FRED: Identifying File Re-Delegation in Android System Services

Sigmund Albert Gorski III, Seaver Thorn, William Enck
North Carolina State University
{sagorski,swthorn,whenck}@ncsu.edu

Haining Chen
Google
hainingc@google.com

Abstract

The security of the Android platform benefits greatly from a privileged middleware that provides indirect access to protected resources. This architecture is further enhanced by privilege separating functionality into many different services and carefully tuning file access control policy to mitigate the impact of software vulnerabilities. However, these services can become confused deputies if they improperly re-delegate file access to third-party applications through remote procedure call (RPC) interfaces. In this paper, we propose a static program analysis tool called FRED, which identifies a mapping between Java-based system service RPC interfaces and the file paths opened within the Java and C/C++ portions of the service. It then combines the Linux-layer file access control policy with the Android-layer permission policy to identify potential file re-delegation. We use FRED to analyze three devices running Android 10 and identify 12 confused deputies that are accessible from third-party applications. These vulnerabilities include five CVEs with moderate severity, demonstrating the utility of semi-automated approaches to discover subtle flaws in access control enforcement.

1 Introduction

Android is a dominant computing platform with more active devices than Microsoft Windows [42]. It is well known for its application-centric security model [4] where each application is granted high-level permissions based on functional needs. Prior work has deeply studied this permission model [14, 15, 48], how applications request permissions [8, 35, 41], and how users approve them [17, 18].

Less discussed are the protected resources behind those permissions. Android has a robust security architecture that has evolved significantly over the past decade [33]. A key attribute of Android's security is a middleware framework that is highly privilege-separated to mitigate the effects of software vulnerabilities. Much of Android's core functionality is broken into separate service components and system daemons. This framework provides an abstraction layer that allows third-party applications to safely and indirectly access protected resources. For example, while Android devices have a number of sockets in `/dev` for accessing GPS information, the Unix file permissions only allow the Location Manager Service to access the sockets directly. The Location Manager Service provides a remote procedure call (RPC) interface for lesser privileged applications to access location information. It is at this interface where permissions are checked.

Research has begun to analyze the correctness of permission checks within the Android framework. Stowaway [19], PScout [6], and Arcade [3] derive a mapping between APIs for third-party applications and permissions. Kratos [38], ACEDroid [2], and ACMiner [22] approximate correctness using consistency. They use static program analysis of system services to identify the different access control checks that occur for each RPC entry point and then determine if specific entry points are missing checks. ARF [23] builds on these works by considering improper re-delegation between RPC entry points. ARF found that Android's RPC entry points frequently call one another, and when the ambient authority of the execution changes (e.g., crossing between processes), confused deputy vulnerabilities can result. Our work extends these prior works by considering sensitive files that can be accessed indirectly through system services.

In this paper, we propose a tool called FRED, which performs a static program analysis of the Java-based system services in the Android framework to identify a mapping between RPC entry points (i.e., deputies) and concrete file paths that may be accessed when the RPC is invoked. Our static program analysis includes both the Java and C/C++ code for these system services. We then combine the Linux-layer Unix file permissions with the Android-layer permission check policy to identify when file access is re-delegated to third-party applications. By manually studying these file re-delegations, we are able to identify confused deputy vulnerabilities.

We applied FRED to three devices running Android 10: an AOSP Pixel 3a, a Google Pixel 3a, and a Samsung Galaxy S20. We found 12 confused deputies that allow third-party

applications to modify or read information from files or directories with a `system` UID or GID. These 12 deputies include three CVEs assigned moderate severity by Google and two CVEs assigned moderate severity by Samsung. The other seven deputies represent minor security issues.

We make the following contributions in this paper:

- *We design* FRED*, which uses static program analysis to identify security-sensitive file paths accessed by the RPC entry points of Android's Java-based system services.* FRED identified 23 RPC entry points of the total 6287 RPC entry points in AOSP 10.0.0 potentially re-relegating access to 51 files. The source code for FRED is available at https://github.com/wspr-ncsu/fred.
- *We use* FRED *to study file re-delegation vulnerabilities in the Android framework.* We identify 12 confused deputy vulnerabilities which can be accessed by third-party applications. Google and Samsung have assigned five CVEs with moderate severity based on these vulnerabilities.

We note that the vast majority of the RPC interfaces in Android's framework exist within the Java-portions of the code base. For these RPC interfaces, FRED extends its data flow and control flow analysis through the Java Native Interface (JNI) bridge to capture file operations that occur within C/C++ code. However, the Android framework also includes a collection of system services written entirely in C/C++ (e.g., the Camera Service). FANS [30] recently performed fuzz testing of these native system services, primarily looking for memory safety vulnerabilities. We leave extending FRED to entirely native system services to future work.

The remainder of this paper proceeds as follows. Section 2 provides background and a motivating example. Section 3 overviews FRED. Section 4 describes the design of FRED. Section 5 evaluates FRED by applying it to three device firmware images. Section 6 discusses limitations. Section 7 overviews related work. Section 8 concludes.

2 Background and Problem

The Android platform is built on Linux primitives, including its traditional Unix-based system calls, file system, and access control. However, unlike traditional Linux distributions, Android provides an extensive application runtime environment that strongly controls software execution. Both Android applications and most of the framework are built upon four types of components: *activities* for user interfaces, *broadcast receivers* for asynchronous communication, *content providers* for data sharing, and *services* for daemon-like servers.

The framework primarily consists of service components, which provide an abstraction layer for accessing sensitive resources including system files and device nodes. Applications make remote procedures calls (RPCs) to service entry points

```
1 boolean removeSharedAccountAsUser(Account ac, int userId) {
2   int uid = getCallingUid();
3   userId = handleIncomingUser(uid);
4   UserAccounts acs = getUserAccounts(userId);
5   boolean deleted = acs.accountsDb.deleteSharedAccount(ac);
6   if (deleted)
7     removeAccountInternal(acs, ac, uid);
8   return deleted;
9 }
```

Figure 1: The RPC entry point `removeSharedAccountAsUser` in the `AccountManagerService` allows any app to remove shared accounts the user does not manage.

using Android's custom *binder* inter-process communication (IPC) message passing system. Services then perform access control checks on RPCs using conditional statements that consider the Android permissions [15, 48] granted to the calling application (usually at install-time). This abstraction layer allows Android to significantly privilege separate functionality into different services running as separate processes and assign least-privilege access control policy to sensitive files using traditional Unix permissions, user identities (UIDs), and group identities (GIDs). Recent versions of Android also use a version of SELinux designed for Android [40]. In a small number of cases where using a framework abstraction layer is not practical, the Android runtime will add GIDs to applications based on the permissions they are granted. In such cases, the application can access associated files directly.

In this model, Android's system services act as privileged *deputies*, which are expected to perform authorization checks before accessing sensitive resources. Failing to perform proper authorization checks at RPC entry points may result in *confused deputy* vulnerabilities. ARF [23] identifies confused deputy vulnerabilities that result when an RPC entry point calls a different (privileged) RPC entry point. The authors found that Android's RPC entry points are highly interconnected and that the calling identity often changes as the execution passes between RPC entry points. However, ARF does not consider confused deputies where an RPC entry point accesses a privileged file. Since file access is determined based on the authority of the service's process and not the original calling application, confused deputy vulnerabilities may result if proper checks are not made.

Figure 1 shows an example of a file-based confused deputy vulnerability in Android discovered with FRED. The figure shows the `removeSharedAccountAsUser` RPC entry point in the `AccountManagerService`. In this context, an account is an online service such as Facebook, Google, or Dropbox. Accounts are created and managed by specific applications and typically, aside from the system, only the applications that create accounts can remove or modify them. A *shared account* is an account that has been shared across multiple users of the device. Traversing the call graph from `acs.accountsDB.deleteSharedAccount` ten methods deep reveals a call to `nativeOpen`. Propagating constant string information shows access to two files: `/data/system_ce/[userId]/accou-`

nts_ce.db and /data/system_de/[userId]/accounts_de.db (where [userId] is the RPC argument). Ultimately, removeSharedAccountAsUser deletes rows in these files to remove the shared accounts of a specified user. It is a confused deputy, because both files are only accessible by system UID processes, and it does not ensure the removal is safe (e.g., by calling isAccountManagedByCaller). We reported this vulnerability to Google and were assigned CVE-2020-0208.

FRED seeks to *semi*-automatically identify such file-based confused deputy vulnerabilities within Android's system services. Fully-automated identification of such vulnerabilities requires modeling the semantics of all safety checks that occur on the path to file access. Doing so generically is not tractable. Instead FRED seeks to reduce the search space from the over 6,000 RPC entry points (Android 10) to the smaller set of RPC entry points that access security-sensitive files, which we term *candidate RPC entry points*.

Definition 1 (Candidate RPC entry points)**.** Let E be the set of all methods handling binder interfaces registered as system services with Android's Service Manager. Let CFG_e be the inter-procedural control flow graph of $e \in E$. We note that CFG_e stops when it encounters a call to another $e' \in E$. The set of candidate RPC entry points $E_c \subseteq E$ contains all e such that CFG_e accesses (e.g., opens) a security-sensitive file.

We note that the access of a security-sensitive file may be a primary function of the RPC entry points, or it may be ancillary to performing a primary functionality (e.g., reading configuration). Differentiating primary function from ancillary function requires modeling the semantics of the RPC entry point's functionality, and heuristics to do so may miss vulnerabilities. Therefore, we do not explicitly attempt to differentiate it via program analysis. Furthermore, manual inspection is ultimately required to determine if a candidate RPC entry point contains an *improper file re-delegation*.

Definition 2 (Improper file re-delegation)**.** Let $e \in E_c$ be a candidate RPC entry point. Let f be a security-sensitive file (or directory) accessed by e. Let c be a caller of e, where c does not have privilege to access f. Then e has an improper file re-delegation if c can violate the secrecy or integrity of f.

We note that whether or not the secrecy or integrity of f is violated is contextual to the semantics of the service functionality. An RPC entry point may safely allow a caller to read or modify a part of a file related to the caller. The violation may also be subjective with respect to the purpose.

3 Overview

FRED seeks to semi-automatically identify improper file re-delegation by Android's system services by automating the discovery of candidate RPC entry points and then using manual inspection to determine if the secrecy or integrity of files is violated. Conceptually, FRED operates by (1) traversing the call graph from RPC entry points to methods that access files (e.g., java.io.FileInputStream), and then (2) performing a backwards data flow analysis from the file path arguments to determine their values from either constant strings or RPC arguments. It then compares the access control policy for those files with the access control policy of the RPC entry point. Performing this static program analysis requires overcoming the following research challenges.

- *Android system developers use a variety of file access method APIs.* The backwards data flow analysis to determine file paths starts at the code instruction that invokes a file access method. If this invocation is within a generic wrapper API, the data flow analysis will not be specific to the RPC entry point.
- *File paths are built from many parts, which are not always available to the analysis.* Paths are frequently constructed using a variety of system environment variables, as well as path and string builder APIs. When the analysis cannot identify constant values for all parts, it should be as precise as possible to match concrete file paths.
- *Java-based services call native methods through JNI.* RPC entry points registered as system services call 387 unique JNI methods in Android 10. Files paths and calls to file access methods may only exist within native code, requiring the analysis to span both the Java and C/C++ portions of services.

Figure 2 shows the overall flow of our approach consisting of the following high-level steps.

Step 1 - Identify File Access Methods: Knowledge of the specific methods used to access files provides more precise mapping of RPC entry points to file paths. For example, attempting to perform a backwards data flow analysis from a generic Java API file method may significantly overapproxiate file paths when that Java API file method is called indirectly via a generic Android API file methods. We use a combination of manual review, program analysis, and manual refinement to identify four types of file methods: libc file methods, JNI file methods, Java API file methods, and Android API file methods. While not fully automated, the manual steps are largely one-time efforts and require minimal effort to transition to a new Android version (e.g,. transitioning from Android 9 to Android 10 took less than 5 hours).

Step 2 - Identify Accessed Sensitive Files: For each RPC entry point, FRED identifies the set of security-sensitive files that it accesses. This step begins by walking the call graph from the RPC entry point to all of the file methods identified in Step 1. FRED then performs a backwards inter-procedural data flow analysis from the file methods to identify file paths that flow to it. FRED analyzes the Java and native code portions of the RPC entry point separately using Soot [29,43] and

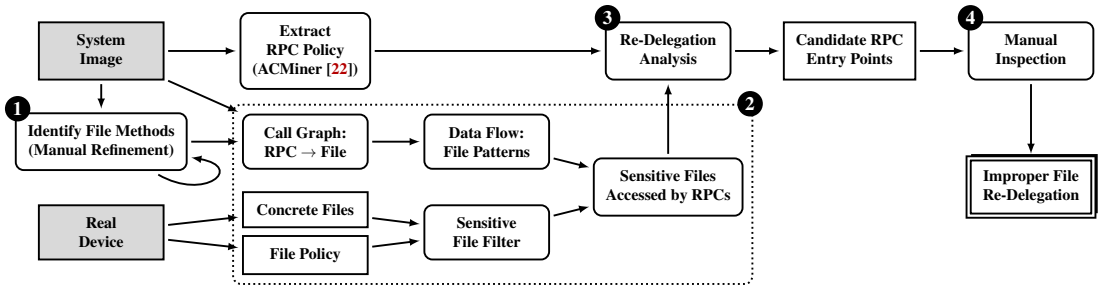


Figure 2: Overview of FRED's static program analysis process

angr [39], respectively, and then identifies which JNI methods are called by the RPC entry point. For Java code, where string construction is more intricate, FRED derives an intermediate representation that describes the string construction. This intermediate representation is then converted into regular expressions for file paths. For native code, where string construction is based on variants of `strcpy`, FRED simply derives the full file paths. Finally, these regular expressions and file paths are matched to a list of security-sensitive files from a real device. For the purposes of this paper, we consider a file to be security sensitive if the UID or GID is `system` or the GID corresponds to a GID mapped to an Android permission.

Note that our approach primarily identifies when files are opened. It does not attempt to differentiate read and write access. First, read and write operations are typically methods invoked on objects returned from the open call. Tracking these objects can be imprecise and cause FRED to miss file accesses. Second, the read and write methods of file access wrappers are unknown and require additional manual effort to identify. Given that we found the number of candidate RPC entry points for file open was manageable (see Section 5), and manual inspection is needed anyway (Step 4), there was no need to build this additional analysis.

Step 3 - Re-Delegation Analysis: Candidate RPC entry points provide indirect file access in ways that may or may not be safe. First, FRED determines if and how each RPC entry point can be called by a third-party application using information and techniques developed for ACMiner [22] and ARF [23]. ACMiner uses Soot to statically identify inconsistent access control checks between similar RPC entry points, and ARF builds on ACMiner to identify re-delegation between RPC entry points. FRED uses ACMiner's mapping of RPC entry points to permissions as input. It also adopts several of ARF's heuristics to determine if the RPC entry point can only be accessed by system services (e.g., UID checks as the first conditional). Finally, for each RPC entry point accessible to third-party applications, FRED combines (a) permission requirements with (b) the Unix permissions for the security sensitive files accessed by the RPC entry point (Step 2). It produces the set of candidate RPC entry points.

Step 4 - Manual Inspection: Since it is not practical to capture all of the ways in which an RPC entry point may allow safe access to sensitive files, FRED relies on manual inspection to identify improper file-redelegation. Since the total number of candidate RPC entry points is relatively small for Android 10, an expert can review how each RPC entry point uses sensitive files. We also identified several heuristics to further reduce this list, incorporating them into Step 3. Finally, the manual inspection should also consult SELinux policy to determine if (a) the service can access the given file and (b) a third-party application can invoke the service. In our study, these additional SELinux checks were simple to confirm manually and no discovered vulnerabilities were limited by SELinux policy. Future versions of FRED could incorporate SELinux policy checks into the pipeline.

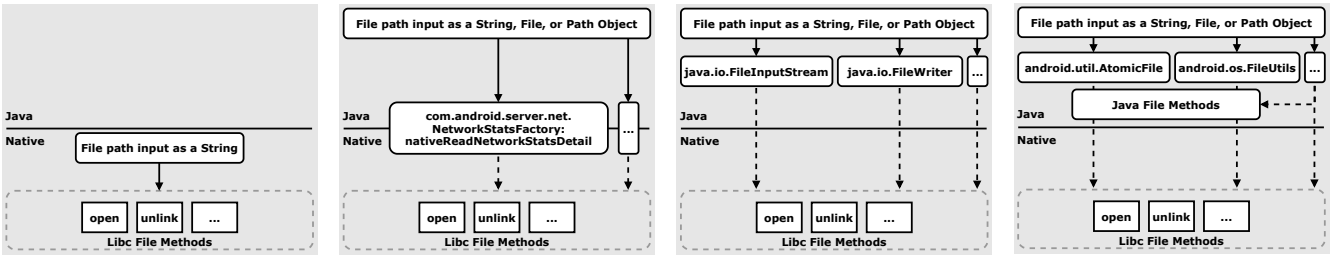
Alternate Approaches: While FRED uses static program analysis to help discover improper file re-delegation, alternative approaches exist for identifying the regular expressions of file paths accessed by RPC entry points. Dynamic analysis provides evidence that files are accessed during execution; however, code coverage and test case generation are frequently limiting factors for dynamic approaches. Recently, Centaur [32] proposed phased concrete-to-symbolic execution (PC2SE) to avoid state space explosion during the initialization phase of Android's system services. This optimized symbolic execution is a promising direction for future work; however, its dependence on concrete execution within an emulator limits its application to non-AOSP firmware images. We apply FRED to non-AOSP firmware in Section 5.

4 Design

This section details the design of FRED following the first three steps described in Section 3.

4.1 Identifying File Methods

FRED seeks to identify the highest abstraction level possible for file methods in order to best match the potential file paths to the RPC entry points. Figure 3 shows four data flow paths of interest. First, Figure 3a depicts when the file path originates in native code and is either directly or indirectly passed



(a) JNI methods using file paths defined in native code

(b) JNI methods using file paths defined in Java code

(c) Java API File Methods

(d) Android API File Methods

Figure 3: Four data flow paths in which file paths arrive as the arguments to libc file methods.

to a libc file methods. Next, Figure 3b depicts when the file path originates in Java code and is passed to a JNI method that is not a generic file method. Finally, Figures 3c and 3d depict when the file paths are passed to the Java or Android API file methods designed as generic file methods.

4.1.1 Libc File Methods

Standard libc file methods are well-known and defined in a handful of header files, including `stdio.h`, `fcntl.h`, `unistd.h`, `sys/stat.h`, and `stdlib.h`. Libc file methods can be distinguished from other methods by their arguments. Specifically, a libc file method has either a string file path or a file descriptor as an argument. We manually explored the approximately 400 methods in these header files and identify 70 libc file methods (e.g., `fopen`, `open`, `rename`, and `unlink`).

Android also uses Fortify [9] to ensure developers properly use standard libc methods. Fortify wraps the standard libc methods at compile time with methods that perform safety checks at both runtime and compile time. Since FRED analyzes compiled native binaries (Section 4.3), these Fortified libc methods must also be included in our list of libc file methods. Our manual investigation identified 6 additional libc file methods for Android 10.0.0.

4.1.2 JNI File Methods

To find the JNI file methods, FRED first uses Soot [29, 43] to extract a list of all JNI methods. Each method is then processed to add back in argument variable names by extracting them from the Java source code using features available in the standard `javac` compiler [34]. Since many JNI file methods have names similar to the libc file methods they invoke, we use keywords from the file methods determined in Section 4.1.1 (e.g., `open`, `read`, `write`, `link`, `unlink`, and `remove`) along with their synonyms to identify potential JNI file methods. We then identified the JNI file methods by further filtering the resulting methods using their arguments, looking for methods with argument names or types such as `file`, `path`, `fd`, and `fileDescriptor`. If the Java source code for the operating system is not available, such as in OEM builds of Android, we decompile the Java code of the OEM build and manually inspect the call sites of the JNI methods to determine the arguments purpose from context in the code.

4.1.3 API File Methods

As shown in Figure 3, two classes of API file methods exist: 1) standard Java API methods, and 2) Android API methods that make use of the standard Java API methods while providing additional functionality. FRED identifies these API file methods using the JNI file methods identified in Section 4.1.2. Intuitively, FRED begins with the set of all API methods in Android. It then traverses the call graph from each API method to find a path to a JNI file method. However, given how heavily interconnected Android's API methods are, relying only on the occurrence of JNI file methods in the call graph to identify API file methods results in many false positives. Therefore, we use an iterative process with manual refinement to improve precision.

Extracting API Methods: FRED extracts the complete set of the Android API methods available to both third-party applications and system developers from the framework Jar file from an AOSP build. Java language abstractions (e.g., abstract classes and interfaces) complicate the identification of callable methods. For example, `java.nio.file.FileSystem` is an abstract class and only defines private methods; however, class implementations are accessible through static methods in the class `java.nio.file.FileSystems`. To consider such overriding methods, FRED uses Soot [29, 43] to load and extract the classes and methods within the Jar file. It then uses Soot's class hierarchy analysis (CHA) [12] of all the classes and methods, including additional methods that override methods from the current set of API methods.

Identifying API File Methods: FRED performs a reaching analysis on the call graph of each API with calls to file methods as sinks. Soot's CHA over-approximates the runtime call graph, introducing edges between methods that do not exist at runtime. As such, FRED's reaching analysis over-approximates the actual API file methods.

To reduce over-approximation, FRED uses the following

semi-automated process. (1) FRED performs a reaching analysis on the API methods, reporting an API as a possible file method if at least one sink is encountered during the reaching analysis. (2) The source code of the reported APIs is examined to determine if they are actually file methods. (3) Based on this inspection, API methods are placed in either an exclude list (if not a file method) or an include list (if a file method). (4) The process is repeated from step (1), ignoring the methods in both the exclude and include lists and repeating until all reported API methods have been examined. We note that iteration is required due to the heavily interconnected nature of the API methods.

This semi-automated process is completed twice: first for Java API file methods and then for Android API file methods. The identification Java API file methods only include those API in `java.*` classes. The identification of Android API file methods additionally uses Java API file methods as sinks.

4.2 Extracting File Paths Used in Java Code

For each RPC entry point, FRED identifies all calls to file methods. It then performs a backwards inter-procedural data flow analysis to identify the file paths passed to those file methods. We now describe FRED's data flow analysis in Java code. Section 4.3 discusses C/C++ code called through JNI.

Java file paths are represented by the three key classes: `java.lang.String`, `java.io.File`, or `java.nio.file.Path`, where `File` and `Path` are wrappers for a `String` representation of the file path. These wrappers use a combination of class constructors and methods to specify the `String` value. Therefore, extracting the file paths used in the file methods can be reduced to reconstructing strings. To extract all the possible file paths accessed for a given RPC entry point, FRED performs a backwards inter-procedural data flow analysis from the variable containing the representation of the file path opened by a file method to the RPC entry point.

Since complete file paths cannot always be determined via static analysis, FRED creates regular expressions to match the concrete security sensitive file paths identified in Section 4.4. However, before generating these regular expressions, FRED captures these partial file paths using an intermediate expression generated during the data flow analysis.

4.2.1 Intermediate Expressions

FRED uses intermediate expressions to represent both file paths and metadata (e.g., source method and statement) for each expression part. The metadata aids post-processing, making it possible to detect situations not handled by the data flow analysis, and to simplify the expressions when transforming them to regular expressions.

FRED's intermediate expressions are trees that join string segments using boolean decisions. Each node in an intermediate expression tree is either a *Leaf* or a *Branch*. As shown in Table 1, each *Leaf* node is sub-divided into *Constant*, *Any*, *Unknown*, and *PlaceHolder*. A *Constant* node identifies literal constants hard-coded into the Java source code. An *Any* node represents a value that could not be determined from the source code. Examples of *Any* nodes include values representing a `UserId`, package name, time stamp, or array value. In Section 4.2.3, these values resolve to regular expressions of either `.*` or `\d+`. In contrast, an *Unknown* node indicates the data flow analysis encountered a situation that it could not handle. Finally, a *PlaceHolder* node is used for values that have not yet been computed by the data flow analysis. The occurrence of *Unknown* and *PlaceHolder* nodes in the final intermediate expression output indicates that the data flow analysis needs modification to handle a special case.

Table 1: Building Blocks for Intermediate Expressions

Node Name	Type	Description
Constant	Leaf	A Java primitive or string constant value.
Any	Leaf	Represents a value that could not be determined.
Unknown	Leaf	Represents a unexpected outcome in the analysis.
PlaceHolder	Leaf	A placeholder for a value that is being computed.
Append	Branch	Concatenates the values of its children.
Or	Branch	Represents the possibility that any one of its children is a valid value.
Loop	Branch	Indicates the existence of a loop.
Parent	Branch	The value is the parent path of its child's value.
Name	Branch	The value is the file name of its child's value.
EnvVar	Branch	Represents a value retrieved from the environment. Its child represents the key used to get the value.
SysVar	Branch	Represents a value retrieved from the system properties. Its child is the key used to retrieve the value.

Branch nodes are primarily either an *Append* or an *Or* node. An *Append* node represents a boolean `AND` operation, where child nodes are concatenated in the order listed. The *Append* node is specifically designed to handle the concatenation of strings and other values that commonly occur when a file path is constructed in Java source code. In contrast, an *Or* node represent a boolean `OR` operation and captures when the data flow analysis encounters a variable with multiple possible values. There are also several other subtypes of *Branch* nodes. Section 4.2.2 describes them in more detail.

4.2.2 Data Flow Analysis

FRED performs a backwards inter-procedural data flow analysis to determine possible values for the file paths passed as arguments to file methods invoked by Java-based RPC entry points. The sink for this backwards data flow analysis is the file path argument. For many file methods, the sink is a string (i.e., `java.lang.String`). When a file methods has multiple string arguments, it is difficult to know which argument is the file path. Fortunately, the regular expressions produced for non-file paths will not match any concrete files from the file system. Therefore FRED conservatively determines values for all string arguments. For file methods that are passed non-string file paths (e.g., `java.io.File` and `java.nio.file.Path`), FRED first performs use-def analysis from the argument to identify the location of the constructor,

which is passed a string. For this discussion, this constructor can be viewed as the sink for the data flow analysis.

At a high level, FRED determines the possible values of file paths by annotating instructions in the Inter-procedural Control Flow Graph (ICFG) of the Android framework with intermediate expressions. This process begins by annotating the data flow sink (i.e., file path argument) with a *PlaceHolder* node. FRED then performs a inter-procedural use-def analysis to traverse the ICFG backwards, using CHA [12] where necessary. If the definition of the argument is a constant value, the *PlaceHolder* node is replaced with a *Constant* node that includes the value. However, there are various scenarios when the definition is not a constant. In these cases, FRED annotates the definition with a new *PlaceHolder* node and recursively[1] attempts to determine the value for this new *PlaceHolder* node. Once the value for a *PlaceHolder* node is determined, the recursive call returns the resulting intermediate expression to the earlier invocation, populating its *PlaceHolder* node, and possibly combining multiple intermediate expressions with *Append* or *Or* branch nodes. We note that while FRED identifies the values for each sink sequentially, it retains the annotations on the ICFG to avoid resolving them multiple times.

As this recursive algorithm proceeds, there are frequently multiple *PlaceHolder* nodes annotated on the ICFG at any point in time. FRED leverages this fact to handle loops, ensuring that the value for a *PlaceHolder* node is only ever computed once. Specifically, FRED constructs a graph of *PlaceHolder* nodes where an edge between two *PlaceHolder* nodes occurs when one references the other in its computing expression. FRED then uses Johnson's algorithm [28] for detecting elementary circuits to locate simple cycles in the graph. When this occurs, the *PlaceHolder* node referencing the head of the simple cycle is replaced with a *Loop* node. Empirically, we found that *Loop* nodes are little more than a source of noise in our analysis. Therefore, FRED currently removes *Loop* nodes when transforming intermediate expressions into regular expressions (Section 4.2.3).

There are various scenarios when the use-def analysis for a *PlaceHolder* node does not find a constant at the definition. FRED handles a variety of special cases including string builders, path builders, directory listings, and parent path and file name access methods. FRED also handles environment variables and system properties by identifying the string-based keys used to look up these values. Finally, FRED handles specific fields in Android framework classes by performing a def-use analysis to determine all possible assignment sites for the framework. This information is later used during the backwards data flow analysis to combine possible values. Appendix A details each of these cases.

[1] While we describe the algorithm as recursive to simplify discussion, it is in fact tail-recursive and our implementation is iterative.

4.2.3 Regular Expression Transformation

After FRED identifies intermediate expressions for the arguments to file methods, it transforms them into regular expressions. This transformation is performed as follows.

Step 1 (Remove Loops): FRED removes all *Loop* nodes from intermediate expressions and any empty *Branch* nodes that result. We empirically found that the loops captured by our data flow analysis did not actually influence the construction of the file paths within the code. As such, *Loop* nodes can be safely removed from intermediate expressions.

Step 2 (Resolve System and Environmental Variables): To resolve system properties, FRED uses the various `.prop` files of the Android system (e.g., `/default.prop` and `/system/build.prop`) to lookup the value based on the key determined during data flow analysis. To resolve environmental variables, FRED uses `adb shell echo ${VARIABLE}` to get their values.

Step 3 (Resolve UserId Variables): Android supports multiple physical users. It is common for file paths to include the UserId of the current physical user to separate user specific files. To avoid unnecessary *Any* nodes, we replace UserId with the string "0", which is the UserID for the primary user (and the only user on our test devices).

Step 4 (Resolve `TVInputManagerService` Regex): The `TVInputManagerService` contains two entry points that open file paths in the `/dev` directory based on a regular expression. For these entry points, FRED extracts the regular expression used to match files and replaces the *Any* node with the extracted expressions.

Step 5 (Convert to DNF): FRED transforms the `AND` and `OR` logic of *Append* and *Or* nodes into disjunctive normal form (DNF), maintaining the order of the append operations throughout the transformation. This process flattens the tree and simplifies file path dependent transformations (e.g., determining the parent or file name of a existing file path).

Step 6 (Resolve Parent and File Names): With the intermediate expression in DNF and the remaining constants resolved, FRED can evaluate the *Parent* and *Name* branch nodes. At this point, *Parent* and *Name* nodes typically have a single *Append* child that concatenates parts of a path. For this *Append* node, FRED locates the last occurrence of the '/' path separator. *Parent* nodes are replaced with a new *Append* node containing all the children that occur before the last '/' from the previous *Append* node. Any text after the '/' within the node that contains the last occurrence of '/' is removed. *Name* nodes are replaced with a new *Append* node containing all the children that occur after the last '/' from the previous *Append* node. Any text before the '/' within the node that contains the last occurrence of '/' is removed. When the child of a *Parent* or *Name* node is an *Any* node, the *Parent* or *Name* node is simply replaced with the *Any* node.

Step 7 (Combine and Normalize): After the previous step, the intermediate expression only contains *Constant*, *Any*, *Ap-*

pend, and *Or* nodes. All adjacent *Constant* nodes under an *Append* node are concatenated into a single *Constant* node. Similarly, all adjacent *Any* nodes under an *Append* node are replaced with a single *Any* node. The single *Any* node is only set to `\d+` if all of the *Any* nodes under the *Append* are `\d+`. Otherwise, a `.*` *Any* node is used. Finally, duplicate and trailing '`/`' characters are removed from *Constant* nodes.

Step 8 (Duplicates Removal): The DNF form of the intermediate expression may cause a *Or* node to have multiple children that resolve to the same regular expression value. FRED removes these duplicates.

Step 9 (Regular Expression Creation): In the final step, FRED creates the regular expression. *Constant* nodes are string values. *Any* nodes are either `.*` or `\d+`, which was determined during data flow analysis. The children of *Append* nodes are concatened. Finally, the children of *Or* nodes are combined by inserting '`|`' between each node.

4.3 Extracting File Paths Used in Native Code

Android's Java-based RPC entry points often use the Java Native Interface to invoke code written in C/C++. Figure 3 shows two scenarios that are not covered by the Java analysis described in Section 4.2. Both scenarios occur when a JNI method that is not a generic file method calls a libc file method. The first scenario defines the file path string within C/C++ code (Figure 3a), whereas the second scenario passes the file path string from Java to C/C++ code (Figure 3b). FRED extracts these file paths from binary `.so` files using angr [39].

Identifying JNI Methods in Native Code: While the `native` keyword identifies JNI methods in Java code, the corresponding C/C++ function is not clearly annotated in either the source or binary code. The C/C++ name also does not always match the Java name (e.g., method overloading). For each `.so` file, we use angr to traverse the CFG from all exposed library functions to identify calls to `jniRegisterNativeMethods`. We then use angr's symbolic execution engine to identify the array of `JNINativeMethod` structures passed as a parameter. This structure provides a mapping from the Java method signature to a function pointer in the `.so` file. We traverse the array to identify the function address of the C/C++ handler for each JNI method. In a small number of cases, we found function addresses to be `NULL` and used function name matching as a fallback. As the function name resolution was verified to be correct for all such `NULL` address cases, we leave additional address resolution techniques to future work.

Identifying File Paths: FRED first traverses the ICFG from the JNI method handler to identify any calls to libc file methods (Section 4.1.1). When a file method is found, FRED uses angr's Reaching Definition Analysis (RDA) to determine the possible values for the relevant file path arguments. To reduce the search space, FRED only includes nodes in the variable's backward slices as potential sources. For each variable definition source, FRED determines if the value originates within the C/C++ code or if it is passed from Java. If it originates in C/C++ code, FRED extracts the constant string. Note that angr correctly propagates strings through standard libc functions (e.g., `strcpy`) and therefore automatically handles string construction. We did not encounter the more complicated string construction methods found in the Java code.

Finally, FRED outputs a JSON file mapping each Java JNI method name to the file paths used in file methods, including when the path originates in Java. FRED then uses Soot's call graph from RPC entry points to JNI calls to supplement the file paths identified in Section 4.2.

4.4 Security-Sensitive File Paths

FRED's re-delegation analysis (Section 4.5) matches the file paths from Sections 4.2 and 4.3 with concrete file paths on a device. Using concrete file paths both reduces noise and helps determine if a re-delegation is possible on a given device. We further consider only security-sensitive concrete file paths to limit manual inspection to areas of potential vulnerabilities.

FRED uses a real device to extract the concrete file paths and file access control policy. Specifically, FRED executes `adb shell ls -laRZ` as root on a rooted device to extract this information. FRED then processes the file system information to resolve symbolic links, taking care to avoid circular paths. In doing so, FRED ignores symbolic links in paths starting with `/sys/.*/subsystem` and `/proc/.*/fd` as these all point back to the root of the file system. We considered statically extracting files from the firmware image using BigMAC [26]; however, it has very limited ability to identify files in `/data`, which contains many security-sensitive files that match file paths determined by FRED.

Finally, FRED classifies each concrete file and directory as security-sensitive based on the file owner and group. Specifically, we observe that most system services run as `system` and Android has multiple files in `/system/etc/permissions` that define GIDs that are automatically given to applications granted specific Android permissions. Therefore, a path is marked security sensitive if it has a owner or group of `system`, or the group corresponds to an Android permission GID.

4.5 Re-Delegation Detection

FRED's re-delegation analysis identifies a set of candidate RPC entry points (Definition 1) for manual inspection. FRED focuses on exploitation by third-party applications as these represent the most significant risk. Malicious applications and services included in OEM builds are outside the scope of this paper. While it is possible for exploited system applications to exploit confused deputies, the interactions between system applications and system services have been considered by prior work [23, 24, 46].

Re-Delegation of System Files: For each RPC entry point that accesses a `system` UID or GID file or directory, FRED checks if the RPC entry point's authorization checks contain at least one system-level permission (i.e., those permissions without a protection level of *normal*, *dangerous*, *instant*, *runtime*, or *pre23*). If so, FRED excludes the RPC entry point from the candidate set. RPC entry points can also be restricted based on specific PIDs, UIDs, and GIDs. Similar to ARF [23], FRED checks if the first conditional statement is a check for a special UID, PID, or GID, and if the RPC entry point includes an authorization check that restricts it to special callers (e.g., calling `getActiveAdminWithPolicyForUidLocked()` in the `DevicePolicyManagerService`). If so, FRED excludes the RPC entry point. The remaining RPC entry points that access `system` UID or GID files and directories are candidates.

Re-Delegation of Other Files: FRED also considers files and directories with a GID that maps to an Android permission. An RPC entry point is a candidate for manual inspection if the permissions mapped to a file's GID is not a subset of the permissions checked by the RPC entry point. Similar to `system` files, FRED excludes RPC entry points that are not accessible to third-party applications.

Reducing Manual Inspection: While performing manual inspection of the AOSP Android 10.0.0, we developed two methods to systematically reduce the number of candidate RPC entry points that require manual inspection.

System-Specified Values - We discovered a number of unresolved *Any* nodes that are the result of values that are both specific to the caller and cannot be influenced by the caller (e.g., application name, data path, and code path). As such, a file path constructed of such components is unique to the caller, implying the caller is intended to have access to these resources. Since these `Any` nodes caused the regular expressions to match the majority of file paths within the file system, they were excluded before matching security-sensitive files.

Safe File Method Callers - We also found that file methods accessing security-sensitive files were often called from the same method. Reviewing these methods, we found that many of them could only be called in safe ways. In total, we found 41 unique callers of file methods where the file being accessed could not be influenced or have its data retrieved by any entry point. We exclude the regular expressions steaming from these 41 sinks from the output of FRED.

5 Evaluation

This section demonstrates FRED's utility by applying it to three Android firmware images and investigates potential vulnerabilities. Recall that FRED automates the discovery of candidate RPC entry points (Definition 1) and manual investigation is required to determine if a given candidate RPC entry point has improper file re-delegation (Definition 2).

5.1 Experimental Setup

Our current implementation of FRED was designed for AOSP version 10.0.0_r1 (i.e., API 29) built for a Pixel 3a device. We previously ran FRED on AOSP version 9.0.0_r11 for a Pixel 3 device, identifying the same vulnerabilities and comparable findings as those highlighted this section. We also ran FRED on two additional Android 10 firmware images. Our study included the following three devices.

- **AOSP Pixel 3a** running a user debug build of Android 10 r1. This device represents the primary target used when developing FRED.
- **Google Pixel 3a** running Android 10 build QQ3A.200805.001. This device represents a target close to our original target, demonstrating FRED's ability to run on firmware images without source code. No changes were required to run FRED on this target.
- **Samsung S20** running Android 10 build QP1A.190711.020. This device has significant differences from our original target. Running FRED on this target required modifications to ACMiner's list of methods to not analyze to address call graph imprecision that prevented analysis from completing on our computing resources. We also added 88 file methods identified by re-running FRED's semi-automated API file method analysis.

Our analysis was run on a Dell R611 server with an Intel Xeon E5-2620 V3 (2.40 GHz) processor and 128 GB RAM running VMware ESXi. A single VM running Ubuntu 18.04.3 and OpenJDK 1.8.0_171 was given full host resources.

ACMiner took approximately 1 hour and 45 minutes to extract authorization checks of RPC entry points for the AOSP Pixel 3a. FRED took approximately 10 minutes to construct a complete list of the security sensitive file paths, 1 hour and 16 minutes to extract the intermediate expressions from Java code, 15 minutes to extract file paths from native code, and 12 minutes to convert the intermediate expressions to regular expressions and identify candidate RPC entry points. For the Google Pixel 3a, all stages of the analysis had a runtime that was virtually the same as that of the AOSP Pixel 3a. However, as a result of the increased code base and the significant increase in the number of RPC entry points, the runtime of FRED on the Samsung S20 doubled to approximately 4 hours when extracting authorization checks and 2 hours and 45 minutes when extracting the intermediate expressions from Java code. All other stages had virtually the same runtime.

5.2 FRED Characterization

We initially analyzed an AOSP Pixel 3 running Android 9.0.0 (the current version when we began our work). We then migrated to different builds of Android 10. The following discussion focuses on the Android 10 builds. However, we

Table 2: Characterization of Program Analysis

	AOSP Pixel 3 (9.0.0)	AOSP Pixel 3a (10.0.0)	Google Pixel 3a (10.0.0)	Samsung S20 (10.0.0)
# API Methods	65,508	73,073	73,461	83,346
# JNI Methods	5,176	5,416	5,419	7,023
# JNI File Methods	368	360	360	360
# Java API File Methods	907	888	888	888
# Android API File Methods	962	1,068	1,087	1,155
# Total File Methods	2,237	2,316	2,335	2,403
# Total RPC Entry Points	5,337	6,287	6,304	12,169
# RPCs with File Methods	1,966	2,927	2,985	4,163
# File Methods in RPCs	661	602	717	766
# JNI Methods in RPCs	365	387	390	860
# Sec. Sensitive File Paths	139,463	157,074	56,434	56,559

include general statics for Android 9.0.0 in Table 2 to provide context as we discuss the minimal manual effort required to migrate FRED between different versions of Android.

5.2.1 File Methods in Practice

JNI File Methods: The procedure in Section 4.1.2 took 24 hours to identify the JNI file methods from a total of 5,176 JNI methods for AOSP 9.0.0. The process only took an additional hour for the 240 JNI methods added to AOSP 10.0.0. As Google 10.0.0 only added 3 JNI methods over AOSP 10.0.0, it only took a few minutes. However, the added 1,607 JNI methods of the Samsung S20 device took an additional 3 hours to evaluate. As shown in Table 2, we identified 368 JNI file methods for AOSP 9.0.0 which was reduced to 360 JNI file methods for AOSP 10.0.0 because of changes in the code of the Java API. We did not find any additional JNI file methods for either the Google or Samsung devices.

API File Methods: While the semi-automated procedure for identifying API file methods (Section 4.1.3) may appear manually intensive, it is largely a one-time cost with minimal additional manual effort needed to transition to a new version or build. For AOSP 9.0.0, the process took approximately 96 hours to identify the API file methods from a total of 65,508 API methods. However, the process only took an additional 5 hours to consider the 7,565 API methods added in AOSP 10.0.0. As there was a minimal addition of 388 API methods for Google 10.0.0 compared to AOSP 10.0.0, the process only took an additional 30 minutes and no adjustments were made manually. However, the additional 10,273 API Methods for the Samsung S20 device took an additional 7 hours.

Table 2 shows that for AOSP 9.0.0, out of the possible 65,508 API methods, FRED identified 2,237 file methods consisting of 368 JNI file methods, 907 Java API file methods, and 962 Android API file methods. For AOSP 10.0.0, out of the possible 73,073 API methods, number of file methods increased to 2,316 despite a decrease in the number JNI file methods (360) and Java API file methods (888). This decrease in JNI and Java API file methods was a result of modifications made to the Java API in Android 10. Across different builds of the same version, we observed no changes in the JNI and Java API file methods. All changes were in the Android API for the Google 10.0.0 and Samsung devices which increased the total number of file methods to 2,335 and 2,403 respectively.

RPC Entry Points Accessing Files: Using these lists of file methods, FRED identified 1,966 RPC entry points containing calls to 661 unique file methods for AOSP 9.0.0. For AOSP 10.0.0, the number of RPC entry points increased to 2,927 while the number of unique file methods decreased to 602. As shown in Table 2, the Google 10.0.0 and Samsung devices both saw an increase in the RPC entry points and the unique file methods called by them over those in AOSP 10.0.0.

5.2.2 Characterizing Security Sensitive Files Paths

As discussed in Section 4.4, a file path is considered security sensitive if it has a UID or GID of `system`, or a GID corresponding to an Android permission. Table 2 illustrates the number of security sensitive files identified for each device. Both AOSP 9.0.0 and AOSP 10.0.0 devices contain over 130,000 security sensitive file paths, while both the Google and Samsung devices have around 56,000 security sensitive file paths. The difference results from the AOSP images being built with the user debug flag (providing root access). The user debug flag also provides a more permissive SELinux policy for `adb`, allowing it to view file paths that are normally restricted by production builds.

5.2.3 Reducing Candidate RPC Entry Points

For AOSP 10.0.0, the file path extraction procedure (Section 4.2) produced 7,331 unique intermediate expressions spanning 2,927 unique RPC entry points and reducing to 462 unique regular expressions. Table 3 shows there was only a minor change in these numbers for the Google device; however, the Samsung device saw a significant increase in all three. This difference is likely attributed to there being almost double the number of RPC entry points for the Samsung device compared to AOSP 10.0.0 (Table 2).

Table 3 also shows the reduction after using the regular expressions to match security-sensitive files and applying the refinements to reduce manual inspection. AOSP 10.0.0 contains 327 unique intermediate expressions spanning 179 unique RPC entry points and reducing to 63 unique regular expressions that match 738 unique security sensitive file paths. Table 4 characterizes the number of intermediate expressions removed by each reduction method. As shown, the most significant reduction in intermediate expressions is a result of our list of 41 file method callers predetermined to be safe. Table 5 shows the number of intermediate expressions that when transformed into regular expressions did not match any security sensitive file paths.

Finally, Table 3 shows the remaining RPC entry points requiring manual inspection after the re-delegation logic is

Table 3: FRED Reducing Candidate RPC Entry Points

	AOSP Pixel 3a	Google Pixel 3a	Samsung S20
All RPC Entry Points With Intermediate Expressions			
# Intermediate Exprs.	7,331	7,281	8,150
# Regexes	462	463	769
# RPC Entry Points	2,927	2,985	4,163
Regex Matches of Security Sensitive Files After Exclusions			
# Intermediate Exprs.	327	590	455
# Regexes	63	88	115
# RPC Entry Points	179	291	380
# Files	738	1,152	1,403
Candidate RPC Entry Points Requiring Manual Inspection			
# Intermediate Exprs.	23	42	113
# Regexes	7	10	36
# RPC Entry Points	23	31	60
# Files	51	44	143

Table 4: Intermediate Expressions Excluded from Analysis

	AOSP Pixel 3a	Google Pixel 3a	Samsung S20
Composed of System Specified Values of the RPC Caller			
# Intermediate Exprs.	228	227	600
# Regexes	16	16	31
# RPC Entry Points	605	609	804
According to a List of Pre-Determined Safe File Method Callers			
# Intermediate Exprs.	5,208	5,173	5,540
# Regexes	136	136	113
# RPC Entry Points	914	920	1,329

applied. Note that while we discuss the results in terms of the number of unique RPC entry points, expressions, and files, a candidate RPC entry point may have multiple files that require separate manual inspections. For AOSP 10.0.0, FRED identified 23 candidate RPC entry points potentially re-delegating access to 51 files. Of these 23 deputies, 21 resulted from Java file accesses and 2 resulted from C/C++ file accesses. Each of the 2 candidate RPC entry points accessed a single file in C/C++ code that was not in the list of files being accessed by the other 21 candidate RPC entry points. Similar results were observed for the Google device, including the same candidate RPC and files from the C/C++ code analysis.

A manual review of the 23 candidate RPC entry points for AOSP 10.0.0 identified 10 deputies that both (a) are accessible by third-party applications and (b) improperly re-delegate access to one or more security-sensitive the files. The remaining 13 deputies safely use the files, either using them in a way that is not accessible to a caller or in a way that the caller could only retrieve or modify a subset of the data in the file by design. The same vulnerable deputies were found in both the Google and Samsung devices.

For the Samsung device, FRED identified significantly more candidate RPC entry points (60) and files (143) compared to both the AOSP 10.0.0 and Google devices. This likely results from the Samsung device containing more RPC entry points accessing files. The Samsung device included the same candidate RPC entry points and files from the C/C++ code analysis as the AOSP 10.0.0 and Google devices. Included in these 60 candidate RPC entry points were 2 that re-delegate access to the same additional file from C/C++ code.

Table 5: Regex Does Not Match Any Security Sensitive Files

	AOSP Pixel 3a	Google Pixel 3a	Samsung S20
# Intermediate Exprs.	1,618	1,291	1,555
# Regexes	279	255	534
# RPC Entry Points	973	908	1,650

5.2.4 Impact of SEAndroid

Our re-delegation analysis did not account for the impact of SEAndroid policy, which may restrict (a) the RPC entry point from accessing a file even if allowed by the Unix permissions and (b) a third-party application from accessing the service.

RPC entry points accessing files: The RPC entry points considered by our analysis are registered with Android's Service Manager. We verified that most of the corresponding services execute with the `system_server` SEAndroid label by checking (1) whether a service gets started in system server's source code, (2) whether the service is running as a thread of system server in `ps -AZ`, or (3) whether SEAndroid allows system server to add the service. One exception is entry points under `com.android.internal.telephony` which runs in the `com.android.phone` process with `radio` SEAndroid label.

For each concrete security-sensitive file, we used `SETools` [1] to determine if the binary policy bundled with the firmware allows the SEAndroid context (e.g., `system_server`) of the entry point's corresponding service to access the SEAndroid context for that file in any way. Empirically, we found that SEAndroid does not affect our analysis with the AOSP or Google Pixel 3a firmware. However, for the Samsung device, the SEAndroid policy removes 3 RPC entry point to file mappings consisting of 2 RPC entry points and 3 files.

Apps accessing RPC entry points: We confirmed that the SELinux policy does not restrict third-party applications from calling the 10 vulnerable AOSP deputies discussed in Section 5.3. SELinux policy defines `call`, `transfer`, and `find` permissions for Binder service operations. The `call` and `transfer` permissions are generally granted to apps so that they can make binder calls into the system server and service manager. The `find` permission is commonly used to allow finding services via the service manager (though Android primarily gates access to services at the RPC entry point level via Android permissions, not at the service level). The AOSP policy allows the `untrusted_app_all` domain to `find` any Binder service with the `app_api_service` SELinux attribute. We confirmed that all four vulnerable AOSP system services in Table 6 have this attribute. For the additional `BlockchainTZService` service in the Samsung device, we did not observe the `app_api_service` attribute. However, due to the more complex nature of the Samsung SELinux policy,

we notified Samsung of the potential vulnerabilities anyway. Interestingly, Samsung informed us that they fixed the two vulnerable deputies adding SELinux policy to prevent third-party applications from accessing `BlockchainTZService`.

5.3 Vulnerability Study

Table 6 shows the vulnerabilities discovered following our manual inspection methodology described in Appendix B. While some deputies may re-delegate access to multiple files, we only report one vulnerability per deputy. We group the re-delegation vulnerabilities into 3 categories: (1) ability to manipulate data in the file(s), (2) exposure of data from or about the file(s), and (3) denial of service. While 10 of the 12 vulnerabilities were originally discovered in AOSP 10.0.0, they were confirmed to also exist in the Google and Samsung devices. The remaining two vulnerabilities are specific to the Samsung device. Finally, in all cases, the file path being improperly accessed was statically defined in the code and not controllable by an attacker.

Responsible Disclosure: We verified all of the potential re-delegations in AOSP through manual inspection of source code. All identified vulnerabilities identified in AOSP have been submitted to Google via Android Security Rewards Program. Of those submitted, 3 received a "moderate" severity rating and have been assigned the common vulnerability and exposures (CVE) identifiers CVE-2020-0208, CVE-2020-0209, and CVE-2020-0210. All 3 CVE vulnerabilities were fixed in the June 2020 security update for Android 10. The two vulnerabilities identified in the Samsung device involved code paths in native code for which we could not inspect. We provided the names of the deputies and corresponding JNI calls to Samsung, who confirmed the vulnerabilities and assigned CVE identifiers CVE-2021-25460 and CVE-2021-25459. Fixes for the Samsung vulnerabilities were released to devices in September 2021.

5.3.1 VC1: Data Manipulation

FRED identified 5 vulnerabilities (1→5 in Table 6) that enable a third-party application to access or manipulate data in security sensitive files. Deputies 1→4 allow third-party applications to manipulate data in the multiple files managed by the `AccountManagerService`. For the purposes of this discussion, we focus only on the key database files of `/data/system_de/0/accounts_de.db` and `/data/system_ce/0/accounts_ce.db` as they are the main source of the vulnerabilities. Both database files have ownership `system:system`.

In a multi-user system, `removeSharedAccountAsUser` and `renameSharedAccountAsUser` (CVE-2020-0208 and CVE-2020-0209) both allow any application running on the current user to remove or rename the shared accounts (e.g., Facebook) of that user. Account manipulation is typically restricted to the application that manages the account. However, through these deputies, any application can modify the shared account data in the two database files, data which requires the caller to have the system level permissions `MANAGE_USERS` and `CREATE_USERS` to create. Similarly, `invalidateAuthToken` and `updateCredentials` both manipulate data in these databases, allowing any application to affect portions of the databases.

Deputy 5, `add` in the `DropBoxManagerService`, allows any caller to clear system logging information which may hide evidence of an attack. A call to `add` allows the caller to write a data packet of any form to a temporary log file in the `/data/system/dropbox` with ownership `system:system`. However, continuous calls to `add` cause the deputy to clear older logging files when the device is low on space or the number of files exceeds 1000. As such, an attacker could hide evidence of their attacks by manipulating this deputy.

We note that Deputy 5 was previously independently identified by Invetter [47] via an alternative analysis approach designed to study input validation. Whether or not Deputy 5 is a vulnerability is also subjective. The RPC documentation indicates that it is designed to discards logged data after reaching 1000 files. However, the service is designed to only be accessed by other system services, as evidenced by the documentation and the system permissions checked by the other RPCs (`READ_LOGS` or `DUMP`). We include it in our vulnerability list, because it allows third-party applications to perform this manipulation.

5.3.2 VC2: Data Leaks

FRED identified 5 deputies (6→10 in Table 6) that leak data from security sensitive files to third-party applications. `getSharedAccountsAsUser` (CVE-2020-0210), like 1→2, affects data in the databases managed by the `AccountManagerService`. It allows any application running on the current user to get a list of the accounts for that user, an action that is typically restricted to the application that manages the account. Similar to 3→4, `getPreviousName` and `isCredentialsUpdateSuggested` both manipulate data in the same databases. Finally, deputies 9→10, return information about security sensitive files, with `getTotalBytes` returning the size of `/data` and `isStorageLow` returning if `/data` is low on space. `/data` has ownership `system:system`.

5.3.3 VC3: Denial of Service

FRED identified 2 deputies (11→12 in Table 6) that result in a local temporary denial of service. Both of these deputies are unique to the Samsung firmware and exist within the `BlockchainTZService`. In both cases, the Java RPC entry points (`sspInit` and `sspExit`) contain no authorization logic and immediately call JNI methods (`nativeSspInit` and `nativeSspExit`). Our native code analysis identified that these JNI methods both access the `/dev/ssp` device node, which is assigned the user and group `system`. We were unable to reverse

Table 6: A description of vulnerabilities, along with the services in which they are present.

Deputy (RPC Service)	Impact	Firmware	File Path	Vulnerability Description
VC1: Data Manipulation				
1. `removeSharedAccountAsUser` (`AccountManagerService`)	Moderate (CVE)	AOSP	Static	Missing check `isAccountManagedByCaller` allows apps of the current user to remove shared accounts they do not manage.
2. `renameSharedAccountAsUser` (`AccountManagerService`)	Moderate (CVE)	AOSP	Static	Missing check `isAccountManagedByCaller` allows apps of the current user to rename shared accounts they do not manage.
3. `invalidateAuthToken` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows callers to invalidate an authorization token for an account they do not manage.
4. `updateCredentials` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows callers to update the credentials of an account they do not manage.
5. `add` (`DropBoxManagerService`)	Minor	AOSP	Static	A missing permission check for writing to system log files allows any app to record system restricted logging information.
VC2: Data Leaks				
6. `getSharedAccountsAsUser` (`AccountManagerService`)	Moderate (CVE)	AOSP	Static	Missing check `isAccountManagedByCaller` allows apps of the current user to get shared accounts they do not manage.
7. `getPreviousName` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows any caller to get the previous name for an account they do not manage.
8. `isCredentialsUpdateSuggested` (`AccountManagerService`)	Minor	AOSP	Static	Missing check `isAccountManagedByCaller` allows non-managing callers to see if the account credentials should be updated.
9. `getTotalBytes` (`StorageStatsService`)	Minor	AOSP	Static	Missing check for `PACKAGE_USAGE_STATS` allows any app to get the size of system restricted directories `/system` and `/data`.
10. `isStorageLow` (`PackageManagerService`)	Minor	AOSP	Static	A lack of any authorization checks allows any app to determine if the system restricted directory `/data` is low on space.
VC3: Denial of Service				
11. `sspInit` (`BlockchainTZService`)	Moderate (CVE)	Samsung	Static	Missing privilege check allows apps to perform temporary denial of service of the `/dev/ssp` device node.
12. `sspExit` (`BlockchainTZService`)	Moderate (CVE)	Samsung	Static	Missing privilege check allows apps to perform temporary denial of service of the `/dev/ssp` device node.

engineer the logic in the native library. We informed Samsung of the potential vulnerability, and they confirmed that the RPC entry points were indeed missing checks. Samsung indicated that the missing checks allow a third-party application to start or terminate the `BlockchainTZService`, leading to a local temporary denial of service of the `/dev/ssp` device node. Samsung assigned two CVEs (CVE-2021-25459 and CVE-2021-25460) and added SELinux policy to ensure only privileged callers can access the `BlockchainTZService`.

5.3.4 Non-Vulnerable RPCs

Of the 23 candidate RPCs for AOSP 10.0.0, 13 were determined to be non-vulnerable. As outlined in the inspection methodology (Appendix B), these were eliminated for two reasons. (1) The arguments passed in or actions of a RPC caller could not effect modifications made to the files by the RPCs or the data returned by the RPCs from the files. (2) The RPCs were determined to be operating correctly by delegating restricted access to a more sensitive file. The same reasoning was used to eliminate the additional 8 candidate RPCs for the Google device. We also applied the reasoning to the additional 35 candidate RPC entry points for the Samsung device; however, the significant amount of new functionality restricted the manual inspection. We focused efforts on RPCs and files with interesting names, but found many new types of Samsung-specific authorization checks, which prevented improper file re-delegation.

6 Limitations

FRED relies on ACMiner [22] and retains many of its limitations. These limitations include the limitations shared by static analysis tools of Android (e.g., native code, runtime modifications, reflection, dynamic code loading, and Message Handlers), the inability to reason about authorization check ordering, and manually defined input. In addition to the manually defined input required by ACMiner, FRED requires a domain expert to define the file methods used for file path extraction. Additionally, as file access control policy is sometimes defined at runtime, FRED can only detect re-delegation instances for files that exist on the device at the time the file system dump is produced. Finally, as the extracted file paths sometime depend on runtime generated values (e.g., values from databases, RPC callers, package information, etc.), FRED approximates these values by replacing them with `.*` in the regular expressions. We found these approximations minimal effect on FRED's ability to detect re-delegation.

7 Related Work

Nearly all modern operating systems rely on privileged deputies to protect security sensitive resources while providing an interface for indirect access by lesser-privileged software. As such, it is no surprise that confused deputy vulnerabilities [25] are a classic software security problem. Felt et al. [19] were the first to directly discuss confused deputies in the context of Android, introducing the concept of permission re-delegation (though Davi et al. [11] had previously

identified privilege escalation attacks). However, these and other works [13, 24, 31, 46] largely considered permission re-delegation in cases where the deputies are applications. Performing static program analysis of Android applications is significantly easier than the Android framework, and there are many tools [5, 20, 21] publicly available for doing so.

ARF [23] was the first tool to consider permission re-delegation in the Android framework, which contains a highly interconnected collection of system services that frequently call RPC interfaces in one another. This work identified that this high degree of interconnection combined with frequent changes in ambient authority can easily result in confused deputy vulnerabilities. However, ARF does not directly consider files as protected resources. Rather, it considers the protection policy for each RPC entry point and identifies when a caller entry point is not at least as restrictive as a callee entry point. Similar to FRED, ARF is built upon ACMiner [22], which identifies all of the RPC entry points in the Android framework and extracts the access control checks that have been hard-coded by developers. ACMiner's design shares much in common with Kratos [38] and AceDroid [2]. These three tools seek to identify missing checks at RPC entry points by using consistency as an approximation for correctness, as there is no ground truth for the access control specification. Invetter [47] combines machine learning and static program analysis of Android system services to identify insecure input validations, which represent service-specific sanity checks when performing operations. A number of their findings relate to missing permission checks in RPC entry points. Finally, FANS [30] fuzzes entirely native system services; however, it does not consider access control policy.

System service RPC entry points are similar to, but different than the Android APIs called by third party applications. These so called "public APIs" are wrappers around binder IPC code that sends messages to a subset of the overall RPC entry points. Mapping public Android APIs to permissions is a well researched topic. Stowaway [16] fuzzes the APIs to determine which permissions are required. PScout [6] uses static call graph analysis of system service code. This approach is refined by Axplorer [7], which uses more detailed program analysis. Most recently, Arcade [3] improved the mapping by including logic requirements (e.g., permission *A* or permission *B* vs. permission *A* and permission *B*). In doing so, both Axplorer and Arcade provide enhanced techniques to statically analyze code for RPC entry points.

Finally, prior work has also considered file access control policy in Android. Zhou et al. [49] compared Unix permissions across device vendors, identifying misconfigurations. The introduction of SELinux for Android [40] gave researchers a new target for analysis [27, 36, 37]. EASEAndroid [45] uses machine learning to classify SELinux audit log events as benign or malicious. SPOKE [44] uses compatibility tests to approximate least-privilege execution and identify over-permissive SEAndroid policy. Chen et al. [10] and BigMAC [26] compare Unix permissions and SEAndroid policy to identify access control policy flaws. FRED's mapping between RPC entry points and files is a useful input to consider in conjunction with SEAndroid policy analysis.

8 Conclusion

The Android framework prevents untrusted third-party applications from directly accessing sensitive resources by using system services as privileged deputies. However, since access control is hand-coded, there is potential for confused deputy vulnerabilities. This paper proposed FRED, a static program analysis tool that maps Android's Java-based RPC entry points to the security-sensitive files they access. By contrasting permission checks by RPC entry points with the corresponding file access control policy, FRED identifies candidate RPC entry point that perform re-delegation. We used FRED to study three devices running Android 10 and identify 10 confused deputy vulnerabilities, three of which were assigned moderate severity CVEs by Google. These results demonstrate the utility of semi-automated approaches to discover subtle flaws in access control enforcement.

Acknowledgements

This work was supported in part by the Army Research Office (ARO) grant W911NF-16-1-0299 and a Google ASPIRE award. Opinions, findings, conclusions, or recommendations in this work are those of the authors and do not reflect the views of the funders.

References

[1] SETools: SELinux Policy Analysis Tools. https://github.com/SELinuxProject/setools, 2021. Accessed Jun. 7, 2021.

[2] Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2018.

[3] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise Android API Protection Mapping Derivation and Reasoning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2018.

[4] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. SoK: Lessons Learned from android Security Research for Appified Software Platforms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016.

[5] 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. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[6] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.

[7] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *Proceedings of the USENIX Security Symposium*, August 2016.

[8] David Barrera, H. G unes Kayacik, Paul C. van Oorshot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, October 2010.

[9] George Burgess. FORTIFY in Android, April 2017.

[10] 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)*, December 2017.

[11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC)*, October 2010.

[12] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[13] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium*, August 2011.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[15] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security & Privacy Magazine*, 7(1), January 2009.

[16] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[17] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdata Akhawe, and David Wagner. How to Ask for Permission. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, 2012.

[18] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[19] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, August 2011.

[20] Xinming Ou Fengguo Wei, Sankardas Roy and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

[21] Michael I. Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2015.

[22] Sigmund Albert Gorski III, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. ACMiner: Extraction and Analysis of Authorization Checks in Android's Middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy (CODASPY)*, March 2019.

[23] Sigmund Albert Gorski III and William Enck. ARF: Identifying Re-Delegation Vulnerabilities in Android System Services. In *Proceedings of the 12th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, May 2019.

[24] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the ISCO Network and Distributed System Security Symposium (NDSS)*, February 2012.

[25] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[26] Grant Hernandez, Dave (Jing) Tian, Anurag Swarnim Yadav, Byron J. Williams, and Kevin R.B. Butler. Bigmac: Fine-grained policy analysis of android firmware. In *Proceedings of the USENIX Security Symposium*, August 2020.

[27] Bumjin Im, Ang Chen, and Dan S. Wallach. An Historical Analysis of the SEAndroid Policy Evolution. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2018.

[28] Donald B Johnson. Finding All The Elementary Circuits of A Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[29] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java Program Analysis: A Retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS)*, October 2011.

[30] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan4, and Jianwei Zhuge. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. August 2020.

[31] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[32] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Pen Liu. System Service Call-oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

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

[34] Oracle. Obtaining Names of Method Parameters. https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html, 2019. Accessed Jan. 15, 2019.

[35] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[36] Elena Reshetova, Filippo Bonazzi, and N. Asokan. Selint: an seandroid policy analysis tool. *CoRR*, abs/1608.02339, 2016.

[37] Elena Reshetova, Filippo Bonazzi, Thomas Nyman, Ravishankar Borgaonkar, and N. Asokan. Characterizing seandroid policies in the wild. *CoRR*, abs/1510.05497, 2015.

[38] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[39] 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.

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

[41] Vincent F. Taylor and Ivan Martinovic. To Update or Not to Update: Insights From a Two-Year Study of Android App Evolution. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, April 2017.

[42] Liam Tung. Bigger than Windows, bigger than iOS: Google now has 2.5 billion active Android devices. https://www.zdnet.com/article/bigger-than-windows-bigger-than-ios-google-now-has-2-5-billion-active-android-devices-after-10-years/, May 2019.

[43] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - A Java Bytecode Optimization Framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research*, November 1999.

[44] 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)*, April 2017.

[45] Ruowen Wang, William Enck, Douglas Reeves, Xinwen Zhang, Peng Ning, Dingbang Xu, Wu Zhou, and Ahmed Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In *Proceedings of the USENIX Security Symposium*, August 2015.

[46] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 623–634, 2013.

[47] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the ACM Conference on Computer and Communcations Security (CCS)*, October 2018.

[48] Yury Zhauniarovich and Olga Gadyatskaya. Small changes, big changes: An updated view on the android permission system. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[49] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Proc. of the IEEE Symposium on Security and Privacy*, 2014.

A Details of Java Path String Reconstruction

There are various scenarios when the use-def analysis for a *PlaceHolder* node does not find a constant at the definition. The following describes the special cases handled by FRED.

String Builders: The Java compiler translates inline string concatenation into calls to the `append()` method of the `StringBuilder` and `StringBuffer` classes. While these classes have many methods, we empirically found that only the `append()` method occurred during our data flow analysis of file paths. Furthermore, these instances were the result of inline string concatenation, which allows FRED to assume all `StringBuilder` and `StringBuffer` objects are only used within the method they are declared. Therefore, FRED handles these classes using an *Append* node.

Path Builders: Android provides two similar APIs for file path construction: `buildPath()` in `android.os.Environment`, and `get()` in `java.nio.file.Paths`. Both methods take a root file path and an array of zero or more file path parts, which are appended to the root path. For example, calling `buildPath()` with arguments `/foo/bar`, `first/part`, and `/second/part` produces the path `/foo/bar/first/part/second/part`. Empirically, we found that the target array of file path parts is generated at compile-time from a variable number of string arguments. This observation allows FRED assumes the array is defined and filled immediately before a call to `buildPath()` and `get()`. Hence, FRED can determine the values in the array. FRED uses an *Append* node to append the values, inserting a '`/`' between each part as needed.

Arrays and Collections: Extracting values from Arrays, Collections, Iterators, and Maps is a known hard problem for static analysis. Therefore, in most cases FRED represents access to them as *Any* nodes. Exceptions include the Paths and Environment scenarios.

Directory Listing: The `java.io.File` class provides five methods for listing files in the directory. Since the return value is runtime dependent, FRED appends an *Any* node to the child path of a known directory path.

Parent Path and File Name: The `java.io.File` and `java.nio.file.Path` classes provide methods for getting the parent path or file name of a existing file path. To model this functionality, FRED uses the *Parent* and *Name* nodes. These *Branch* nodes wrap an expression for the existing path, indicating that either the parent path or file name needs to be resolved once the existing path expression is known.

Environmental Variable and System Properties: Some file paths are constructed from the values of environmental variables and system properties. However, these runtime values can be looked up using a string that exists in the Java source code. Therefore, FRED uses its data flow analysis to determine the lookup string, which can be resolved later. As such, FRED encodes lookup strings into intermediate expressions using *EnvVar* and *SysVar* nodes.

Fields in Android Classes: The use-def analysis occasionally encountered class fields which are initialized outside of the path from the RPC entry point to the file method. To handle these fields, we first perform a forward def-use analysis to determine all possible assignment sites for all the fields of the Android framework. Later, during our backwards data flow analysis, FRED replaced uses of class fields with their assignment sites and resolves them normally, combining all of the possible values with a *Or* node. For any class field whose value could not be resolved, FRED uses an *Any* node (e.g., fields in `android.content.pm.PackageParser` and `android.content.pm.ApplicationInfo`).

B Manual Inspection Methodology

We used the following methodology when inspecting the candidate RPC entry points to determine if a vulnerability exists.

1. Have FRED dump call graph representations of all candidate RPC entry points to aid in the analysis.
2. Confirm that the RPC entry point is callable by a third party application by reviewing the authorization logic in the source/decompiled code. While FRED's ability determine if a RPC entry point can be called by third party applications is effective for both the AOSP and Google builds of Android 10, Samsung's unique authorization checks caused some reported RPC entry points to not be callable from third party applications.
3. Using the source code and call graph dump, for each file path, determine if the statement that opens the file path is actually reachable from the RPC entry point and is not a result of the over approximation nature of CHA call graphs.
4. For each file path of a candidate RPC entry point, determine if the file path can be influenced by the caller of the RPC entry point through arguments passed into the RPC or through other means by examining the call graph dumps and source/decompiled code. In practice, we observed no such RPCs.
5. For each file path of a candidate RPC entry point, using the source/decompiled code and call graph dumps, determine if the arguments passed in or actions of a RPC caller effect modifications made to a file by the RPC or the data returned by the RPC from a file.
6. As it is common for RPC entry points to delegate partial access to files with generally more restrictive access control policy, determine if this flow is intended and properly protected by some form of authorization logic. If the flow is determined to not be intended or properly protected by some form of authorization logic then it is considered a vulnerability, withstanding the SELinux policy confirmation.
7. Confirm the SELinux policy does not restrict (a) the RPC entry point from accessing the file path and (b) third-party applications from calling the RPC entry point.