Triaging Android Systems Using Bayesian Attack Graphs

Yu-Tsung Lee
CSE department
Penn State University
State College, USA,
yxl74@psu.edu

Rahul George
CSE department
Penn State University
State College, USA,
rtg64@psu.edu

Haining Chen
Android Security
Google
Mountain View, USA
hainingc@google.com

Kevin Chan
DEVCOM
Army Research Lab
Adelphi, USA
kevin.s.chan.civ@army.mil

Trent Jaeger
CSE department
Penn State University
State College, USA,
trj1@psu.edu

Abstract—**Mobile computing systems, such as Android, face additional risks because their business models allow the deployment of untrusted, third-party apps. Unlike remote adversaries, these apps may exploit filesystem resources shared with more privileged apps and services to escalate privilege. Despite advancements in Android access control enforcement, adversaries continue to discover new vulnerabilities that exploit filesystem resources. A challenge is to prioritize the many privileged apps and services in an Android system for proactive vulnerability analysis against such attacks. To solve this problem, we propose a method to triage Android systems by transforming Android access control policies into Bayesian attack graphs automatically. Using the Bayesian attack graphs, we propose to prioritize programs based on their exploit probabilities (i.e., likelihood that this program may be exploited) and node centrality (i.e., importance of this program in propagating attacks). We perform a first feasibility and efficacy analysis of our approach by generating Bayesian attack graphs for Android 12 systems consisting of hundreds of applications, finding one new vulnerability and correlating recently discovered vulnerabilities. Our preliminary results show that this method offers a promising systematic approach for defenders to assess Android systems and identify the most crucial programs to test for vulnerabilities.**

Index Terms—**Attack graphs; Access control policy analysis; Graph centrality**

I. Introduction

The business model of mobile computing systems like Android [6] allows the use of many third-party applications from untrusted sources. These untrusted apps perform many functions to extend the usability of these mobile devices. To implement these functions, the untrusted apps often share access to filesystem resources with some privileged apps and services, which enables more effective collaboration to implement desired functionality.

However, sharing access to filesystem resources provides an attack surface that malicious apps may aim to exploit. Android and its OEMs have reported several CVEs that allow untrusted apps to exploit privileged apps and services to modify privileged system resources, putting the Android platform at risk. Some vulnerabilities (e.g., CVE-2023-20943 [7] and CVE-2023-21093 [8]) permit an untrusted app to modify privileged files by using a vulnerable privileged app as a *confused deputy* [35]. Several privileged apps may modify files used by even more privileged Android services and, in some cases, the Android kernel, potentially leading to system compromise. Other vulnerabilities (e.g., CVE-2022-22292 [4] and CVE-2023-21093 [8] again) enable untrusted apps to cause the execution of code of their choice to modify privileged resources even more flexibly.

To counter such threats, Android systems employ a combination of access control mechanisms [51], including *mandatory access control* (MAC) enforcement in the form of SELinux in Android [5], in addition to traditional UNIX access control. However, even with SELinux in Android, the zero-day exploits highlighted above have been found. As a result, Android systems have recently added a new defense called Scoped Storage [31], [47] to reduce attack vectors in the use of external storage (i.e., one particular filesystem partition). However, Scoped Storage is not yet applied by all apps and services [46] and does not prevent exploits on other parts of the filesystem. Thus, vulnerabilities that compromise privileged processes could still remain.

Researchers have recognized that the complexity of access control policies, in particular MAC policies, may contribute to the challenge of predicting and preventing host attacks. To address this challenge, researchers have proposed access control policy analysis [2], [39], [41]. Fundamentally, access control policy analysis methods compute the data flows among subjects and objects authorized by MAC policies. However, the number of data flows authorized by MAC policies remains large, over 100,000 for recent Android MAC policies [48]. While researchers have proposed access control policy analysis techniques to identify those data flows that may be leveraged in attacks [40], [21] or that could be implicated in data leaks [20], [83], [84], even for combinations of MAC and DAC policies [20], [36], [48], the large number of data flows limits the ability of defenders to prevent attacks.

A challenge is to prioritize the many privileged apps and services in an Android system for proactive vulnerability analysis against any attacks that may be allowed given the combination of access control policies. To achieve this goal, we leverage the insight that *attack graphs* [68], [25], [61] describe how attack goals may be achieved to triage hosts (i.e., Android devices). Twenty-plus years of research on the theory and practice of attack graphs has produced methods to reason about the exploit probabilities [64] to assess the ability to achieve attack goals in systems to compute risk measures [59], [82], [37], place defenses [61], [60], [58], [57], and perform deceptive maneuvers [53], [29], [28], [55].

While such works have advanced knowledge significantly, they have been applied primarily to generate and analyze attack graphs for known vulnerabilities in network systems. Recently, researchers have conjectured that attack graphs can be produced for hosts (e.g., Android devices) to assess attacks on zero-day vulnerabilities using access control policies [18], but we are not aware any prior methods to compute or analyze host attack graphs.

In this paper, we propose the first method to generate Bayesian attack graphs [64] automatically to triage hosts for filesystem vulnerabilities. Bayesian attack graphs represent the paths of attack operations that an attacker may exploit to achieve a given attack objective. In this paper, *attack operations* represent the ability of a less privileged program (e.g., in terms of the Google privilege levels [32]) to modify a filesystem resource accessible to a more privileged program. *Exploit probabilities* are estimated to express the likelihood that an attack operation may succeed.

We solve three key challenges to build such a method. First, we need to determine how the permissions authorized by access control policies may lead to attacks on a host. Fortunately, recent access control policy analysis work [48] for Android systems identifies the permissions that may be used in attacks on filesystem resources. We use this information to construct an attack graph structure (i.e., nodes and edges) that describes the attack paths available to adversaries given a combination of Android access control policies. Second, a challenge is to determine the likelihood that an attack will be successful in a sufficiently accurate manner. Researchers have previously found that attacks from filesystem accesses are caused by a lack of filtering [79], [14]. Thus, we investigate how the presence of filtering code may be used to estimate *exploit probabilities*, proposing a preliminary approach and suggesting future extensions. We leverage this knowledge together with the first method to generate Bayesian attack graphs [64] for hosts automatically. Third, to determine which programs have the greatest impact on achieving attack goals, we leverage methods that compute *node centrality* [75], which identifies the nodes that are most important for propagating communication in a graph. In Android systems, centrality shows which nodes are most important for connecting third-party apps (i.e., adversaries) to privileged Android processes.

To evaluate our proposed method, we construct the SHEPHERD attack graph analysis system, which converts access control policies and program's filesystem operations into attack graphs for host security analysis. The SHEPHERD system uses Android access control policies to generate the possible attack operations in an Android system. Then, the SHEPHERD system performs automated program analysis to estimate exploit probabilities to produce Bayesian attack graphs for hosts. SHEPHERD then computes node centrality over an attack graph to triage hosts by prioritizing the programs whose security is most important to the host at large. We use the results of the SHEPHERD system analysis (e.g., exploit probabilities and node centralities) to identify the programs upon which to perform vulnerability assessment. We have found that both the centrality and exploit probability provide valuable guidance in focusing vulnerability assessment efforts.

Our analysis demonstrates that the results of access control policy analysis can be translated into attack graphs comprising 176-453 nodes (representing programs) and 3,295 to 24,314 weighted edges (with weights based on exploit probabilities). Through our investigation, we identified the distribution of edges relative to Android privilege levels and discovered that Pixel3a phones exhibit a comparatively lower attack surface when it comes to untrusted apps, as compared to other OEM phones. The ranking of exploit probability further assisted in triaging which apps to test, resulting in the discovery of a zero-day vulnerability in one of the highly ranked apps. This discovery underscores the effectiveness of exploit probability as a metric. On the other hand, we also found that one of the highly ranked programs, based on node centrality, has a recently found vulnerability. This finding indicates that an adversary could easily propagate attacks to higher privileged programs, potentially leading to more severe damage without such analysis. To mitigate such scenarios, we recommend prioritizing the testing of programs with high exploit probabilities and centrality values to identify and block the propagation of potential attacks proactively.

We made the following contributions in this paper:

- We propose a method for converting access control policy analysis results into a Bayesian attack graph for filesystem attacks.
- We propose a preliminary method for using program analysis to estimate the exploit probabilities for Bayesian attack graphs.
- We compute exploit probabilities and node centrality values to generate Bayesian attack graphs for three Android 12 systems, discovering a zero-day vulnerability and highlighting the importance of blocking a recent zero-day vulnerability.

This study serves as a preliminary investigation on the generation and use of Bayesian attack graphs for Android systems. Importantly, our technique generates attack graphs from the attack operations allowed by access control, rather than from known vulnerabilities. While we believe that the method for computing attack operations from access control policies is complete for filesystem resources, our method for estimating exploit probabilities is preliminary and can be greatly improved, as we discuss in Section VIII. As the exploit probability is used to prioritize programs for vulnerability analysis directly and indirectly (i.e., via node centrality), we hope to motivate future research on this topic. Nonetheless, even with our preliminary method, we collected information sufficient to expose a zero-day vulnerability.

II. MOTIVATION

In this section, we motivate the goals of our work. We first present an example that demonstrates how unprivileged apps in Android systems may exploit authorized access to exploit privileged processes in Section II-A. Next, we describe how

TABLE I: Google's Process Privilege Levels [32]

Process Level[1]	Level Membership Requirements
Root Process (T5)	Process running with UID root
System Process (T4)	Process running with UID system
Service Process (T3)	AOSP core service providers
Trusted Application Process (T2)	AOSP default and vendor apps
Untrusted Application Process (T1)	Third-party applications
Isolated Process (T0)	Processes assumed to be under adversary control

[1] Google's process privilege levels from high to low with root processes being most privileged (T5) and isolated processes being the least privileged (T0).

the adversary performs such attacks by circumventing Android permissions in Section II-B. Finally, we examine why access control policy analysis alone is not capable of detecting multi-stage attacks.

A. An Example Attack

A recent vulnerability (CVE-2023-21093) has been identified in the MediaProvider Android service [8], which poses a significant risk by granting unauthorized access to the many privileged files accessible to this service. The MediaProvider service retrieves files on behalf of untrusted apps, and through a vulnerable code path that allows the use of "../", these untrusted apps can gain access to certain privileged files. For example, an untrusted app can use this vulnerability to replace the APK (i.e., code) files of another app. This attack enables an unprivileged app at privilege level T1 to get arbitrary execution at privilege level T3 (see Table I, described in the next section), which has broad access to resources of even more privileged processes (i.e., at T4 and T5) whose exploitation may compromise the Android system.

Such attacks are possible because higher privileged processes (e.g., pre-installed apps, system services) may utilize filesystem resources that are under the control of untrusted apps. Researchers have enumerated four types of *attack operations* that attackers may apply [48]. First, if a program uses an adversary-controlled symbolic link, it may be redirected to access files to which the adversary is not authorized, which is a *symlink attack*, a type of confused deputy attack [35]. Programs also may be prone to similar attacks that use adversary-controlled bind mounts or hard links as well. Second, a program may be lured into a link traversal attack by using adversary-controlled input to build the pathname, called a *luring attack*. Third, if a program uses a file created by an adversary, the adversary can control inputs used by that program to trigger vulnerabilities in input processing, which we call a *file attack* in this paper. Fourth, when a program expects to create a file, but an adversary creates the file first, this is called a *squatting attack*.

B. Propagating Attacks in Android

One way that Android systems aim to limit the ability of untrusted third-party apps to attack Android systems is by creating a layered defense, based on Android privilege levels [32] shown in Table I. Android systems are configured with six privilege levels, where the access control policy is designed to limit the ability for subjects at one privilege level from accessing resources controlled by subjects of a lower privilege level.

While the use of privilege levels reduces the ability of subjects at higher privilege levels from using resources controlled by subjects at lower privilege levels, it cannot eliminate such interactions. In some cases, privileged subjects provide services to untrusted apps implemented using filesystem resources, requiring that the privileged subjects are authorized to access resources controlled by untrusted apps, breaking isolation between privilege levels. A risk is that an adversary may be able to exploit a subject at a higher privilege level to propagate an attack to a subject at a yet higher privilege level, as in the example.

C. Limitations of Access Policy Analysis

Currently, researchers apply access control policy analysis [39], [42], [2] to triage systems to detect cases where such attacks are possible, but do not identify how attacks may be propagated systematically. First, some access control policy analyses compute the authorized data flows that result from a system's access control policy [2], [36], [20]. However, the fine-grained access control policies deployed in Android systems result in hundreds of thousands of data flows, so it is unclear which data flows are more relevant for attacks. Those access control analyses have been designed specifically for assessing Android systems [83], [20], [84] aim to detect malicious data leakage. While this is a real problem to consider, these analyses do not consider host compromise.

Some access control analyses have focused on identifying the specific permissions that allow attack operations [40], [21], [48]. These analyses aim to evaluate access control policies against Biba integrity [15] to detect the authorized data flows that may allow an information flow from a lower integrity subject (i.e., *adversary*) to a higher integrity subject (i.e., *victim*), which are called *integrity violations*. While only a small fraction of the authorized data flows may cause integrity violations, the number of integrity violations is still more than can be assessed manually. Recent work [48] found that nine Android systems authorized between 297 and 3,736 integrity violations — each may have multiple of the four attack operations defined in Section II-A — still too many to determine how an untrusted app may propagate an attack to a system process. Thus, it remains unclear how a defender may use such information to make choices about where to focus resources to improve defenses.

III. Threat Model

We adopt a threat model that is typical of access control analyses that compute integrity violations. We assume that any subject may modify any part of the filesystem to which they are authorized in any manner authorized. Thus, a subject with access to write a directory may add files and other filesystem objects allowed by its permissions and the filesystem partition's configuration. For example, some Android filesystem partitions prohibit the use of symbolic links. In addition, if a subject has write permission to a file, we assume that the subject may modify that file. We assume that a subject that is assigned a lower privilege level in the Google privilege levels, see Section II-B and Table I, may modify filesystem resources

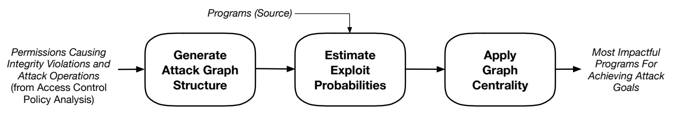

Programs (Source)
Permissions Causing Integrity Violations and Attack Operations (from Access Control Policy Analysis)
Generate Attack Graph Structure
Estimate Exploit Probabilities
Apply Graph Centrality
Most Impactful Programs For Achieving Attack Goals

Fig. 1: *System goal*: leverage the authorized permissions from available access control policies that can be attacked (*attack operations* on *integrity violations* in Section VI-A) to identify the programs whose exploitation would have the greatest impact for adversaries to reach attack goals

to which they are authorized to launch attack operations (i.e., attack operations listed in Section II-A) against subjects at a higher privilege level.

We assume that only subjects at level T1 and level T2 of the Google privilege levels may initiate an attack. Subjects at level T1 include untrusted third-party applications that we assume may initiate attacks. Subjects at level T2 may include OEM-specific applications, but these may be untrusted as well. A recent study [30] indicates that many pre-installed OEM applications lack end-to-end quality control and reuse untrusted third-party libraries. As a result, such applications may also initiate attacks. Attack paths once initiated may be propagated through subjects through confused deputy style attacks, compromising even higher privilege level processes.

We trust the Android Linux-based operating system to enforce filesystem permissions and to protect itself from compromise from apps at level T4 and below. Specifically, we trust the Android Linux-based operating system to satisfy the reference monitor concept [11].

In this paper, we do not consider whether processes at the same Google privilege level attack each other at present. Such lateral movement within privilege levels could provide other paths to achieve attack goals, but would not increase the cross-level attack operations. We discuss these implications further in Section VIII. We assume that all the initial attack targets are above privilege level T1 since its trivial to obtain level T1 permission, and therefore not valuable as target.

IV. Overview

The goal of this work is leverage available access control policies to identify the program(s) whose hardening would have the greatest impact towards preventing adversaries from reaching attack goals in Android systems. We approach this problem in three steps shown in Figure 1. First, we utilize the output of existing access control policy analysis that generates *attack operations* [48] (see Section VI-A). Using these attack operations, we define a method to construct the attack graph structure (i.e., nodes and edges) to show the attack paths in a system, described in Section V-B. Second, we define a static analysis approach for Android programs to estimate the exploit probability presented by these attack operations to construct Bayesian attack graphs [64] for hosts in Section V-C. Third, we utilize a recent *centrality* technique to compute the impact that each node in the attack graph has on enabling adversaries to reach attack goals in Section V-D. Specifically, we apply a measure based on *X-non-backtracking (X-NB) centrality* [75], which has been proposed for node immunization in networks to prevent the spreading of communications. In the context of this paper, we use X-NB centrality to estimate the importance of a program to enabling adversaries to propagate attacks.

There are two main challenges faced in this work. First, we aim to convert the results of access control policy analysis into the form of a Bayesian attack graph [64]. As far as we are aware, this is the first method to generate Bayesian attack graphs for hosts. While access control policy analysis techniques now produce valuable inputs for generating attack graphs, such as attack operations, past attack graphs techniques do not leverage access control policies nor predict the exploit probabilities used in Bayesian attack graphs. Prior work on network attack graphs leverages a third-party assessment of known program vulnerabilities, often from CVSS scores [71]. However, there are not yet broadly accepted metrics for unknown (i.e., zero-day) vulnerabilities. In this work, we propose to combine access control policy analysis results, which show which programs face which kind of attack operations, with the static analysis of programs to estimate exploit probabilities automatically. In this paper, we propose a preliminary method to estimate exploit probabilities by identifying the inputs that may be enable attacks (sources), program operations that attacked (sinks), and filtering that may prevent attacks (constraints on source-sink flows).

The second challenge is determine how to estimate the impact of a program on host compromise. Intuitively, one approach is to identify the nodes necessary to completely mediate all flows [11] from adversaries to attack goals by solving the min-cut problem. Unfortunately, this oversimplifies the problem. In general, because we cannot guarantee one program removes all the threats for downstream subjects by its own sanitization, the problem is more accurately modeled as a *directed multicut problem* [33] or *set cover problem* [57], which are NP-complete problems, rather than min-cut. Instead, we propose to apply *graph centrality* as the key measure [23] to estimate the impact of a program with respect to host compromise. Intuitively, the centrality of a node in a graph identifies the impact the removal of the node has on limiting the propagation of a communication (e.g., an infection or attack) in the graph. Such techniques are applied to a broad range of networks, including social networks and communication networks. In this case, we apply centrality to directed graphs representing Bayesian attack graphs for hosts where edges represent flows from adversaries to attack goals. A variety of methods have been proposed to compute the centrality of nodes and edges in graphs [54], [50], [65], but recent work in computing a form of centrality called *X-non-backtracking (X-NB) centrality* [75] predicts the effect of removing a node on the propagation of communications more accurately. We apply X-NB centrality for our Bayesian attack graphs along with our methods to estimate exploit probabilities to identify programs to assess for vulnerabilities in Android systems.

V. Bayesian Attack Graph Analysis

A. Attack Graph Model

In this paper, we adopt the attack graph model of Miehling et al. [53].

A Bayesian attack graph [64], $\mathcal{G}$, is defined as the (fixed) tuple $\mathcal{G} = (\mathcal{N}, \theta, \mathcal{E}, \mathcal{P})$ where

- $\mathcal{N} = 1, \ldots, N$ is the set of nodes, termed *attributes*.
- θ is the set of the node types. We assume that each non-leaf[1] attribute (node), can be one of two types, $\theta_i \in \{\wedge(AND), \vee(OR)\}$. We denote the respective sets of nodes by $N_\wedge$ and $N_\vee$
- $\mathcal{E}$ is the set of directed edges, termed *exploits*.
- $\mathcal{P}$ is the set of exploit probabilities associated with edges. Each exploit (directed edge), $e = (i, j) \in \mathcal{E}$, has an associated probability $\mathcal{P}(e) = \alpha_{ij} \in [0, 1]$.

In this work, we only generate directed, acyclic attack graphs, as in prior work [64] by relying on a monotonicity assumption that attackers will not give up capabilities that they attain [10]. This assumption is consistent with the expectation of attacks in Android systems, where each exploit (i.e., edges represent attack operations) must always target an attribute (i.e., the successor node of the edge, which is a victim program) that has a higher privilege level than the attacking attribute (predecessor node of the edge).

An attack graph contains *leaf nodes*, $\mathcal{N}_L \subseteq \mathcal{N}$, nodes without predecessors. We assume that leaf nodes at Google privilege level 1 and 2 are the adversary-controlled nodes that may initiate attacks. On the other hand, an attack graph also contains *root nodes*, $\mathcal{N}_R \subseteq \mathcal{N}$, nodes lacking successors. Such root nodes may or may not occur at the highest privilege level. As a result, some subjects that may be exploited may not enable complete host exploitation. However, such nodes may still provide worthwhile targets for the adversary. Thus, we consider all root nodes to be attack goals in the attack graph regardless of their privilege level. Miehling et al. state that a subset of the root nodes may be classified as the *critical nodes*, $\mathcal{N}_C \subseteq \mathcal{N}_R$. In this paper, we assume $\mathcal{N}_C = \mathcal{N}_R$, and also refer to these nodes as *goal nodes*.

The attack graph model also considers *node types* in θ. Node types $\wedge$ (AND) or $\vee$ (OR) dictate whether all or only one of the predecessor nodes must be controlled by an adversary, respectively, before the node comes under adversary control. For filesystem attacks studied in this paper, all the nodes are $\vee$ (OR) nodes. We discuss why this is the case in Section V-B. In addition, edges are associated with exploit probabilities, $\mathcal{P}(e)$ for edge e. In this paper, we estimate exploit probabilities for each program based on the attack operations available to adversaries given the access control policy and file system configuration using the attack surface [38] of the program relative to the attack operation as we describe in Section. In general, such probabilities should also consider the impact of the program, such as the attack surface of the program [38], as we discuss in Section V-C.

[1]Non-leaf nodes are defined as nodes with at least one predecessor.

B. Generate Attack Graph Structure

The first challenge is to develop a method to transform the results from an access control policy analysis (e.g., attack operations) into an attack graph model in the form on Definition V-A. In this section, we describe the creation of the basic structure of the attack graph, its nodes and edges.

As described in Section VI-A, an *integrity violation* in an access control policy authorizes an information flow from a low-integrity subject to a high-integrity subject through an object [48]. In Android systems, specifically, an integrity violation occurs when one subject (i.e., an adversary) that is authorized to modify (e.g., write) an object has a lower privilege level [32] than another subject (i.e., a victim) that is authorized to use (e.g., read or execute) that object. Each integrity violation identifies a situation in which adversaries may attempt attack operations on victims.

To transform these integrity violations to an attack graph, we must convert them into nodes and edges per Definition V-A for the filesystem attacks found in Android systems. In general, attack graph nodes represent the capabilities available to adversaries to attempt attacks. These are the subjects (programs) in the system that adversaries may use to attack other subjects. Integrity violations specifically identify that the subject with a lower privilege level has the permissions to attempt an attack on a subject with a higher privilege level. As a result, nodes are used to represent subjects, both the lower-privilege level subjects (i.e., adversaries) and the higher-privilege level subjects (i.e., victims).

The attack graph model associates nodes with types, either $\wedge$ (AND) or $\vee$ (OR). Node types describe the conditions in which an attack will be successful on this node (i.e., a subject) as a function of the attack operations that may be performed by its immediate predecessors (i.e., lower-privilege level subjects or adversaries). Either all (AND) of the attacks by the predecessors must succeed or any (OR) attack by a predecessor must succeed for the (successor) node to be exploited. For filesystem attacks, each adversary acts alone, leveraging its own authorized permissions to attack a victim. Regardless of the specific types of attack operations that adversaries may try to exploit an integrity violation, only one such attack operation needs to succeed for an attack to be successful. As a result, all the nodes in the attack graph are assigned the $\vee$ (OR) node type.

Edges in the attack graph represent the ability of one subject (i.e., the lower-privileged adversary) to launch attack operations against another subject (i.e., the higher-privileged victim). Access control policy analysis computes integrity violations, which identify the objects that the adversary can modify that the victim can access. Note that a particular adversary-victim pair may appear in multiple integrity violations, as an adversary may be able to exploit multiple objects used by the same victim. We describe four different types of attack operations in Section II-A. If any of those attack operations are possible using any of the integrity violations between a adversary-victim pair, an attack graph edge is added from the adversary to the victim.

C. Estimate Exploit Probabilities

A significant challenge is to estimate the exploit probabilities for the Bayesian attack graph. Traditionally, attack graphs are generated from known CVEs, which are associated with severity scores. Thus, exploit probabilities are estimated from those severity scores. However, when attack graphs are generated from integrity violations, we have no prior basis for estimating exploit probabilities. In this section, we investigate the problem of generating exploit probabilities and propose a preliminary approach. We discuss the need for further research on this problem in Section‘VIII.

One approach could be to estimate exploit probabilities based on which of the four types of attack operations (see Section II-A) may be possible between two subjects (i.e., each adversary and victim pair). However, knowing that an adversary can perform an attack operation does not tell us anything about whether this attack operation has any chance of success on a particular program. For example, a program may never use adversary input in code that builds pathnames, so luring traversal attack operations can be ruled out against that program. However, to make such judgements we need to examine the program code.

Thus, we propose to compute exploit probabilities based on the program code and how it may be prone to attack operations. The challenge is to devise a program attack model that reflects how attacks may be perpetrated. For filesystem attacks, a program may be threatened because it may create a pathname that may be used in a filesystem operation (e.g., `open`) that can be attacked (i.e., access an adversary-controlled file or use an adversary-controlled link or directory in name resolution). In our proof-of-concept system, we model the entry points that may provide inputs to build pathnames as *sources* and the filesystem operations that use pathnames as *sinks*, where one or more sources may flow to one or more sinks. Using program analysis, we can capture control and/or data flows between sources and sinks. While the preliminary implementation (see Section VI) only uses control flows, we aim to define an approach for estimating exploit probability that can be useful for either form of source-to-sink flows.

In general, the exploit probability for an attack on a filesystem operation (sink) depends on: (1) whether inputs may specify a vulnerable pathname at a source and (2) whether (control and data) program flows can propagate those inputs to a sink. If an adversary can provide input (e.g., via an intent) or the program generates a vulnerable pathname itself (e.g., corresponds to a resource in an integrity violation), then the probability of exploit is higher. To estimate (1), we define a *source factor* to represent the likelihood that an input at a source can correspond to a vulnerable pathname. Second, a program needs a flow to propagate a vulnerable input from a source to a sink, but the program may have operations that change the value and/or conditionals that may filter flows and/or values that can reach the sink. To estimate (2), we define a *flow factor*, which represents the likelihood that a flow may filter [14], [79] a vulnerable pathname input before reaching a sink. Values for both the source factors and flow factors may range from 0 (i.e., no evidence) to ∞ (i.e., definitive evidence). See Section VI for how we estimate the source and flow factors in the preliminary implementation.

As a result, we propose the following exploit probability equation: $\mathcal{P} = (1 + s)/(1 + s + f)$, where s is the source factor and f is the flow factor. Note that the exploit probability value increases as the flow factor decreases and as the source factor increases. If there is no filtering (i.e., the flow factor is 0), then the exploit probability is 1. If there is no evidence of an exploitable value at a source, we exclude the flow from consideration (i.e., $\mathcal{P} = 0$). If there is lots of evidence of an exploitable value at the source for one source-to-sink flow, then the exploit probability will approach 1.

We compute the exploit probability for each source-to-sink control flow found in the program (i.e., for all sources and sinks identified) and assign each program an exploit probability based on the maximum exploit probability for any of its source-to-sink flows. If there are no source-to-sink flows in a program, its exploit probability is 0. We use this approach because we assume that adversaries will focus on source-to-sink flows that are the easiest to exploit. Thus, even though there may be many flows, the flows that matter to adversaries are the ones with the best combination of source values and few defenses (i.e., filtering).

D. Triaging Attack Graphs Using Centrality

The goal of this work is to identify the program(s) that have the greatest impact on the propagation of filesystem attacks on a host. To do this, we want to estimate this impact for each attack graph node (i.e., program). As described in Section IV, a variety of methods have been proposed to determine where defenses should be added to thwart attacks using attack graphs. However, these approaches do not account for attack propagation [57] or depend on information that is not available for assessing the impact of zero-day attacks on hosts [53], [29], [28], [55].

Instead, we propose to identify the program most in need of defense in a host based on the host attack graph using its *centrality* relative to epidemic thresholds [50]. In general, centrality is a measure of the ”influence” of a node on a system of interconnected components derived from its connectivity, although the impact of such influence depends on the semantics of the interconnections. For example, in network epidemiology, the epidemic threshold is the number or density of nodes that is required for an epidemic to occur. Infected nodes with a greater centrality have a greater connectivity, and thus have a greater impact on the spread of the infection to epidemic levels. Thus, an aim is to remove nodes that would increase the epidemic threshold of a network to reduce the likelihood of epidemic occurring. By computing each node’s centrality to a system, we can identify the key parties who we want to prevent becoming infected to lower the risk of an epidemic. In this paper, the claim is that increasing the ”epidemic threshold” of an Android system the most reduces the risk of catastrophic host compromise the most by preventing the attack options from reaching epidemic scope in the system.

Researchers have found that computing the non-backtracking (i.e., no edge is traversed twice in succession) centrality [50], [65] is a useful measure for identifying the node that increases the epidemic threshold by the greatest amount. Researchers claim that the recent X-NB centrality measure ”approximates the true effect of a node’s removal in the epidemic threshold” [75]. In our attack graphs, this measure estimates the impact of each program to enable other attacks to become possible (i.e., based on each’s exploit probabilities) in the system.

VI. Implementation

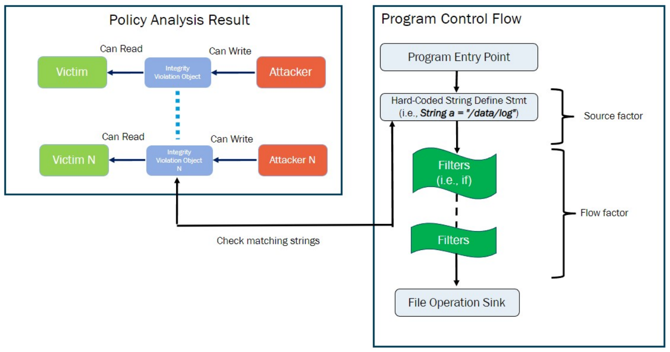
We have built the Shepherd attack graph system to implement the steps shown in Figure 1. Shepherd stores attack graphs according to the model of Definition V-A. First, Shepherd uses the integrity violations and attack operations generated by the open-source PolyScope access control analysis system[2] [48] to generate host attack graphs. Second, Shepherd employs Java static analysis tools to estimate exploit probabilities. Third, Shepherd leverages an open-source implementation [49] of the approximate X-NB centrality procedure [75] to estimate centrality for attack graph nodes. Shepherd stores attack graphs in a Neo4j graph database [1]. Shepherd also provides an event-based method to modify attack graphs incrementally based on system changes, such as the addition of new programs, new permissions, and new attack surfaces (i.e., leaf nodes from which attacks may be initiated), but those features are beyond the scope of this paper.

First, Shepherd computes the attack graph structure as described in Section V-B. We use PolyScope to compute the authorized integrity violations and their attack operations for Android systems from a combination of Android access control policies, including SELinux in Android mandatory access control, UNIX discretionary access control, and specialized policies (e.g., Android permissions and Android Scoped Storage [47]). We format this information to enable the NetworkX library [34] to create graphs satisfying the attack graph model of Definition V-A encoded in graphML.

Second, Shepherd employs a methodology outlined in Section V-C to compute exploit probability for Android APK programs. To achieve this, we design a static analysis using a combination of Soot [78], Flowdroid [12], [13], and TIRO [85] to identify sources, sinks and filtering operations. Initially, Shepherd utilizes Soot to lift code within the APK file into Jimple intermediate representation (IR), which Flowdroid operates on. Subsequently, we leverage Flowdroid to handle lifecycle-related control flow transfers and construct an accurate control flow graph (CFG) for the Android program. By using Flowdroid’s entrypoint analysis, we identify two types of sources: inter-app sources and intra-app sources. Inter-app sources refer to application entry points that can be invoked by Android IPCs, which may carry adversarial input from external apps that could potentially be used to construct pathnames, as in *luring attacks* (see Section II-A). Intra-app sources encompass Android lifecycle components

[2]PolyScope is available at https://github.com/yxl74/PolyScope.

Fig. 2: Exploit Probability Implementation



(e.g., onCreate(), onClick()) that are integral to the normal execution of the application. Exploitation of intra-app sources requires the victim app to use pathnames corresponding to integrity violation resources, as in CVE-2020-13833 [3]. Intra-app sources may be exploited in *symlink attacks*, *squatting attacks*, and *file attacks* (see Section II-A).

The outline of our implementation is shown in Figure 2. We designate Java file open method signatures as sinks and identify source-to-sink control flows. We included methods from *java.io* and *java.nio*, consisting of 24 methods related to file opening, 12 methods related to file reading, and 15 methods related to file writing. We did not include third-party libraries file operations at present. Then, we leverage TIRO’s call graph traversal component to perform a depth-first search on the application call graph to locate all control flow paths from the identified sources to the designated sinks.

Given these targeted flows from sources to sinks, Shepherd provides a proof-of-concept approach to estimate the source and flow factors s and f, respectively (see the *exploit probability equation* in Section V-C). First, we assign a non-zero source factor when a source is assigned a hard-coded string value that is a substring of a pathname of an integrity violation object. In such cases, we assume there is some probability that the program intends to create an unsafe pathname. Shepherd gathers all strings defined along a source-to-sink flow that contain the ”/” character. Then, we use string matching to identify all hard-coded strings that are a substring of a pathname of an integrity violation object as shown in Figure 2. The source factor is assigned to the number of characters in the maximal matching substring on a flow (i.e., typically, there is only one string assigned on a flow), not including ”/” characters. Second, since filtering is often implemented using conditional checks, we currently estimate the flow factor as the number of conditional checks along a source-to-sink flow (e.g., IfStmt in Soot).

Third, Shepherd utilizes an open source implementation [49] of the approximate X-NB centrality procedure [75] to estimate the impact of each attack graph node (i.e., programs) on achieving attack goals as described in Section V-D. We compute the X-NB centrality of the attack graph by leveraging the auxiliary non-backtracking matrix for the attack graph as described in recent work [74]. Additionally, we use the

TABLE II: Computed Attack Graph Sizes (Nodes and Edges)

	Pixel 3a 12.0	OnePlus 8T 12.0	Galaxy S20 12.0
Nodes	176	440	453
Adversary	68	86	50
Goal	36	133	101
Edges	3,295	17,033	24,314

Python based SciPy library [81] to perform the underlying mathematical computations, such as computing the largest eigenvalue.

A. Limitations

We posit that our proof-of-concept methodology captures the essence of estimating the likelihood of successful exploitation. We acknowledge that both the flow and source factors in our approach can be further refined.

In terms of flow factor, there are techniques other than conditional checks that can be used to defend against malicious inputs. We envision the future use of symbolic execution techniques for Android programs to generate path constraints to assess program filtering. Angr [69] recently added the capability to execute Android applications symbolically. However, it requires users to patch the behavior of inter-procedural methods manually.

Regarding the source factor, our future work will explore how to apply techniques in *string analysis* [], which is a static analysis technique to determine the possible values a string object may take at particular program points. Currently, we use intents and hard-coded strings as sources, but these may be modified by the program, essentially creating new sources to consider.

VII. Evaluation

For the evaluation, we use attack graphs constructed from three different Android devices, including Pixel3a from Google, Galaxy S20 from Samsung, and 8T from Oneplus. We evaluate Android 12 versions for each of these devices. We run the PolyScope tool [48] to generate integrity violations and attack operations for fresh firmware images of each device and use SHEPHERD to construct attack graphs in graphML format.

A. Host Attack Graph Flows

Table II summarizes information about the computed attack graph flows. The first three rows list the counts related to the number of nodes in the attack graphs for each of the three systems evaluated. The row *Nodes* lists the total number of nodes in each attack graph. The *Adversary* and *Goal* nodes list the number of adversary-controlled nodes from which attacks may originate in the attack graph and the number of attack goal nodes that are the ultimate targets of attacks, respectively. In this case, these counts equal the number of leaf and root nodes[3], respectively, in the attack graph as defined in Section V-A. The fourth row shows the number of *Edges* total in the attack graphs for the three systems.

An important consideration is how many options are available to an adversary to escalate their privilege from one

[3]Recall that only leaf nodes at level 1 and 2 may be adversary-controlled initially.

TABLE III: Distribution of Cross-Privilege Level Edges (Number of Edges)

	Pixel 3a 12.0	Oneplus8T 12.0	Galaxy S20 12.0
T1 → T2[1]	1,068	7,555	8,003
T1 → T3	273	731	623
T1 → T4	142	1,653	1,615
T1 → T5	108	164	71
T2 → T3	600	1,528	3,981
T2 → T4	305	2,825	7,415
T2 → T5	246	484	1,082
T3 → T4	84	340	634
T3 → T5	117	105	114
T4 → T5	352	1,648	776

TABLE IV: Exploit Probability Stats

	Pixel 3a 12.0	OnePlus 8T 12.0	Galaxy S20 12.0
Programs[1]	50	144	245
Level 2	43	92	177
Level 3	4	7	1
Level 4	3	45	67
Level 1	21	79	44
C-Programs	107	209	144
Time-Out	19	18	20

[1] Total programs analyzed
We do not analyze Level 1 apps, as they are leaf nodes.
We leave analyzing C programs as future work
Level 1 programs are leaf nodes

privilege level to another. Table III shows the distribution of the number of edges among each pair of privilege levels in the attack graphs for each of the three systems. An edge is included in a count for a pair of privilege levels (x, y), where x is the lower privilege level and y is the higher privilege level, when the adversary's level is x and victim's level is y.

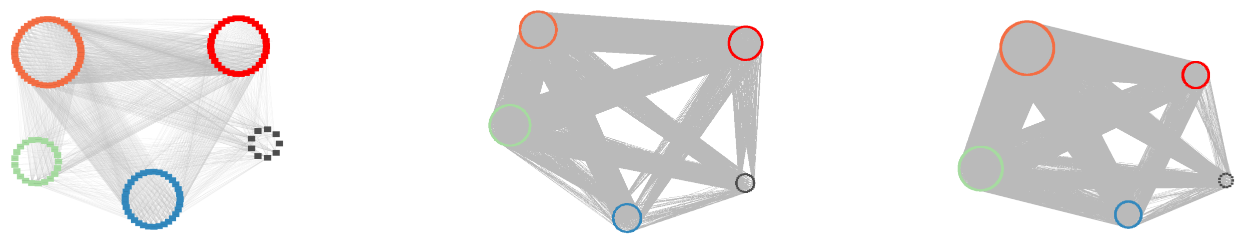
Figure 3 shows three attack graph visualizations by grouping the nodes from the same privilege level together. Across all three systems, we can see that the Pixel3a Android 12 systems do not have as many cross-level edges in Figure 3(a). This suggests that the Pixel3a Android 12 system is better protected compared to Samsung Android 12 and OnePlus Android 12 systems in Figure 3(b) and Figure 3(c), respectively. For example, the connectivity between level 2 nodes to level 3/4 nodes are much less on the pixel3a system. Recall that level 2 nodes may also initiate attacks as described in the threat model in Section III.

B. Exploit Probability

We perform analysis on 439 Java programs collected from three phones, distributed among the systems and privilege levels as shown in Table IV. Given that we are conducting static analysis on a substantial volume of programs, we have established a timeout duration of 300 seconds for each static analysis. The number of programs that exceeded this timeout is indicated in the bottom row of Table IV. Upon analysis, it becomes evident the that majority of Java programs are level 2 programs. These programs, being pre-installed apps implemented by the OEMs, normally have privileges that third-party applications (i.e., Level 1 programs) cannot obtain directly (e.g., signature-level Android permission). Therefore, it is important to evaluate their exploit probability and assess their ability to prevent exploitation.

Figure 4 shows the exploit probability distribution for programs from all three systems tested. The data reveals that

Fig. 3: Attack Graph Visualization



(a) Pixel3a Android 12 (b) Oneplus8T Android 12 (c) Galaxy S20 Android 12

Red: Level 1; Orange: Level 2; Green: Level 3; Blue: Level 4; Black: Level 5

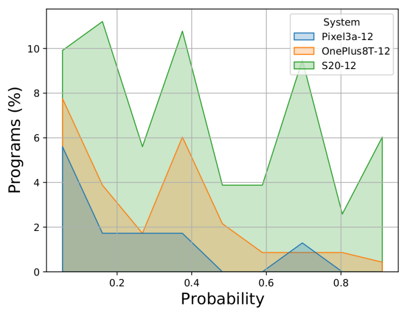
Google phones generally exhibit lower probabilities compared to other OEM phones, indicating a potential correlation with the effective utilization of Android file APIs. However, it is important to note that our static analysis did not consider file access through providers and the Android storage access framework [9] as a sink. These APIs either require user consent or is protected by Android's FileProvider class which has mechanisms to prevent path traversal attacks. Consequently, applications that employ these mechanisms for file access are likely to yield lower probabilities of exploits. For the two OEM phones, notable spikes in probability occur at approximately 0.2, 0.4, and 0.7. This observation supports our hypothesis that the functionality of apps influences their file usage. There will be apps with minimal file usage, apps with mild file usage and apps with heavy file usage. Apps with heavy file usage are more likely to have control flow paths that leads to file operations without many control flow transfers (i.e., app components directly operate on file pathnames).

We show the programs with highest exploit probabilities in Table V. The program *com.sec.android.app.servicemodeapp*, ranked #19 out of 214 programs with probability of 0.73, has a previously reported symlink vulnerability [48]. In addition to the aforementioned vulnerability, we found and reported a zero-day vulnerability on *com.oneplus.findmyphone* app, ranked #20 out of 144 programs with probability of 0.46, which we will discuss in detail in later section. The identification of these vulnerabilities highlights how exploit probability can help with prioritizing testing efforts and provide meaningful edge weights for centrality calculation. To enhance the robustness of our preliminary study, further investigation is needed into the applications that our model ranks as highly vulnerable. This would validate whether they are indeed susceptible to exploits. Moreover, cross-referencing our findings with existing CVE data could provide valuable insights into the effectiveness of our static analysis-based exploit probability metric in identifying apps with severe vulnerabilities.

C. Triaging with Centrality

We examine the impact of the centrality approach on the generated attack graphs. Table VI shows the programs associated with the top-5 nodes found computing the approximate X-NB centrality procedure [75] of the attack graph for each system.

Fig. 4: Exploit Probability Distribution Across Programs



On the Oneplus 8T phone, we found that the #16 ranked centrality program com.nearme.play has a recently reported vulnerability [46]. This program is running as privileged app and stores game related executable files in legacy external storage location where adversary could modify. This vulnerability poses a significant risk as a malicious code execution exploit in a centralized node can cause severe damage due to its strong connections with privileged programs. This finding highlights the critical importance to test highly centralized programs for potential vulnerabilities.

As shown in Section VII-B, exploit probability offers valuable insight on security of individual program. In addition, graph centrality offers insight into the severity of an exploit when a program is compromised. We performed triaging by combining graph centrality and exploit probability to prioritize testing efforts. We first identify programs with an exploit probability above a statistical threshold (i.e., one standard deviation above the mean exploit probability), then begin penetration testing starting from programs with highest centrality rankings. Although rather straightforward, the problematic programs we highlighted shows the potential of such a hybrid approach.

D. Vulnerability Discussion

We discuss the vulnerabilities mentioned in Section VII-B, which is a new vulnerability in the *findmyphone* application running as privileged app on Oneplus8T Android 12 devices. We give priority to testing due to its higher probability ranking and the utilization of files located in well-known problematic legacy locations, as indicated by the results of our static analysis. In one of the app components *com.baidu.location*, the application reads files from the */sdcard/backup/.SystemConfig* directcory, which is located in legacy location of external stor-

TABLE V: Top-5 Ranked Subjects Based on Exploit Probability

	Pixel 3a 12.0	**Oneplus8T 12.0**	**Galaxy S20 12.0**
1	L2: com.android.managedprovisioning	L2: com.oneplus.mms	L2: com.samsung.android.messaging
2	L2: com.google.android.apps.tips	L2: com.android.packageinstaller	L4: com.samsung.android.vtcamerasettings
3	L2: com.google.android.packageinstaller	L4: com.oneplus.coreservice	L2: com.sec.android.app.camera
4	L2: com.google.android.markup	L2: com.google.android.providers.media.module	L2: com.sec.android.gallery3d
5	L2: om.google.android.permissioncontroller	L2: net.oneplus.launcher	L2: com.samsung.android.scloud

TABLE VI: Top-5 Ranked Subjects Using X-NB Centrality for All Test Options

	Pixel 3a 12.0	**Oneplus 8T 12.0**	**Galaxy S20 12.0**
1	L2: com.google.android.apps.gcs	L2: com.android.packageinstaller	L2: com.android.shell
2	L2: com.google.android.projection.gearhead:projection	L4: com.oneplus.coreservice	L2: com.samsung.android.scs
3	L2: com.google.android.apps.pixelmigrate	L4: com.oneplus.filemanager	L1: com.google.android.youtube
4	L2: com.google.android.apps.wearables.maestro.companion	L2: net.oneplus.launcher	L4: android.hardware.sensors@2.0-service.multihal
5	L2: com.google.android.apps.nbu.files	L2: com.oneplus.opbugreportlite	L2: com.samsung.android.app.notes

age (e.g., not protected by the Scoped Storage defense [31]). Untrusted applications with the legacy storage flag or the ”manage external storage” permission can squat the *.cuid* file in the vulnerable directory and provide malicious input file to the application. We suspect that the cuid file contains user identification number for the Baidu location database. By providing incorrect UID, attacker could perform a denial-of-service attack to the security sensitive phone finding application. We have reported this vulnerability to Oneplus through their vulnerability reporting portal.

VIII. Discussion

The evaluation shows that Android access control policies can be converted into an attack graph format that enables identification of the Android subjects (programs) that may be prone to exploitation (i.e., have high exploit probabilities) and/or be central to adversaries attack goals (i.e., propagate communications impacting privileged processes). That several new and/or recent vulnerabilities were found in highly ranked programs indicates that the examining programs from these perspectives can be helpful in focusing efforts to detect latent vulnerabilities. Without such techniques, there is no real basis for vetting any programs, so latent vulnerabilities may not be discovered.

The methods proposed here for exploit probability estimation are admittedly preliminary, proof-of-concept approaches. We find the source factor estimate based on the detection hard-coded strings that match substrings in integrity violations was useful in identifying real problems. The method for estimating the filtering factor using conditionals (i.e., control-flow decisions) has potential for improvement. In the future, we envision exploring both operations that impact data flows, such as operations that compute values used in file system operations. In addition, we will explore the use of path constraints associated with the data used in filesystem operations collected from symbolic execution. We also plan to expand the breadth of our analysis by including C programs using static analysis tool like LLVM [45].

Further, we plan to explore automated testing techniques for these classes of vulnerabilities, akin to fuzz testing for filesystem vulnerabilities. Prior work has explored methods for the filesystem to generate filesystem attacks automatically [80], but such methods cannot induce progrms to perform vulnerable file system operations. Fuzz testing [52] is capable of driving the program to the unsafe filesystem operations and solving sets of constraints to generate inputs that may imply attacks. We will explore automating the testing of programs identified via triaging in the future.

We also want to further examine how to combining exploit probability and centrality together to produce better triaging. Some of the top exploit probability nodes have only a modest impact on propagating attacks (i.e., not be very highly ranked for centrality). From Figure 3, we can see that Samsung S20 and OnePlus 8T have a wide set of attack edges between privilege levels. Thus, even for the most vulnerable nodes, they may only have a modest impact on the propagation of attacks. Perhaps we can introduce some weighted mechanisms to better fuse the two metrics.

IX. Related Work

Filesystem Security Researchers have long known about filesystem attacks, but have found it difficult to prevent programs from falling victim to such attacks. A variety of defenses have been proposed as program or library extensions [24], [27], [76] or kernel extensions [73], [19], [77]. Researchers have found that defenses from either perspective (i.e., program/library or kernel) are limited because they have an incomplete view of the other perspective [17]. As a result, the main defense for preventing filesystem vulnerabilities is access control. Researchers have proposed MAC enforcement for Android systems [86], [16], leading to the deployment of SEAndroid [72] and subsequent defenses [47]. However, the attack operations we consider in this paper abuse available MAC and DAC permissions.

Access Control Risk. Researchers have proposed access control models that leverage risk in the access control decision process [63], [43], [22], [62], [67], [56], [66]. Chen and Crampton [22] propose to estimate the risk regarding subject trustworthiness and the appropriateness of permission assignments and utilize such estimates in authorizing operations. Bijon *et al.* [43] examine which relationships in an RBAC policy may be made risk-aware and examine how to apply metrics and constraints to express risk. Petracca *et al.* [63] provide methods for estimating risk in access control enforcement as it is accumulated at runtime. However, producing metrics, constraints, and estimates for risk involves a significant amount of subjectivity.

Attack Graphs. A wide variety of research in attack graphs has been undertaken since early papers from the mid-1990s [25], [26]. Early work on utilizing attack graphs to configure defenses leveraged graph mediation problems, such as min-cut and set cover [57]. While providing mediation is a start, cuts do not account for how mediation may be incomplete, allowing some threats to be propagated. More recent work focuses on enabling defenders to compete in attack graph games through the placement of sensors dynamically based on tracking adversary strategies [53], [29], [28], [55], [44]. Determining adversary strategies for exploiting zero-day vulnerabilities is more difficult to determine. In this paper, we leverage the relative attack capabilities of the attack operations available, but in general, more information about the victim is desirable. In addition, the focus of prior work has been network attack graphs or reachability games in general, rather than host attack graphs.

Centrality and Immunization. Researchers have proposed several definitions of centrality based on the NB-matrix [54], [50], [65], [75]. Such work has been applied to problems of identifying the nodes with the most influence [54] and immunizing the graph against influence [75]. Researchers have also proposed other techniques for immunizing graphs [23]. We leverage a technique that reduces the epidemic threshold [70], [75] to reduce the breadth of attacks available to adversaries. These techniques focus on decreasing the largest NB-eigenvalue instead since that approach can provide a tighter bound to the true epidemic threshold in certain cases.

X. Conclusions

Despite the widespread utilization of access control mechanisms in modern Android systems, multi-stage attacks still persist and pose challenges for effective defense. In this paper, we propose a triaging method that accurately identifies programs with significant security impact on the host system. Our method stands out by constructing comprehensive attack graphs combining policy analysis, program static analysis, and graph analysis techniques, which compute a prioritization of vulnerable testing efforts. We applied our method to three different devices running the most widely used Android version and observed that some programs highlighted by our technique exhibited recent vulnerabilities as well as the discovery of a new zero-day vulnerability. Although the presented implementation is preliminary, it highlights that our proposed technique can provide valuable insight on identifying the most impactful programs for vulnerability testing.

Acknowledgments

We would like to thank our anonymous reviewers, especially our shepherd Sebastien Bardin, for the invaluable guidance on revision of this paper and their insightful feedback. This research was sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2- 0045 (ARL Cyber Security CRA) and National Science Foundation grant CNS-1816282. Any views, opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory of the U.S. government. The U.S. government is authorized to reproduce and distribute reprints for government purposes notwithstanding any copyright notation here on.

References

[1] Neo4j. https://neo4j.com/.

[2] SETools. Accessed Dec 2019. URL: https://github.com/TresysTechnology/setools.

[3] Samsung Security Update, June 2020. URL: https://security.samsungmobile.com/securityUpdate.smsb.

[4] Samsung Security Update, February 2022. URL: https://security.samsungmobile.com/securityUpdate.smsb.

[5] Security-Enhanced Linux in Android, Sep 2022. URL: https://source.android.com/docs/security/features/selinux.

[6] Android Open Source Project, April 2023. URL: https://source.android.com/.

[7] Android Security Bulletin-April 2023, Feb 2023. URL: https://source.android.com/docs/security/bulletin/2023-02-01.

[8] Android Security Bulletin-April 2023, April 2023. URL: https://source.android.com/docs/security/bulletin/2023-04-01.

[9] Open Files Using Storage Access Framework, Feb 2023. URL: https://developer.android.com/guide/topics/providers/document-provider.

[10] Paul Ammann, Duminda Wijesekera, and Saket Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 217–224, 2002.

[11] James P Anderson. Computer security technology planning study. Technical report, ANDERSON (JAMES P) AND CO FORT WASHINGTON PA FORT WASHINGTON, 1972.

[12] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, jun 2014. doi:10.1145/2666356.2594299.

[13] Steven Arzt, Siegfried Rasthofer, Robert Hahn, and Eric Bodden. Using targeted symbolic execution for reducing false-positives in dataflow analysis. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, page 1–6, New York, NY, USA, 2015. Association for Computing Machinery. URL: https://doi.org/10.1145/2771284.2771285.

[14] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 387–401. IEEE Computer Society, 2008.

[15] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

[16] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Network and Distributed System Security Symposium(NDSS)*, 2012.

[17] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *Proceedings of 2009 IEEE Symposium on Security and Privacy*, 2009.

[18] Frank Capobianco, Rahul George, Kaiming Huang, Trent Jaeger, Srikanth Krishnamurthy, Zhiyun Qian, Mathias Payer, and Paul Yu. Employing attack graphs for intrusion detection. In *Proceedings of the New Security Paradigms Workshop*, pages 16–30, 2019.

[19] Suresh Chari, Shai Halevi, and Wietse Venema. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *Proceedings of the 17th Network and Distributed System Security Symposium(NDSS)*, 2010.

[20] Haining Chen, Ninghui Li, William Enck, Yousra Aafer, and Xiangyu Zhang. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[21] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the 16th Network and Distributed System Security Symposium(NDSS)*, pages 11–16, 2009.

[22] Liang Chen and Jason Crampton. Risk-aware role-based access control. In *Proceedings of the 7th International Workshop on Security and Trust Management*, pages 140–156, 2011.

[23] Chen Chen and Hanghang Tong and B. Aditya Prakash and Charalampos E. Tsourakakis and Tina Eliassi-Rad and Christos Faloutsos, and Duen Horng Chau. Modeling security threats. *IEEE Transactions on Knowledge and Data Engineering*, 28(1):113–126, 2016.

[24] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-hartman. RaceGuard: Kernel Protection from Temporary File Race Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[25] Marc Dacier and Yves Deswarte. Privilege graph: an extension to the typed access matrix model. In *European Symposium on Research in Computer Security*, pages 319–334. Springer, 1994.

[26] Marc Dacier, Yves Deswarte, and Mohamed Kaâniche. Models and tools for quantitative assessment of operational security. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 177–186. Springer, 1996.

[27] Drew Dean and Alan Hu. Fixing Races for Fun and Profit. In *Proceedings of the 13th conference on USENIX Security Symposium*, 2004.

[28] Karel Durkota, Viliam Lisý, Branislav Bosanský, and Christopher Kiekintveld. Optimal network security hardening using attack graph games. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 526–532, 2015.

[29] Karel Durkota, Viliam Lisý, Christopher Kiekintveld, Branislav Bosanský, and Michal Pechoucek. Case studies of network defense with attack graph games. *IEEE Intelligent Systems*, 31(5):24–30, 2016.

[30] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. An Analysis of Pre-installed Android Software. *arXiv preprint arXiv:1905.02713*, 2019.

[31] Google. Storage Updates in Android 11. Accessed June 2020. URL: https://developer.android.com/preview/privacy/storage.

[32] Google. Security Overview, 2019. Accessed Jan. 10, 2020. URL: https://source.android.com/security/overview/updates-resources#process_types.

[33] Anupam Gupta. Improved results for directed multicut. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2006.

[34] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[35] Norm Hardy. The Confused Deputy: or Why Capabilities Might Have Been Invented. *ACM Special Interest Group in Operating Systems, Operation System Review*, 22(4), 1988.

[36] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *Proceedings of the USENIX Security Symposium*, 2020.

[37] John Homer, Su Zhang, Xinming Ou, David Schmidt, Yanhui Du, S. Raj Rajagopalan, and Anoop Singhal. Aggregating vulnerability metrics in enterprise networks using attack graphs. *Journal of Computer Security*, 21(4):561–597, 2013.

[38] Michael Howard, Jon Pincus, and Jeannette M Wing. Measuring relative attack surfaces. In *Computer security in the 21st century*, pages 109–137. Springer, 2005.

[39] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Managing Access Control Policies Using Access Control Spaces. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, page 3–12, New York, NY, USA, 2002.

[40] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12^{th} USENIX Security Symposium*, 2003.

[41] Trent Jaeger and Jonathon E Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2):158–190, 2001.

[42] Trent Jaeger, Xiaolan Zhang, and Antony Edwards. Policy management using access control spaces. *ACM Transaction on Information and System Security*, 6(3):327–364, 2003.

[43] Khalid Zaman Bijon, Ram Krishnan, and Ravi S. Sandhu. A framework for risk-aware role based access control. In *Proceedings of the 2013 IEEE Conference on Communications and Network Security*, pages 462–469, 2013.

[44] Abhishek Ninad Kulkarni and Jie Fu. Synthesis of deceptive strategies in reachability games with action misperception. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 217–223, 2020.

[45] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[46] Yu-Tsung Lee, Haining Chen, William Enck, Hayawardh Vijayakumar, Ninghui Li, Zhiyun Qian, Giuseppe Petracca, and Trent Jaeger. Polyscope: Multi-policy access control analysis to triage android scoped storage, 2023. `arXiv:2302.13506`.

[47] Yu-Tsung Lee, Haining Chen, and Trent Jaeger. Demystifying Android's Scoped Storage Defense. *IEEE Security & Privacy*, 19(5), 2021.

[48] Yu-Tsung Lee, William Enck, Haining Chen, Hayawardh Vijayakumar, Ninghui Li, Daimeng Wang, Zhiyun Qian, Giuseppe Petracca, and Trent Jaeger. Polyscope: Multi-policy access control analysis to triage android systems. In *Proceedings of the 30th USENIX Security Symposium*, August 2021.

[49] Leo Torres. Node Immunization with Non-backtracking Eigenvalues, July 2017. (Accessed June 2019). URL: https://github.com/leotrs/inbox.

[50] Travis Martin, Xiao Zhang, and Mark E. J. Newman. Localization and centrality in networks. *Physics Review E*, 90(052808), 2014.

[51] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model, 2019.

[52] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[53] Erik Miehling, Mohammad Rasouli, and Demosthenis Teneketzis. Optimal defense policies for partially observable spreading processes on bayesian attack graphs. In *Proceedings of the second ACM workshop on moving target defense*, pages 67–76, 2015.

[54] Flaviano Morone and Hernan A.Makse. Influence maximization in complex networks through optimal percolation. *Nature*, 524(7563), 2015.

[55] Thanh Hai Nguyen, Mason Wright, Michael P. Wellman, and Satinder P. Singh. Multistage attack graph security games: Heuristic strategies, with empirical game-theoretic analysis. *Secure Communication Networks*, 2018.

[56] Qun Ni, Elisa Bertino, and Jorge Lobo. Risk-based access control systems built on fuzzy inferences. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 250–260, 2010.

[57] Steven Noel and Sushil Jajodia. Optimal IDS sensor placement and alert prioritization using attack graphs. *Journal of Network and Systems Management*, 16(3):259–275, 2008.

[58] Steven Noel, Eric Robertson, and Sushil Jajodia. Correlating intrusion events and building attack scenarios through attack graph distances. In *20th Annual Computer Security Applications Conference*, pages 350–359. IEEE, 2004.

[59] Steven Noel, Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Measuring security risk of networks using attack graphs. *International Journal of Next Generation Computing*, 1(1), 2010.

[60] Xinming Ou, Wayne F Boyer, and Miles A McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345, 2006.

[61] Xinming Ou, Sudhakar Govindavajhala, and Andrew W Appel. Mulval: A logic-based network security analyzer. In *USENIX security symposium*, volume 8, pages 113–128. Baltimore, MD, 2005.

[62] Pau-Chen Cheng, Pankaj Rohatgi, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Schuett Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *Proceedings*

of the 2007 IEEE Symposium on Security and Privacy, pages 222–230, 2007.

[63] Giuseppe Petracca, Frank Capobianco, Christian Skalka, and Trent Jaeger. On risk in access control enforcement. In *Proceedings of the 22nd ACM Symposium on Access Control Models and Technologies*, June 2017.

[64] Nayot Poolsappasit, Rinku Dewri, and Indrajit Ray. Dynamic security risk management using bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing*, 9(1):61–74, 2011.

[65] Filippo Radicchi and Claudio Castellano. Leveraging percolation theory to single out influential spreaders in networks. *Physics Review E*, 93(062314), 2016.

[66] Farzad Salim, Jason Reid, Ed Dawson, and Uwe Dulleck. An approach to access control under uncertainty. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security*, pages 1–8, 2011.

[67] Savith Kandala, Ravi S. Sandhu, and Venkata Bhamidipati. An attribute-based framework for risk-adaptive access control models. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security*, pages 236–241, 2011.

[68] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, 2002.

[69] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[70] Munik Shrestha, Samuel V. Scarpino, and Cristopher Moore. Message-passing approach for recurrent-state epidemic models on networks. *Physics Review E*, 92(022821), 2015.

[71] Anoop Singhal and Xinming Ou. Security risk analysis of enterprise networks using probabilistic attack graphs. Technical Report NIST Interagency Report 7788, National Institute of Standards and Technology, 2011.

[72] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the 20th Network and Distributed Systems Symposium (NDSS)*, 2013.

[73] Kyung suk Lee and Steve J. Chapin. Detection of File-based Race Conditions. *International Journal of Information Security*, 2005.

[74] Leo Torres, Kevin S Chan, Hanghang Tong, and Tina Eliassi-Rad. Node immunization with non-backtracking eigenvalues. *arXiv preprint arXiv:2002.12309*, 2020.

[75] Leo Torres, Kevin S Chan, Hanghang Tong, and Tina Eliassi-Rad. Nonbacktracking eigenvalues under node removal: X-centrality and targeted immunization. *SIAM Journal on Mathematics of Data Science*, 3(2):656–675, 2021.

[76] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *USENIX Conference on File and Storage Technologies*, 2008.

[77] Eugene Tsyrklevich and Bennet Yee. Dynamic Detection and Prevention of Race Conditions in File Accesses. In *USENIX Security Symposium*, 2003.

[78] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. CASCON '99, page 13. IBM Press, 1999.

[79] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting Resource Access by Inferring Programmer Expectations. In *Proceedings of the* 23^{rd} *USENIX Security Symposium*, August 2014.

[80] Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger. STING: Finding Name Resolution Vulnerabilities in Programs. In *Presented as part of the 21st USENIX Security Symposium*, 2012.

[81] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[82] Lingyu Wang, Sushil Jajodia, Anoop Singhal, Pengsu Cheng, and Steven Noel. k-zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(1):30–44, 2014.

[83] Ruowen Wang, Ahmed M. Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[84] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-scale Semi-supervised Learning. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 351–366, 2015.

[85] Michelle Y. Wong and David Lie. Tackling runtime-based obfuscation in android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1247–1262, Baltimore, MD, August 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/wong.

[86] Liang Xie, Xinwen Zhang, Ashwin Chaugule, Trent Jaeger, and Sencun Zhu. Designing System-Level Defenses against Cellphone Malware. In *28th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2009.