforcing mode[4]. The policy contains 5,094 `allow` rules and 59 `neverallow` rules developed by policy analysts. This policy is loaded as existing knowledge into EASEAndroid's knowledge base, used by the pattern-to-rule distance measurer.

The audit logs contain a total of over 14 million denied access events. After eliminating duplicate entries, we identify approximately 145K unique access events and further generalize them into 3,530 access patterns. For example, third-party app process ids under `/proc/` are generalized as `/proc/app_pid` in access patterns.

The subjects in the audit logs consist of 113 system (built-in) binaries, 1,182 external binaries (e.g., installed by `adb`), and 626 Android apps, which are captured because they perform system-level operations that do not go through Android framework/Dalvik VM (normal app operations are already allowed by the policy).

Initial Known Malicious Access Patterns In the initial knowledge base, we prepare a small set of known malicious access patterns as the initial seed to kick off learning. The set contains 9 confirmed exploit kits with their 17 malicious access patterns (e.g., psneuter CVE-2011-1149, Motochopper CVE-2013-2596, vroot CVE-2013-6282 and several exploit apps). Note that we do not have known benign access patterns initially as we rely on the `allow` rules in the existing policy.

Ground Truth To analyze the classification result of benign and malicious access patterns, we use a later version of human-written policy (6,337 `allow` rules, 94 `neverallow` rules) as the ground truth. We also consult with experienced policy analysts about the result.

5.3 Coverage & Precision of the Classification by EASEAndroid

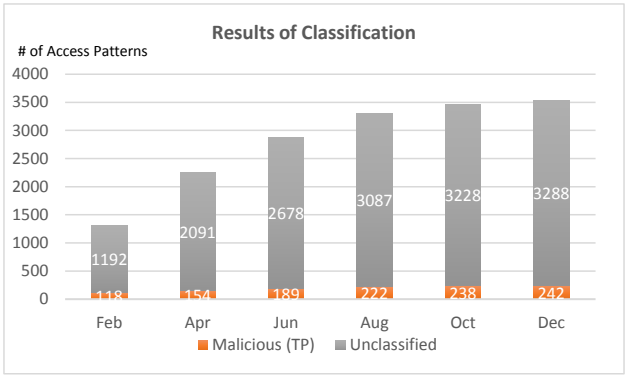
5.3.1 Coverage compared with naive matching

To illustrate the effect of EASEAndroid's learning coverage, we design a *naive matching* tool as a baseline to compare with EASEAndroid learning when both analyzing the same set of new access patterns from the audit logs, as shown in Figure 3. The naive matching tool is a dumb access pattern matching tool with no learning capability. It only uses the known subjects and access patterns in the initial knowledge base and can only match new access patterns directly related to them, based on the subjects and objects in the syscall entries in audit logs. In contrast, EASEAndroid starts from the initial knowledge base and keeps expanding the knowledge base.

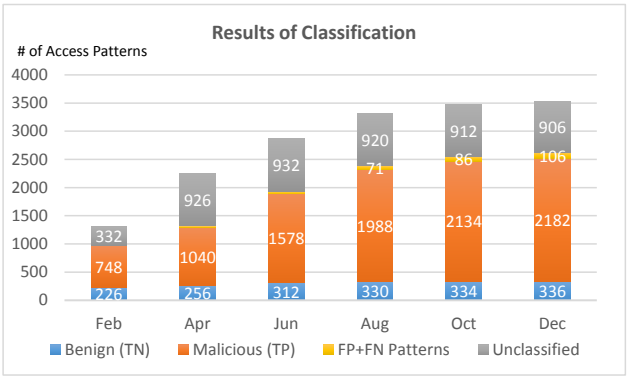
As the audit logs are continuously collected over the year, we setup 6 analyses at a rate of every two months. Each analysis takes as input the accumulated audit logs from Jan 2014 to the current month (e.g., "Feb" is 2-month logs, "Apr" is 4-month logs, "Dec" is the entire year's logs). It is a typical scenario of semi-supervised learning with incremental input data. It also follows the nature that new benign/malicious patterns are gradually accumulated in audit logs over time.

As shown in Figure 3, EASEAndroid dramatically outperforms the naive matching in each analysis. As the total number of access pattern keeps increasing, EASEAndroid's coverage reaches about 74% in the final December analysis. EASEAndroid also discovers that the majority of denied access patterns in real world are malicious and they keep emerging while benign access patterns gradually stabilize. In contrast, the coverage of naive matching remains around 7%, because it can only match access patterns related to the initial known ones, the 9 exploit kits, which are updated with a small set of

[4] Some devices are found being rooted and may switch to permissive mode. See Section 5.5



(a) Access patterns matched by naive matching with no learning capability as the baseline. A small set of new malicious patterns are matched related to the 9 exploits since they are updated and keep trying some new malicious patterns over time. But the majority is still unclassified.



(b) Access patterns classified by EASEAndroid starting from the initial knowledge. New patterns classified in each analysis become the incremental knowledge to classify more patterns in the next analysis. Benign patterns get stable over time, while new malicious patterns keep emerging.

Figure 3: The comparison between naive matching and EASEAndroid on analyzing the same set of access patterns

Threshold Setting	Classified Malicious (TP+FP)	Classified Benign (TN+FN)	Remain Unclassified	True Malicious (TP)	False Malicious (FP)	True Benign (TN)	False Benign (FN)
$\sigma = 55\%, Dist \leq 2, c_{ij} > 0.55$	77.2%	14.0%	8.8%	62.96%	37.04%	58.65%	41.35%
$\sigma = 65\%, Dist \leq 1, c_{ij} > 0.65$	70.0%	11.8%	18.2%	88.73%	11.27%	71.35%	28.65%
$\sigma = 75\%, Dist \leq 1, c_{ij} > 0.75$	65.7%	10.9%	23.4%	91.35%	8.65%	88.92%	11.08%
$\sigma = 85\%, Dist = 0, c_{ij} > 0.85$	63.9%	10.5%	25.7%	96.81%	3.19%	90.81%	9.19%
$\sigma = 95\%, Dist = 0, c_{ij} > 0.95$	53.1%	9.2%	37.7%	97.27%	2.73%	100.00%	0.00%

Table 1: The coverage and precision of EASEAndroid with different threshold settings after comparison with ground truth. The first three columns summarize the overall classification coverage over the 3,530 patterns. The following four columns give more details about the percentages of each set of true/false-classified benign and malicious patterns. Row 4 is the threshold setting used in Figure 3(b).

new access patterns over time. But it still leaves the majority unclassified.

Specifically, all three learners of EASEAndroid contribute to the high classification coverage. In the first February analysis, EASEAndroid first matches 118 malicious access patterns, same as naive matching. Then it performs multiple learning iterations with the current audit logs in both automated and semi-automated mode. In summary, in automated mode, the NN classifier finds 282 patterns using threshold $(m, \sigma) = (10, 85\%)$ in $IsMajorityKnown$. The pattern-to-rule distance measurer finds 95 patterns using $Dist(ap_i) = 0$ with existing `neverallow` rules. The co-occurrence learner finds 110 patterns with $c_{ij} > 0.85$. In semi-automated mode, we relax the thresholds to (10,75%), $Dist(ap_i) \leq 1$, $c_{ij} > 0.75$, respectively and further find 143 patterns based on the majority vote of the three learners.

As for benign access patterns in the February analysis, since the initial knowledge lacks benign patterns, the pattern-to-rule distance measurer classifies the first 23 benign patterns using $Dist(ap_i) = 0$ with existing `allow` rules and adds them to the knowledge base. Then the three learners contribute the remaining 203 using the same threshold settings.

Due to the strict thresholds, the automated mode classifies access patterns with no false positives or negatives. However, in the semi-automated mode, we do find 34 false-benign (False-Negative[5]) access patterns and the 72 false-malicious (False-Positive) ones in the final December analysis, mainly due to two reasons. First, a small set of access patterns are shared by both privileged benign system binaries and malicious apps with similar occurrences (e.g., both access `/proc/stat`), which make EASEAndroid hard to distinguish with relaxed thresholds. Second, some mis-classified patterns in early analyses affect the learning precision in later ones. In fact, there are only 4 false-malicious patterns in the February analysis. But then the NN classifier uses them to mistakenly find more false-malicious ones in the following analyses. Nevertheless, this limitation of the semi-automated mode is expected. Therefore in practice, the semi-automated mode requires policy analysts to verify the result to avoid error propagation. Analysts can also input extra constraints and knowledge about the access patterns of privileged system binaries to help EASEAndroid increase the precision.

There are still 906 access patterns unclassified in the

[5]We treat malicious as positive, benign as negative.

final analysis due to their low occurrence (less than five days throughout the year). After manual analysis, we find that some access patterns are likely malicious and might be isolated attack attempts in the wild. However, the statistics is too low to reach the threshold. In such cases, we have to wait for more similar access patterns coming in future audit logs. In Section 6, we discuss the limitation that isolated/targeted attacks may evade EASEAndroid's detection if they are not widely spread.

5.3.2 Coverage & precision with different threshold settings

The thresholds for the three learners play an important role on the coverage and the precision of EASEAndroid. In practice, it is important to find a balance between the coverage and the precision. In this section, we further investigate the detailed coverage and precision difference by choosing 5 different threshold settings from very relaxed to very strict as shown in Table 1. The listed threshold settings are for the automated mode. The semi-automated mode is relaxed by reducing 10% on both *IsMajorityKnown* (minimum neighbors unchanged) and c_{ij}, and increasing 1 in $Dist(ap_i)$. The first three columns show the overall percentage (adds to 100%) summarizing the classified malicious and benign and unclassified over the total 3,530 access patterns. The following four columns provide the more detailed TP/FP and TN/FN percentage of classified malicious and benign access patterns.

We can see that the thresholds in Row 1 is largely relaxed. Although it has the highest coverage, both FP (37.04%) and FN (41.35%) are too high, making it practically useless. In contrast, Row 5's thresholds achieve 100% correctness on classifying benign patterns. But it also leaves 37.7% patterns unclassified. The middle 3 rows are more balanced. Row 3 and 4 are candidates for practical use. In practice, analysts can also use multiple thresholds respectively, such as with Row 4 and 5 together, and only need to investigate the diff of their learning results since we have high confidence with the result of Row 5.

Admittedly, each individual threshold for each learner may have a different effect on the final classification result. To analyze more detailed threshold difference, or find an optimal vector of thresholds, a cross-validation [27] can be performed with multiple real-world audit log sets.

5.4 Case Study of Refinement Generation & Comparison with Human Policy

In the last December analysis in Figure 3, the policy refinement generator finally generates 51 new `allow` rules from the 336 benign access patterns, and 280 new `neverallow` rules from the 2,182 malicious access patterns, by extending identified incomplete rules and creating new fine-grained security labels to replace existing coarse-grained ones. In this section, we use the following example as a case study to illustrate the generated refinement.

EASEAndroid classified as benign 9 access patterns that read some time-zone data files under `/data/misc/zoneinfo`. These access patterns are found in multiple Android framework-related binaries (subjects) in `/system/bin`, including `surfaceflinger`,`dhcpcd`,`pppd` and a vendor-specific daemon. In the 6-tuples, the time-zone data files carry `system_data_file`, which is the default label for all files under `/data`. Naively generating a rule with this label (using `audit2allow`) would over-grant the subjects with permissions to access all files under `/data`.

EASEAndroid instead finds that these files all share the same `/data/misc/zoneinfo` file path prefix, and thus derives a new label `zoneinfo_file` specifically for them, adding to the labeling definition `file_contexts`:

```
/data/misc/zoneinfo/.* \
u:object_r:zoneinfo_file:s0
```

In practice, the full path prefix can be transformed into an underscore-joined label to keep the semantics and prevent conflict (though abbreviation may be required). EASEAndroid also creates a new attribute `access_zoneinfo_domain` to group the above subject domains, as the following:

```
attribute access_zoneinfo_domain;
typeattribute surfaceflinger \
  access_zoneinfo_domain;...
```

Finally, EASEAndroid generates a new rule based on the new labels defined above:

```
allow access_zoneinfo_domain
zoneinfo_file:file {open read}
```

This rule only covers the 9 patterns observed by EASEAndroid, thus preventing unnecessary accesses being granted, following the least privilege principle.

The rules generated by EASEAndroid for the 336 benign access patterns are compared with human-written rules in the later policy version. Semantically, all access patterns allowed by EASEAndroid rules are also permitted by human-written rules. However, syntactically, EASEAndroid in general creates a larger set of more-specific rules, while human-written rules are more concise with the frequent use of policy `macros`, which ease the policy writing and are expanded during compile time [7]. It may be desirable to aggregate EASEAndroid's more specific rules for better human-readability; this remains for future work.

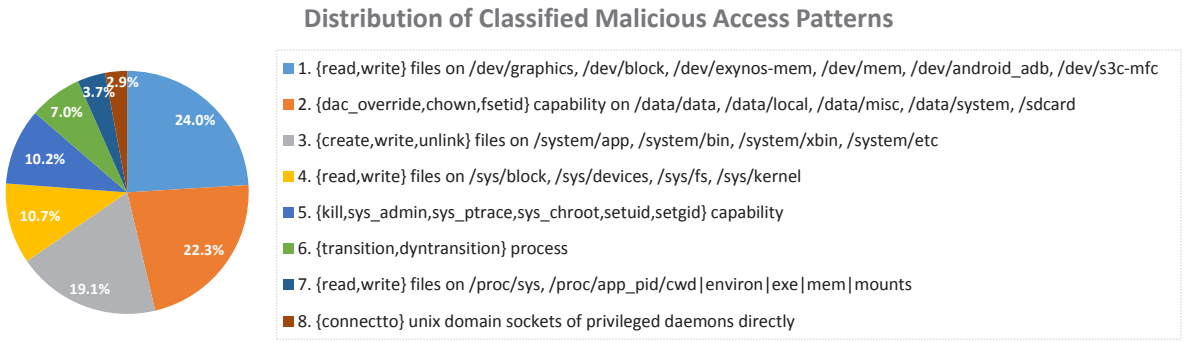


Figure 4: The distribution of malicious access patterns classified by EASEAndroid.

5.5 Case Study of Classified Malicious Access Patterns

For the one-year dataset of audit logs processed by EASEAndroid, 2,182 access patterns are classified as malicious. The reader is reminded that the starting point of analysis is 17 access patterns derived (manually) from 9 confirmed exploit kits. The access patterns newly classified as malicious by EASEAndroid capture malicious behavior much more precisely than has previously been possible. To the best of our knowledge, this is the first large-scale study of system-level malicious access patterns from real-world Android devices.

The subjects in these malicious access patterns are mostly untrusted third-party shell binaries and apps. For the purpose of understanding and discussion, they are categorized based on the permissions [5] (shown in braces) and the objects they accessed, which were mainly privileged files. Figure 4 shows the resulting 8 categories of malicious access patterns, each discussed below. Two of them (modify `/sys/fs/selinux` and `transition` to privileged domains) are new attacks in Android which directly target the SEAndroid MAC mechanism itself.

1. Exploit `/dev` nodes

The most common malicious access patterns are the ones that exploit various vulnerabilities in device nodes under `/dev`. For instance, EASEAndroid found 62 different shell binaries and exploit apps trying to directly read and write `/dev/graphics/*` (exploiting a previously-known framebuffer vulnerability). After identifying these subjects, EASEAndroid further discovered that they bundled various exploits targeting several other device nodes as well (including vendor-specific nodes). Some of these subjects were found to successfully gain root privileges. However, note that a good SEAndroid policy is still able to provide protection even on a rooted device (e.g., even `init` has limited permissions), as long as the Linux kernel is not compromised.[6]

2. Request file-related privileged capabilities

The second most frequent category of malicious access patterns are that subjects try to use privileged capabilities to modify the file mode bits and ownership of various files. This is a classic privilege escalation attack step; external binary files pushed to the device (e.g., in `/data/local/tmp`) may be given unintended capabilities, and important data files (e.g., in `/data/system`) can be made writable for attacks to proceed.

3. Modify `/system` partition

This is also a common step in exploits that the `/system` partition is modified with new binaries added such as `su`,`busybox`. Normally, the `/system` partition is mounted as read-only. But some subjects were able to remount the partition as writable. However, they were still captured by the audit logs because their domain labels were not allowed to write `system_file` under `/system`.

4. Access `/sys` filesystem

`/sys` is a virtual filesystem that exports kernel-level information to userspace, normally used by privileged system daemons. EASEAndroid found that untrusted subjects also try to directly access `/sys`, particularly `/sys/fs/selinux`, which contains the policy content and runtime state. Untrusted subjects may try to modify the policy content, either to switch to the permissive mode, or to get more permissions. We believe this is a new type of attack directly against SEAndroid MAC mechanism. Although this new attack is expected to emerge, it is still surprising to discover that the new attack has already become popular in the wild.

5. Request process-related privileged capabilities

EASEAndroid also found that some untrusted subjects ask for privileged process capabilities, such as `kill`'ing other processes, or `sys_admin` managing a list of functionalities [5]. The most common example is to use `sys_ptrace` to ptrace another process. This capability is attempted by several third-party management/moni-

[6]Since certain subjects gain root, they may be able to rollback to the permissive mode. The audit logs might just record the malicious access patterns but not actually block them.

tor apps, and game hacking apps (to modify other game apps' score/rewards).

6. Transition to privileged domains
Another new type of attack directly targeting SEAndroid is that untrusted apps try (some succeed) to gain more privileges by transforming their subject domains from `untrusted_app` to domains with higher privileges, including `init`, `init_shell`, `system_app`, and vendor daemon domains.[7] Interestingly, this case is found due to the conflict reported by the majority-vote in the semi-automated mode. An access pattern classified as malicious by both the NN classifier and the co-occurrence learner, is classified as benign by the pattern-to-rule distance measurer, because it is close to an `allow` rule. The conflict indicates that the subject carries a wrong domain.

7. Access `/proc` filesystem
Like `/sys`, `/proc` is also frequently accessed, especially by third-party management/monitor apps. Although reading `/proc/app_pid/*` might not be directly damaging, the information can be leveraged as a side-channel to compose attacks [20]. Besides, EASEAndroid also showed that certain apps try to write `/proc/sys/kernel/kptr_restrict` to gain access to the kernel symbol table, a common step in kernel exploits.

8. Connect to Unix sockets of privileged daemons
Unix domain socket is a more complicated case in SEAndroid. By design, some Unix sockets in system daemons such as `adbd`, `debuggerd` can be connected by apps, while others are reserved only for privileged daemons. EASEAndroid is able to distinguish these two cases, mainly by the co-occurrence learner. It found one new benign access pattern between two vendor daemons and several malicious ones that untrusted apps try to directly connect to Unix sockets of highly privileged daemons, such as `init`.

In summary, with EASEAndroid's learning, we find a group of interesting malicious access patterns and new attacks in Android. EASEAndroid also generates 280 fine-grained `neverallow` rules. 52 rules are found in the later policy. But others still require deeper investigation, since the knowledge learned by current EASEAndroid prototype may not be sufficient to understand the attack mechanisms behind these malicious access patterns.

6 Discussion

Blurred line between benign and malicious In practice, the line between benign and malicious might be blurred and subjective. It depends on specific security requirements and use cases to determine whether an access pattern is really benign or malicious. For example, individual users may like rooting their own devices and using the game hacking apps mentioned above, while game developers treat them as malicious because they bypass the in-app purchase. Nevertheless, EASEAndroid's learning provides more detailed evidence of access patterns' semantics for policy analysts to make the final decision.

Information missed by audit logs EASEAndroid relies on audit logs to learn new access patterns and derive policy refinements. However, two types of information could be missed or not available in audit logs, which can cause EASEAndroid to miss important knowledge. First, by default, audit logs only capture system-level operations that are denied by the policy currently loaded in a device. If the policy is too permissive or has too coarse-grained allow rules, malicious access patterns could be mistakenly allowed and missed by audit logs. To mitigate this issue, policy analysts should use `auditallow` to mark coarse-grained or uncertain rules so that audit logs can capture the operations allowed by these rules for EASEAndroid's analysis.

Second, framework-level operations are not available in audit logs, because they are controlled by Android permission model. But these upper-level operations contain valuable semantics (e.g., attack mechanisms). Without them, it is difficult to explain and distinguish certain benign/malicious access patterns in audit logs. Since Android 5.0, `logcat` is involved in SEAndroid auditing. In future, EASEAndroid can integrate logs from `logcat` to have more semantics in the knowledge base.

Countermeasure against EASEAndroid Similar to tampering virus sampling in AntiVirus programs, attackers can disable or compromise the audit log mechanism (logging and uploading) to avoid malicious access patterns being learned. Currently, we rely on Linux kernel protection [11] to ensure the integrity of audit log mechanism. And we argue that enabling audit log with policy refinement updates is a recommended security service for the majority users to have the latest security protection (or mandatory for enterprise users).

By design, if a malicious access pattern is widely spread and affects a large number of normal users, audit logs can catch the pattern for EASEAndroid to analyze. However, it is possible that isolated or targeted attacks may evade the detection of EASEAndroid if they are not popular enough to reach the thresholds, or deliberately avoid having semantic correlation with known malicious access patterns. In such cases, although EASEAndroid may leave them as unclassified, it still helps narrow down the scope for policy analysts to investigate. And policy analysts can input extra knowledge into EASEAndroid to help increase the coverage and precision.

It is also potentially possible that attackers manipulate the co-occurrence rate by intentionally forcing the be-

[7] Policy analysts suggest that it is also possible that some daemons may have zero-day vulnerabilities that are exploited to run attacks.

nign and malicious patterns to co-occur in one log, such as triggering the benign pattern first and then launching the attack. Such data poisoning attack may fool EASEAndroid's learning, which requires extra constraints or more logs from different devices to dilute the poisoned logs [14].

7 Related Work

Though SEAndroid is fairly new, SELinux has been developed and researched for years, including SELinux policy analysis and verification [10, 23, 36, 42], policy visualization [40], policy conflict resolving [26], policy simplifying [33, 34], policy comparison [19], policy information-flow integrity measurement [22, 24, 25, 37], etc. Also, the above research work usually assumes a relatively complete SELinux policy that has already been well developed. And the analysis usually focuses on stable desktop Linux system or only a few specific application programs (e.g., `sshd`, `httpd`). Due to the architecture difference, SEAndroid faces different challenges from SELinux, because current SEAndroid policy is still incomplete and under active development and continuous refinement.

In terms of SELinux policy generation, Polgen proposed by MITRE is a tool that guides policy analysts to develop policies based on system call traces [39]. However, it does not have machine learning capability and only focuses on system call traces from a single application program, which is not scalable. Madison proposed by Redhat is an extension of `audit2allow` that can generate policy similar to the reference policy style, such as using `macros` [30]. However, like `audit2allow`, it cannot create new security labels to cover new access patterns.

There is very little SELinux research related to machine learning. Marouf et al. proposed a similar approach to Polgen that analyzes system call traces to simplify SELinux policy [32]. Markowsky et al. proposed an IDS system that uses SELinux denials as input to an SVM classifier to detect attacks [31]. But there is no policy analysis or refinement.

Android SafetyNet [1] is a new security service provided by Google, which includes analyzing SELinux logs collected from Android devices, though no specific technical details about the SELinux log analysis have been disclosed.

8 Conclusion

Developing SEAndroid policies is a non-trivial task. In this paper, we have proposed EASEAndroid, the first SEAndroid audit log analytic platform for automatic policy analysis and refinement. EASEAndroid innovatively applies semi-supervised learning to MAC policy development. It has been evaluated with 1.3 million audit logs from real-world devices. It successfully discovered over 2,500 new benign and malicious access patterns, generated 331 policy rules, and found 2 new attacks in the wild directly targeting SEAndroid MAC mechanism.

Acknowledgement

We would like to thank Michael Grace, Kunal Patel and Xiaoyong Zhou from Samsung Research America for their valuable input for this paper. We also like to thank the paper shepherd and anonymous reviewers for their support to publish this paper.

This work is done in Samsung Research America. All data used to conduct the experiments was handled according to Samsung strict policies as explained in Appendix A. William Enck's work in this paper is supported by NSF grant CNS-1253346. Douglas Reeves's work in this paper is supported by ARO under MURI grant W911NF-09-1-0525. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Samsung or the funding agencies.

References

[1] Android SafetyNet, Google. `https://developer.android.com/training/safetynet`.

[2] End User License Agreement, Samsung Software. `http://www.samsung.com/us/Legal/PH_Warranty_2_EULA_INDEVICE_SINGLE_EULA.pdf`.

[3] Samsung Privacy Policy. `http://www.samsung.com/us/common/privacy.html`.

[4] Security-Enhanced Linux in Android. `https://source.android.com/devices/tech/security/selinux`.

[5] SELinux ObjectClassesPerms. `http://selinuxproject.org/page/ObjectClassesPerms`.

[6] SELinux Project. `http://selinuxproject.org`.

[7] SELinux Type Statements. `http://selinuxproject.org/page/TypeStatements`.

[8] setools, Tresys Technology. `https://github.com/TresysTechnology/setools`.

[9] Validating SELinux. `https://source.android.com/devices/tech/security/selinux/validate.html`.

[10] ALAM, M., SEIFERT, J.-P., LI, Q., AND ZHANG, X. Usage Control Platformization via Trustworthy SELinux. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security* (2008), ASIACCS '08, ACM, pp. 245–248.

[11] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), CCS '14, ACM, pp. 90–102.

[12] BADGER, L., STERNE, D., SHERMAN, D., WALKER, K., AND HAGHIGHAT, S. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium* (June 1995).

[13] BENTLEY, J. L. A Survey of Techniques for Fixed Radius Near Neighbor Searching. Tech. rep., Stanford University, Stanford, CA, USA, 1975.

[14] BIGGIO, B., CORONA, I., FUMERA, G., GIACINTO, G., AND ROLI, F. Bagging Classifiers for Fighting Poisoning Attacks in Adversarial Classification Tasks. In *Proceedings of the 10th International Conference on Multiple Classifier Systems* (Berlin, Heidelberg, 2011), MCS'11, Springer-Verlag, pp. 350–359.

[15] BULLINARIA, J. A., AND LEVY, J. P. Extracting semantic representations from word co-occurrence statistics: A computational study. *Behavior Research Methods 39*, 3 (2007), 510–526.

[16] CARLSON, A., BETTERIDGE, J., KISIEL, B., SETTLES, B., JR, E. R. H., AND MITCHELL, T. M. Toward an Architecture for Never-Ending Language Learning. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI '10.

[17] CARLSON, A., BETTERIDGE, J., WANG, R. C., MITCHELL, T. M., CARLOS, S., AND BRAZIL, S. P. Coupled Semi-Supervised Learning for Information Extraction. In *Proceedings of the third ACM international conference on Web search and data mining* (2010), WSDM '10, pp. 101–110.

[18] CHAPELLE, O., SCHOLKOPF, B., AND ZIEN, A. *Semi-Supervised Learning*. The MIT Press, Sept. 2006.

[19] CHEN, H., LI, N., AND MAO, Z. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS '09* (2009).

[20] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security '14* (2014), no. August, pp. 1037–1052.

[21] DONG, X. L., GABRILOVICH, E., HEITZ, G., HORN, W., LAO, N., MURPHY, K., STROHMANN, T., SUN, S., AND ZHANG, W. Knowledge Vault : A Web-Scale Approach to Probabilistic Knowledge Fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* (2014), KDD '14, pp. 601–610.

[22] GANAPATHY, V., JAEGER, T., AND JHA, S. Retrofitting Legacy Code for Authorization Policy Enforcement. In *2006 IEEE Symposium on Security and Privacy* (2006), S&P '06, Ieee, pp. 15 pp.–229.

[23] HICKS, B., RUEDA, S., AND CLAIR, L. S. A Logical Specification and Analysis for SELinux MLS Policy. *ACM Transactions on Information and System Security (TISSEC) 13*, 3 (2010), 1–31.

[24] JAEGER, T., SAILER, R., AND SHANKAR, U. PRIMA: Policy-reduced Integrity Measurement Architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies* (2006), SACMAT '06, pp. 19–28.

[25] JAEGER, T., SAILER, R., AND ZHANG, X. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security '03* (2003).

[26] JAEGER, T., SAILER, R., AND ZHANG, X. Resolving Constraint Conflicts. In *Proceedings of the ninth ACM symposium on Access control models and technologies* (New York, New York, USA, 2004), SACMAT '04, ACM Press, pp. 105–114.

[27] KOHAVI, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2* (San Francisco, CA, USA, 1995), IJCAI'95, pp. 1137–1143.

[28] LOSCOCCO, P., AND SMALLEY, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference '01* (2001), no. February, pp. 29–42.

[29] LOSCOCCO, P. A., SMALLEY, S. D., MUCKELBAUER, P. A., TAYLOR, R. C., TURNER, S. J., AND FARRELL, J. F. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *Proceedings of the 21st National Information Systems Security Conference* (Oct. 1998).

[30] MACMILLAN, K. Madison : A New Approach to Policy Generation. In *SELnux Symposium '07* (2007).

[31] MARKOWSKY, L. Towards Making SELinux Smart Leveraging SELinux to Protect End Nodes in a Federated Environment. Tech. rep., University of Maine, Orono, 2012.

[32] MAROUF, S., PHUONG, D. M., AND SHEHAB, M. A Learning-Based Approach for SELinux Policy Optimization with Type Mining. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (New York, New York, USA, 2010), CSIIRW '10, ACM Press, p. 70.

[33] NAKAMURA, Y. Simplifying Policy Management with SELinux Policy Editor. In *SELinux Symposium '05* (2005), pp. 1–34.

[34] NAKAMURA, Y., SAMESHIMA, Y., AND TABATA, T. SEEdit: SELinux Security Policy Configuration System with Higher Level Language. In *Proceedings of the 23rd conference on Large installation system administration* (2009), LISA '09.

[35] SALTZER, J., AND SCHROEDER, M. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9 (Sept. 1975).

[36] SASTURKAR, A., STOLLER, S. D., RAMAKRISHNAN, C. R., SCIENCE, C., AND BROOK, S. Policy Analysis for Administrative Role Based Access Control. In *19th IEEE Computer Security Foundations Workshop* (2006), CSFW '06.

[37] SHANKAR, U., JAEGER, T., AND SAILER, R. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *NDSS '06* (2006).

[38] SMALLEY, S., AND CRAIG, R. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS '13* (2013).

[39] SNIFFEN, B. T., HARRIS, D. R., AND RAMSDELL, J. D. Guided Policy Generation for Application Authors. In *SELnux Symposium '06* (2006).

[40] XU, W., SHEHAB, M., AND AHN, G.-J. J. Visualization Based Policy Analysis: Case Study in SELinux. In *Proceedings of the 13th ACM Symposium on Access control models and technologies* (2008), SACMAT '08, pp. 165–174.

[41] YIANILOS, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms* (1993), vol. 93 of *SODA*, pp. 311–321.

[42] ZANIN, G., LA, R., INFORMATICA, D., AND SALARIA, V. Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. In *Proceedings of the ninth ACM symposium on Access control models and technologies* (2004), SACMAT '04, pp. 136–145.

[43] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy* (2012), S&P '12, IEEE, pp. 95–109.

[44] ZHU, X. Semi-Supervised Learning Literature Survey. Tech. rep., University of Wisconsin - Madison, 2008.

A Data Collection Policies

Collection of audit logs used in this research strictly followed the Privacy Policy of Samsung [3], and conformed to the conditions described in Samsung's End User License Agreement [2]. Audit logs were collected anonymously from users who consented to provide diagnostic and usage data to help Samsung improve the quality and the performance of its products and services. Only Samsung authorized employees, using Samsung's internal computer systems, had access to the audit logs. No individual audit log information was released outside of Samsung while conducting the experiments described in this paper.