BIGMAC: Fine-Grained Policy Analysis of Android Firmware

Grant Hernandez[1], Dave (Jing) Tian[2], Anurag Swarnim Yadav[1], Byron J. Williams[1], and Kevin R. B. Butler[1]

[1]Florida Institute for Cyber Security (FICS) Research, University of Florida, Gainesville, FL, USA
{grant.hernandez,anuragswar.yadav,butler}@ufl.edu, byron@cise.ufl.edu

[2]Purdue University, West Lafayette, IN, USA
daveti@purdue.edu

Abstract

The Android operating system is the world's dominant mobile computing platform. To defend against malicious applications and external attack, Android relies upon a complex combination of discretionary and mandatory access control mechanisms, including Linux capabilities, to maintain least privilege. To understand the impact and interaction between these layers, we created a framework called BIGMAC that combines and instantiates all layers of the policy together in a fine grained graph supporting millions of edges. Our model filters out paths and types not in use on actual systems that policy analysis alone would consider. Unlike previous work which requires a rooted device, using only static firmware and Android domain knowledge, we are able to extract and recreate the security state of a running system, achieving a process credential recovery at best 74.7% and a filesystem DAC and MAC accuracy of over 98%. Using BIGMAC, we develop attack queries to discover sets of objects that can be influenced by untrusted applications and external peripherals. Our evaluation against Samsung S8+ and LG G7 firmwares reveals multiple policy concerns, including untrusted apps on LG being able to communicate with a kernel monitoring service, Samsung S8+ allowing IPC from untrusted apps to some root processes, at least 24 processes with the *CAP_SYS_ADMIN* capability, and *system_server* with the capability to load kernel modules. We have reported our findings to the corresponding vendors and release BIGMAC for the community.

1 Introduction

The Android operating system is the world's most dominant mobile computing platform. As smartphones increasingly become primary computing devices, Android commands an overwhelming market share, representing 88% of all end-user smartphone sales in the second quarter of 2018 [16]. As such, assuring the security of Android devices is of paramount importance.

To this end, Android supports a complex combination of access control mechanisms. Substantial past work has examined the security model of the Android middleware layer, which mediates access decisions made by Android applications [13]. However, underneath this middleware lies a Linux kernel and at this layer, the security model is similar to that of many other Linux-based systems. The access control framework includes not only a discretionary access control (DAC) mechanism common to UNIX operating systems, but supports mandatory access control (MAC) as well through Security-Enhanced Android (SEAndroid) [37], a customized version of SELinux. Moreover, other mechanisms such as Linux capabilities are also used to assure least privilege and to minimize trusted processes. To maintain the integrity of an Android system against local and remote adversaries, all of these mechanisms must work in conjunction with each other; however, the complexity of creating and interacting with SEAndroid policies, MAC rule interactions, labeled filesystems, capabilities, and DAC policies creates a challenging environment for assuring security goals.

In this paper, we introduce a framework called BIGMAC for reasoning about the complex Android policy environment based on extracting policies and metadata from Android device firmware images. BIGMAC goes beyond the analysis of the SEAndroid MAC policy to consider how policy is instantiated on devices through processes, objects, and inter-process communication endpoints, SELinux type relationships between Android objects, filesystem extended attributes, and Linux capabilities. BIGMAC recovers system init scripts and simulates the behavior of commands that affect the filesystem in order to accurately model filesystem contexts and Android credentials. We demonstrate that compared to the ground truth provided by a running Android device, BIGMAC successfully labels over 98% of DAC and MAC filesystem metadata, and up to 74.7% of running process metadata can be identically reconstructed.

BIGMAC is valuable as a means of determining potential vectors of attack based on policy inconsistencies found, and we demonstrate that past exploits can be modeled with it. We are also, to our knowledge, the first to model the impact of external peripherals (e.g., USB interfaces) and their relation to a device's security policy. Supporting millions of access

control edges, BIGMAC provides one of the finest-grained frameworks for policy examination currently available for reasoning about access control on Android devices. To benefit the community, we will open source BIGMAC and all of the collection scripts for reproducibility.[1]

Our contributions are as follows:

- We develop a new framework, BIGMAC, to extract, graph, and query Android security policies from static firmware images, without the need for a rooted device, and recover runtime security state by instantiating processes, files, and IPC objects with a 98% accuracy.
- We combine MAC, DAC, capabilities, and tagged external input sources to create an instantiated, fine-grained, whole-system attack-graph supporting millions of edges. Our Prolog engine then provides an interactive user interface to query the attack graph.
- We evaluate BIGMAC against Samsung S8+ and LG G7 firmware and discover apps on LG able to communicate with a kernel monitoring service, Samsung apps able to talk to multiple root processes via binder, at least 24 processes with the dangerous *CAP_SYS_ADMIN* capability, and *system_server* being able to load kernel modules. We reported our findings to the corresponding vendors.

Outline The rest of this paper is structured as follows: Section 2 develops the necessary background on Android security policies. Section 3 describes the design of BIGMAC and Section 4 the implementation. In Section 5 we evaluate BIGMAC against multiple Android firmware images and demonstrate how it can assist in discovering privilege escalation paths. We discuss our findings in Section 6, compare key related work in Section 7, and finally conclude in Section 8.

2 Background

Linux Access Control Historically, access control has been implemented using object ownership, group membership, and their respective permission bits for read, write, and execute. On modern UNIX inspired systems, such as Linux, this model persists with the creation and assignment of User IDs (UIDs) and Group IDs (GIDs), along with read, write, and execute permissions for each class of identifier. Privilege separation is primarily based on different UID and GID assignments to processes and file objects. This class of access control is formally known as Discretionary Access Control (DAC) because the permissions are assigned to new and owned objects at the discretion of the users (i.e., subjects or processes). This means that the system's access matrix is not fixed at runtime and may change dynamically, possibly leading to a loss of integrity. The all-powerful root user with the UID of zero has hard-coded superuser permissions in the kernel's code, allowing it to override all DAC permissions at will. For example, a root-owned process can accidentally create a world-readable and writable file that it trusts to be well-formed, but a lower privileged user could take advantage of this hole to affect the runtime state of the root owned process, possibly leading to confused deputy style attacks or even a privilege escalation.

To help limit the damage of a root process compromise, the Linux kernel divided up root's permissions using a Capability-like (CAP) system [22]. Currently there are 38 capabilities with a wide-range of strengths. This works under the model that processes that usually need root-like permissions do not require all of its extensive set of capabilities. As such, system administrators are free to permanently drop any capabilities from processes that do not need them. For example, a process, such as Apache HTTPD, that needs to listen on privileged ports (those less than 1024) can only be granted the `CAP_NET_BIND_SERVICE` capability instead of all 38. This capability model works some applications, but its lack of granularity can still lead to over-permissioned processes, which can be dangerous in the face of untrusted processes and users.

To further lock down Linux-based systems, a Mandatory Access Control (MAC) scheme can also be deployed. MAC unlike DAC has the last word on actions taken on objects and is fixed at runtime (e.g., its access matrix is fixed). This has the major advantage in that the policy can be written and verified statically before being applied to a running system and set to immutable. There have been many MAC systems proposed in the literature, but today, the most popular implementation of MAC on Linux-based systems is SELinux [38]. SELinux has three core components: subjects, objects, and actions. The subject is an active process or device that is responsible for the flow of information between objects. Subjects are given access to objects via "allow" rules that permits the process to read or modify the object, leading to changes in the system state. Objects are resources such as files, sockets, and network interfaces which are accessed by subjects (i.e., active processes and devices) and classified according to the resource they provide. Every object has a set of permissions and a class identifier defining its purpose and services that the object handles. These allow rules form a security policy that ideally grants the minimal set of permissions required to complete a task (e.g., read only when writing permission is unnecessary). Both subjects and objects have a set of security attributes which the operating system can query to determine whether a requested action is allowed by the policy or not.

2.1 Android Security Model

Today, Android OS's kernel uses a modified version of Linux that benefits from the above access control implementations.

DAC on Android Android's use of DAC involves a fixed set of UID/GIDs for system purposes and ranges of UIDs

[1] https://github.com/FICS/BigMAC

for dynamically installed applications. On Android, a strong effort is made to limit the number of processes running as root, and as such the typical high privileged process is usually running as the `system` (UID 1000) user or a role-specific UID, such as `radio`, or `graphics` instead. This denies most running processes access to powerful root permissions, unless they have been explicitly granted or inherited the capabilities. At the app level, untrusted applications are each assigned a unique UID from a fixed range, preventing them from owning any files besides the ones included during their installation.

SEAndroid SELinux was introduced to the Android platform with a targeted policy in 2013 [37]. These policies are a set of rules which are based on file labels. These labels consist of a user, role, type, and level. On Android, the type is the primary identifier. Policies are rules which determine what types and actions a process has access to. From a security perspective, items are grouped together based on their accessibility.

SEAndroid extends SELinux to support new Android specific hooks, such as Binder IPC. Additional extensions include adding access classes for services, properties, etc. They also include modifying user-space configuration files and processes, as shown in Figure 1, to help apply security labels to Android-specific objects. Besides these changes, the core functionality is largely unchanged, with the primary goal being mandatory, system-wide, type enforcement. The *Binder* implementation introduced new LSM hooks to the Binder driver which lets SEAndroid monitor and control interprocess communication between applications.

The SEAndroid user space introduced modifications to existing SELinux components including: init, Zygote, bionic (the Android C library), and the package manager. The Android init process is responsible for loading the security policy early during the boot process, interpreting *init.rc* commands without access to the shell, and enforcing aspects of the security policy (i.e., access to system properties). The Zygote process is responsible for spawning Android application processes. Zygote starts when the system boots and loads common framework code. Zygote can then set the security label of the socket interfaces and the security context for running applications. Bionic, Android's specific libc implementation, was modified to get and set extended file system attributes and to store file security labels. The package manager makes decisions on permissions requested by an application to determine if requested permissions can actually be granted. These user space components enforce SEAndroid security policy beyond the kernel layer [37].

Middleware Application writers are abstracted away from the system access control models and instead focus on Android middleware permissions. These permissions are capability-like, but instead grant applications access to resources and services provided by the Android operating system, not the Linux kernel. For the purposes of this paper, we do not examine

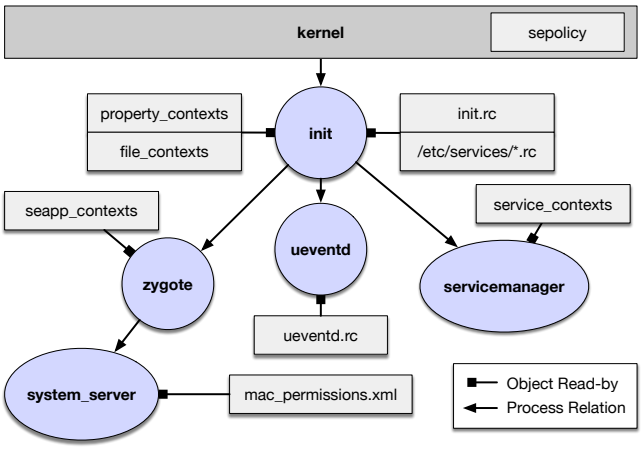


Figure 1: The relationship between key Android processes and their DAC and MAC configuration files.

Android middleware permissions and instead focus on the combination of DAC, MAC, and capabilities and their impact on the security of the system.

3 Design

To examine and query the security policy of an Android firmware image, we design BIGMAC, as shown in Figure 2, to extract, recover, simulate and fully instantiate all objects, which include files, processes, and Interprocess Communication (IPC) end-points. This method is realized using a scalable approach that combines Android domain knowledge and files extracted from firmware images to recreate a running system's state. The recreation is possible without requiring complicated emulation techniques that may not work on all device-specific firmware images or on a diverse corpus of physical devices, of which no rooting method may be available.[2]

To begin, we describe how we extract MAC, DAC, and capability (CAP) information from firmware images, including saving key files, simulating Android's init daemon, and associating raw files with SELinux file types and domains. Next we explain our dataflow graph which processes an SEPolicy's access vectors into a simplified read/write model. Then we discuss how we recreate a running system's process hierarchy and process metadata from abstract SELinux domains. Following this discussion, we show how we flatten our dataflow graph into *true objects* and expand our process tree into actual processes by propagating credentials using a fork/exec model. Finally we overlay the dataflow graph onto the process list to create an attack graph that contains all of the possible read/write interactions between processes (e.g., IPC) and the filesystem (e.g., files). Once we have a completed attack graph, we convert this to facts for our query engine to explore the data

[2] If a rooted device *is* available (such as on a developer phone), BIGMAC can work from runtime security policies too.

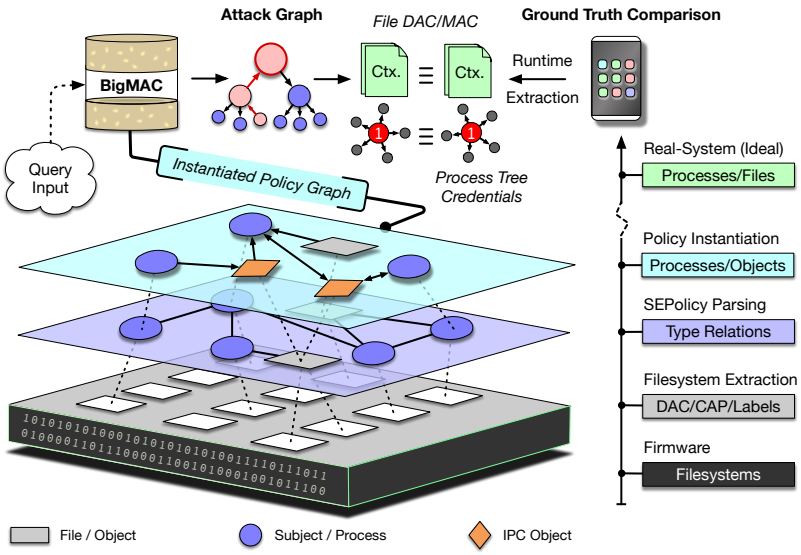


Figure 2: An overview of BIGMAC's layers of abstraction from device firmware all the way to kernel object recovery.

paths between processes and the filesystem.

3.1 Security Policy Extraction

The Android ecosystem is made up of many Original Equipment Manufacturers (OEMs), each with their own proprietary extensions to the Android Open Source Project (AOSP). This fragmentation also extends to the Android image distribution formats, which vary wildly between OEMs. To canonicalize firmware images, we use an open source Android image extractor [39]. With the extracted images, we walk the filesystems to extract the DAC file permissions and extended attributes (xattrs), which include SELinux MAC labels and Linux capabilities. We save the filesystem metadata and extract all of the files shown in Figure 1 which include Android object SELinux associations (services, properties, and apps), the Android property list, the raw SEPolicy binary file, and finally the init system files which contain useful DAC/MAC/CAP metadata, plus a list of native daemons started at boot.

Init Boot Simulation Without simulating a device boot, we do not recover any policy information for the */data*, */dev*, and */sys* directories – all of which are crucial to model. These directories and their files do not exist in the static firmware as they are created during the boot process. In order to capture these changes to the extracted security policy, we parse the saved init system files, following the Android init daemon's specification [17].

Android's init system is custom to the platform and has two major components: the *init* daemon itself, which is responsible for starting and managing native daemon services, and the *uevent* daemon, which monitors the kernel for device state change notifications. Init is the first process executed on the system and it handles boot state changes, manages services, and executes Run Commands (RC). These RCs include triggers for handling boot and property change events, actions which are executed in a trigger, and service definitions. During different phases of the boot process, files are created and services are started. BIGMAC implements key RCs that affect the security state to capture these changes before proceeding further with the graph instantiation. These include the creation of directories, files, and the changing of file owner, group, and permission modes. Additionally, we save the service definitions for later spawning of processes and simulation of their runtime credentials.

Early in the boot process, init will spawn the *uevent* daemon. This sets customized DAC information for certain files in the */dev* and */sys* directories based off of a simplified configuration file. The format is a single entry per line, with optional glob match, containing a user, group, and permission. We heuristically apply these customizations to the filesystem in order to have properly labeled and credentialed objects for these directories.

Backing File Recovery In order to fully instantiate abstract, MAC-only, SELinux types into processes and objects, we need to associate them with concrete files containing DAC and capability information. To start, we decompile the binary SEPolicy back into a connected multi-edge directional graph representation, G_s, via the Access Vector rules (AVRules), which link a source and target type via an action and a class. As shown in Figure 3, G_s, is the subject graph and the genesis of all derived graphs. This policy includes all of the types, T, and attributes, A, used during SELinux type enforcement. From this abstract policy, we divide all of the types into subjects (domains), S, and objects, O, and begin to instantiate them by correlating their

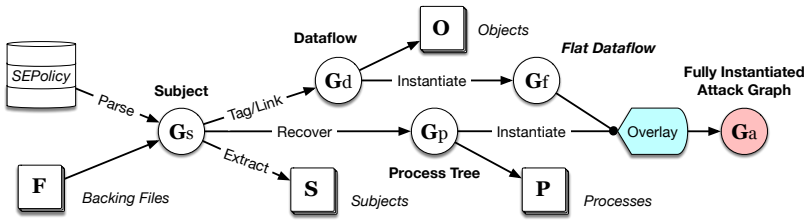


Figure 3: The key graph derivations used by BIGMAC to create a fully-instantiated attack graph, G_a.

policy types to real files, F, on the filesystem (backing files). For file objects this is straightforward as their types directly link to filesystem object types captured during extraction. To do this for subjects, we must invert Type Enforcement rules (TERules) related to process transitions via an object execution. A TERule connects two domains S_i and S_j via a transition using a 2-tuple containing object O_j and class C_p: $S_i \xrightarrow{(O_j,C_p)} S_j$. For process transitions, this reads as domain S_i uses the object O_j, which is an instance of F_j, to transition to the domain of S_j via the class C_p, which is a *process*. This is an *explicit* encoding of a process transition during the `exec` of a binary F_j. By parsing these rules, we can back-propagate the binaries associated with subject types. We define the set of subjects with at least one backing file to be *SB*. Subjects may have more than one backing file, but if they have none, they cannot be fully instantiated. Example rules from an SEPolicy to allow *init* to execute *mediaserver* are:

```
allow init mediaserver_exec:file {open,
    read, execute};
allow init mediaserver:process
    {transition};
type_transition init
    mediaserver_exec:process mediaserver;
```

Where S_i = init, S_j = mediaserver, O_j = mediaserver_exec, and C_p = process.

Not all Android domains have an explicit TERule due to a type of AVRule called *dyntransition*. A dynamic transition is encoded from a subject to another subject via the *process* class. It is the same as the TERule, minus the backing object, which means a subject does not need to `exec` to change its MAC label. The common Android platform domains that perform these transitions are init and Zygote. Zygote is started by init upon boot and is used as a warmed-up virtual-machine process source to fork app processes. As such, depending on the class of app (untrusted, system, privileged, etc.) the new forked process will be dynamically transitioned to a new domain. Applications are dynamically installed by the user and as such cannot have hardcoded TERules. This leads to all applications of a certain class sharing the same security label, unless customized by `seapp_contexts`. By recovering these dynamic transition edges, we are able to recreate the entire subject type hierarchy for later use in recovering the instantiated process tree.

3.2 Dataflow Graph

Now with a full set of subject nodes, S, and those that have concrete backing files, *SB*, we begin to create a dataflow graph, G_d, from the subject graph, G_s. Figure 3 shows this and the relationships between the other derived graphs of BIGMAC. In our model, we want to capture the dataflow between subjects as we are primarily interested in privilege escalation attacks, which involve other higher privileged subjects. To be clear, our dataflow model captures the potential transfer of bytes from one node to another following a directed edge. In the raw SEPolicy, each AVRule contains individual multiple access vectors, such as read, write, open, getattr, and so on. For our model, we are only interested in the vectors that imply a read or a write. As such we define a reduction mapping of AVRule actions to the simple dataflow properties of read (R) or write (W) as is listed in Table 8 in the Appendix.

As we are interested in an instantiated graph, we do not include any S not in *SB*. Without at least one associated executable, the subject is considered abstract. Therefore, for each *SB*, we examine all AVRules to other S or O nodes. We also ignore subject to subject edges and instead infer an implicit object to maintain a strict object-subject edge invariant.[3] Our model divides objects into two sets: IPC objects O_{IPC} and file objects O_{File}. The key difference between objects is that IPCs have a single owning subject (the creator/manager), inheriting its credentials, while file objects contain one or more backing files, F_i, each with separate MAC/DAC/CAP data. This split allows us to selectively and intelligently instantiate objects depending on their usage and context. For IPC objects we consolidate many individual AVRule classes into this object and tag it with the underlying AV class. For instance, all classes derived from *socket* commons class and Android's custom *binder* class are all considered IPC objects. Android defines specific AV classes for use in middleware, including *property service* and *service_manager*. For the property service, which in Android's case is the init process, we omit these flows as they allow for too many easy, and less likely to be exploitable paths to init from many processes. For the service manager, we examine the AVRules that have the AV of *add*, which allows us to find the owning domain. This is important for having correct service IPC credentials and edges. Once we have inferred an

[3] The only subject-subject edges are self-edges (domain can interact with itself), file descriptor transfer (`fd:use`), and ptracing.

object from an AVRule edge, we insert it into G_d with a read and/or write edge to the adjacent subject.

As all subjects and objects are iterated through, we also expand attributes A_s. These attributes contain members that are common to all AVRules. Instead of leaving these attributes linked into the graph, we fully expand all of their edges into each member. This means we have a large increase in edge count, but no longer have to consider attribute membership during graph queries. This graph is bipartite and in the worst case has $\mathcal{O}(|S|*|O|)$ edges and $\mathcal{O}(|S|+|O|)$ nodes.

3.3 Process Inflation

Using the subject hierarchy and backing files we recovered in Section 3.1, we can now begin to fully instantiate the subject nodes into individual process instances with virtual Process Identifiers (PIDs). Starting at the root *kernel* subject, we fork the *init* subject and assign the starting credentials of root (uid=0, gid=0, groups=[], sid=u:r:init:s0, cap=ALL). The kernel is assigned a PID of zero and *init* a PID of one. From here, we bring back the service definitions extracted in Section 3.1 in order to examine and filter all the children of *init*. For each child subject of *init* and for each backing file of that subject, we lookup a service definition that matches the backing file path in our virtual filesystem, is enabled, and not a oneshot process. With a potential match, we use the service security options, if any, to determine the process's user, group, supplemental groups, credentials, and security identifier. The defaults if these aren't included are to fork a service as root with all capabilities [17].

Once a process has been assigned credentials, it is inserted into a concrete process tree, G_p, with proper Linux process semantics (parent, children, inherited credentials, etc.). With boot now simulated, we employ Android domain knowledge to fix the processes forked by Zygote. These include apps and the *system_server*. The credentials for the system server are hardcoded in the Zygote source code and we apply them manually.

3.4 Attack Graph Instantiation

To begin our final instantiation, we must first fully expand all object nodes within G_d. Specifically, we split all file nodes into individual file instances. For example, the *system_data_file* object has many backing files. To make sure that each file's DAC information is considered, we split this node while duplicating edges to and from the original adjacent nodes. This greatly increases the edge count, but allows for a concrete yes or no answer for future DAC and MAC queries. We call this new graph G_f. With the two primary input graphs, G_f for flattened dataflow and G_p for the process tree we, for each process P_i of subject S_j in G_p, we copy all in and out edges from S_j in G_f to P_i in the process tree. We effectively overlay the dataflow graph onto the process tree, giving the concrete processes concrete edges to all objects allowed to be read from or written to by the original MAC policy. This final graph, G_a, is used to generate facts for all of our future attack queries using the logic engine.

3.5 Logic-based Query Engine

To explore our BIGMAC attack graph, we design a Prolog-based query engine as the backend, providing multiple query interfaces to end users, as shown below:

```
query_mac(S,T,C,P).
query_mac_dac(S,T,C,P).
query_mac_dac_cap(S,T,C,B,P).
query_mac_dac_cap_ext(S,T,C,B,E,P).
```

$\mathcal{S}$ represents the starting node; $\mathcal{T}$ stands for the target node; $\mathcal{C}$ is the cut-off parameter used to limit the length of a path; $\mathcal{B}$ specifies the target Linux capability; $\mathcal{E}$ determines the external attack surface type, such as USB and Bluetooth; and $\mathcal{P}$ contains the returned paths of the query. Both the starting node and the target node can be a process or a object (e.g., a file or IPC). They can also be wildcards, which is represented by underscore (_) in Prolog. Each query interface applies different policy layers and filtering mechanisms. For example, `query_mac_dac_cap_ext(_,zygote,3,CAP_SYS_ADMIN,usb,P).` requests all (attack) paths which target the Zygote process, pass both MAC and DAC checking, have a maximum path length of 3, achieve CAP_SYS_ADMIN capability, and can be launched from USB connections. As the attack graph is built upon the MAC policy, each viable path within the graph is allowed by the MAC. To support different query interfaces, we apply different policy and filtering checks on a viable path. For instance, *query_mac_dac_cap_ext* predicate finds a viable path within the graph, such that:

```
find_a_path(S,T,C,B,E,P) :-
  graph_travel(S,T,[S],P,C),
  dac_path(P),
  cap_path(P,B),
  ext_path(P,E).
```

The *graph_travel* predicate uses Depth First Search (DFS) to find a path, which is based on the MAC policy. The *dac_path* predicate then checks the corresponding DAC policy by looking into every adjacent pair of nodes within the path using the *dac* predicate. Each pair of nodes (A,B) is a combination of a process and a system object with any ordering. The *dac* predicate checks for root user, owner, group, and others based on the DAC information within each node. We perform the following query, $\forall A,B \in \mathcal{S} \cup \mathcal{T}$, perform:

```
dac(A,B) :-
  dac_sub_obj(A,B);
  dac_obj_sub(A,B).

dac_sub_obj(A,B) :-
  is_sub(A),
  dac_sub_obj_allow(A,B).

dac_obj_sub(A,B) :-
  is_obj(A),
  dac_obj_sub_allow(A,B).

dac_sub_obj_allow(A,B) :-
```

```
    is_root(A);
    is_owner(A,B);
    group_sub_obj_allow(A,B);
    other_sub_obj_allow(A,B).

  dac_obj_sub_allow(A,B) :-
    is_root(B);
    is_owner(B,A);
    group_obj_sub_allow(A,B);
    other_obj_sub_allow(A,B).
```

The *cap_path* predicate checks if a given path can achieve a certain Linux capability by examining the last node within the path. In case of the last node being a system object, we also need to look at the previous node which is the last process node within the path. Because each process node has its capability information encoded, the final check is to see if the requested capability is contained within the capability list of the node.

```
  cap_path(P,C) :-
    cap_last(P,C);
    cap_prev(P,C).

  cap_last(P,C) :-
    last(P,A),
    is_sub(A),
    cap_supp(A,C).

  cap_prev(P,C) :-
    prev(P,A),
    is_sub(A),
    cap_supp(A,C).
```

Similarly, the *ext_path* predicate checks if a given path starts from an external attack surface, e.g., USB, by inspecting the starting node within the path. If the starting node is a system object and can be reached via external connections, the path is an attack path that can be triggered externally. Because each system object has its external connection information encoded, the final check is also a membership checking between the specific external attack and the surface list of the node.

```
  ext_path(P,E) :-
    first(P,A),
    is_obj(A),
    ext_supp(A,E).
```

4 Implementation

Our implementation of BIGMAC is based upon Python 3.5 and SWI-Prolog, and it employs the NetworkX library for graphing and the SETools package [7] [4] for decompiling SEPolicies into their original types and rules.

4.1 Firmware Extraction

BIGMAC uses an open source extraction library for the raw firmware, which supports 18 firmware vendors across many Android versions [39]. We extended this tool to support newer 8.0 Samsung images, which use the LZ4 compression on the raw disk images. For this paper, we only extracted Google, Samsung, and LG images, but we would be able to support many other vendors as the files we extract from disk images are standardized by the platform.

The extraction tool starts by recursively unzipping firmware images, handling each layer based on the file type. We hook into the `ext4` disk image extraction routine and disable the permission changes to maintain the original DAC and MAC information. Then in our separate frontend, we perform a filesystem walk of each disk image to extract out all metadata provided by the `stat`, `getxattr`, and `readlink` calls. We store this in a dictionary indexed by filename. Next we walk our in-memory Virtual Filesystem (VFS) using regular expressions to extract out every file shown in Figure 1. Additionally we extract out the `build.prop` files to resolve properties containing key metadata like the Android version and the hardware configuration. We save all of these raw files in an image-specific database for later processing. There are some quirks between different Android major versions that our tool also takes into account. Most of these involve the major changes to the security policy file splitting by Project Treble [18]. Effectively, the platform (Google) and the vendor now have separate files for all of those shown in Figure 1. Our tool handles this transparently, allowing us to analyze version 7.0 and above.

4.2 System Boot Emulation

To recover an approximate state of a running system, we combine all known policy files at various stages. The most important set of files besides the raw SEPolicy are the Android init scripts. These are text based, shell-like commands that execute sequentially in blocks, but do not support constructs for control-flow or looping [17]. This is a significant advantage for BIGMAC as we do not have to support arbitrary code and can instead selectively handle the most important commands. We implemented the `mkdir`, `chown`, `chmod`, `trigger`, `enable`, and `mount` commands. The first three change the state of the filesystem, including DAC information; `trigger` raises an event that causes other sections to be executed; `enable` starts a service; and `mount` mounts a filesystem to a hardpoint. The mount command is particularly important to handle as it affects the effective SEContext that files are assigned during our simulated `restorecon` process. During device ground-truth analysis, most of the wrong MAC/DAC data came from not handling these quirks. Each vendor and device model comes with its own set of quirks in the init system. For example, on a Pixel 8.1.0 image, we had to set the property `vold.decrypt` to `trigger_post_fs_data` in order for init simulator to execute the proper boot sections that create the */data* directory. Another quirk we discovered was that some OEMs add their own custom group and user Android IDs (AID) that deviate from Android platform. This affects our equivalent `getpwnam` function which is unable to map these names to AIDs. To fix this, we plan on extracting out an exported table, `android_id`,

[4] SETools does not recover the policy source, but it recovers the effective policy after the source has had all of its macros expanded, comments removed, and `neverallow` rules checked.

from the Bionic `libc.so` binary.

4.3 Android Credential Simulation

After booting init and modifying the filesystem, we use recovered service definitions from the init files to properly assign runtime credentials to inflated processes. Service definitions contain their starting user, group, and capabilities. These are instrumental in getting accurate DAC and CAP information and improving the overall fidelity of our attack graphs. From the spawned children of init, we also perform a conservative reparenting of children processes that have no service definition, yet are able to be transitioned to by the init daemon. This helps fix the process tree in case a child of init also forks its own helper processes, which we noticed with the `location` service (`loc_launcher`). Overall, the credential simulation is a best-effort approximation given the available information and constraints recovered from the MAC policy.

4.4 Logic-based Query Engine

We implement our logic-based query engine using SWI-Prolog[5] and provide all four query interfaces described in 3.5. To parse the output from Prolog (a list of paths), we created an interactive query frontend using the GNU readline library. This gives us support for interactive tab completion of all the possible processes and objects to use as source and target nodes. It also allows us to save, restore, and pretty print all queries. Within Prolog, we needed to add a cutoff parameter to prevent path explosion. The cut-off parameter (C) drops the path if its length is beyond the value specified. We also implement an *is_uniq* predicate to filter paths which contain duplicated elements, making sure that paths within the length of the cut-off also do not contain loops. To collect all possible paths, we use Prolog's Depth First Search (DFS) with iterative deepening ($findall$). In total, we wrote 5,997 lines of Python across the extraction tool, graph creator, and query frontend. Our Prolog engine was 343 lines.

5 Evaluation

Our evaluation is broken up into two sections: a *ground truth study* and *case studies* of BIGMAC on Android firmware. The ground truth study's purpose is to evaluate BIGMAC's accuracy in recovering security policy from static firmware. This part is limited in the Android devices and vendors because we require real rooted devices to compare against. The case studies have no such limitations as BIGMAC works on firmware. As such, we evaluate images from Samsung, LG, and Google. For reproducibility of our results, all of the firmware used is described in Table 9 of the Appendix.

[5] We initially used GNU Prolog but it crashed due to large edge counts.

5.1 Ground Truth Comparison

To evaluate the feasibility of recreating the Android runtime system state from firmware, we use custom scripts loaded via `adb` to collect MAC, DAC, and process information from a rooted Google Pixel 1 and Samsung Galaxy S7 edge. For the Pixel, we reflashed and rooted it with three different AOSP versions: Android 7.1.2, 8.1.0, and 9.0.0.

For the file and process recovery tables, we have marked metrics as True Positive (TP, found and accurate), False Positive (FP, recovered file not accurate or it was extra), and False Negative (FN, not found in recovery, but it exists on a real system).

File Permissions As shown in Table 1, we are able to fully replicate most major directories within the running system from static firmware, including */vendor* and */system*. To be precise, we can recover all of the MAC data within the system by parsing *file_contexts* and applying it to files from the firmware. This is equivalent to a running system performing a `restorecon` operation. For filesystems created during the runtime, such as */dev* and */sys*, we are able to infer potential files by parsing `uevent.rc` files, which would normally be loaded by `ueventd` during the phone's early boot. This file contains glob patterns to match device or sysfs nodes in order to apply a user, group, and file mode. We conservatively instantiate files we see referenced in this file, which leads us to be able to recover many character devices. Unfortunately, these files contain more device nodes than are actually found on running phones, as seen in the "Extra Files" (false positives) rows of Table 1. This is a difficult balance to strike. Without recovering `/dev`, many SELinux contexts have no backing files, which means we cannot fully instantiate the associated file objects, leading to false negatives.

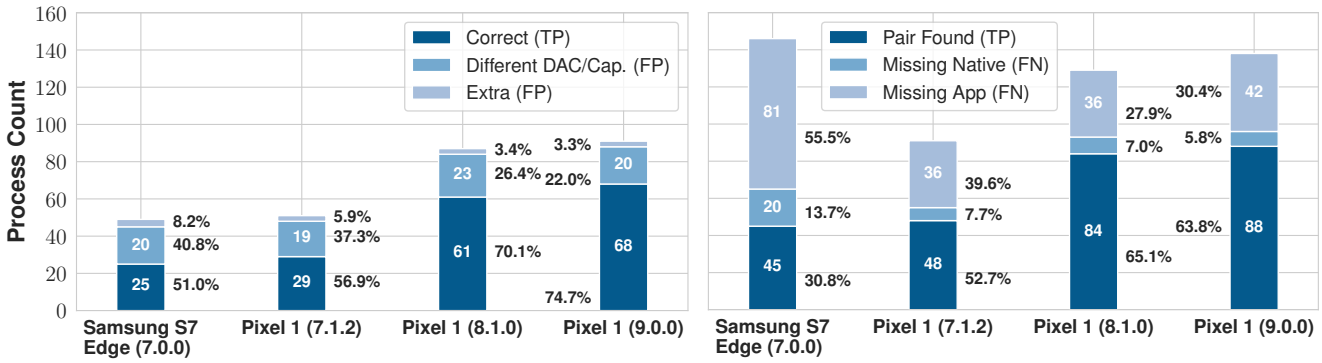
Since we are limited by the content provided by firmware, some directories, namely */data* and */odm*, have a high false-negative rate. These filesystems are only created after the first boot or are vendor-specific. For example, the missing 5,350 files on the Pixel 1 (8.1.0) in */data* are mostly caches for applications. Our attack graph model is primarily focused on system-level directories and files, and we can safely ignore the verbose contents of these directories. **Of the file DAC and MAC data that is possible to recover, our TP positive rate is greater than 98% for all images.**

Process Tree In order to get actionable results from our attack queries, we need as close to an accurate of a picture of the running system as possible. Our process recovery involves a significant amount of information collection and firmware file parsing in order to make the best instantiation possible. Along the way, we employ device and version agnostic algorithms to do so, which is key to support a large range of firmware.

The results of our process recovery are in Figure 4. For the S7 edge in column 1a, we instantiate 49 processes, 25 of which

	Samsung S7 Edge (7.0.0)			Pixel 1 (7.1.2)			Pixel 1 (8.1.0)			Pixel 1 (9.0.0)		
	Path	Count	%Files	Path	Count	%Files	Path	Count	%Files	Path	Count	%Files
Good Files (TP)	*/system*	5,233	93.1%	*/system*	2,301	67.6%	*/system*	2,512	57.4%	*/system*	2,827	60.0%
	/data	115	2.0%	*/vendor*	630	18.5%	*/vendor*	1,264	28.9%	*/vendor*	1,269	26.9%
	/dev	40	0.7%	*/data*	115	3.4%	*/data*	111	2.5%	*/data*	143	3.0%
Different DAC/MAC (FP)	*/dev*	46	0.8%	*/dev*	28	0.8%	*/dev*	46	1.1%	*/dev*	46	1.0%
	/mnt	7	0.1%	*/sbin*	5	0.1%	*/data*	6	0.1%	*/odm*	10	0.2%
	/system	5	0.1%	*/mnt*	2	0.1%	*/sbin*	4	0.1%	*/sbin*	4	0.1%
Extra Files (FP)	*/dev*	73	1.3%	*/dev*	167	4.9%	*/dev*	169	3.9%	*/dev*	169	3.6%
	/system	6	0.1%	*/cache*	4	0.1%	*/data*	10	0.2%	*/cache*	4	0.1%
	/acct	1	0.0%	*/acct*	1	0.0%	*/cache*	4	0.1%	*/acct*	1	0.0%
	Total:	5,621	100%	**Total:**	3,405	100%	**Total:**	5,780	100%	**Total:**	4,709	100%
	DAC/MAC Correct:		98.7%	**DAC/MAC Correct:**		98.6%	**DAC/MAC Correct:**		98.4%	**DAC/MAC Correct:**		98.4%
Missing Files (FN)	*/data*	7,407	75.6%	*/data*	4,425	77.3%	*/data*	5,350	74.2%	*/data*	5,188	73.8%
	/dev	906	9.2%	*/dev*	649	11.3%	*/dev*	856	11.9%	*/dev*	793	11.3%
	/mnt	841	8.6%	*/config*	326	5.7%	*/config*	676	9.4%	*/config*	768	10.9%
	Total:	9,798	100%	**Total:**	8,961	100%	**Total:**	9,821	100%	**Total:**	7,034	100%

Table 1: A comparison of BIGMAC's file recovery and their corresponding MAC and DAC metadata from firmware images versus running devices. Only the top three filesystem paths are shown. The "Good Files" (TP) category shows how many files had 100% identical metadata to the running filesystem. "Different MAC/DAC" (FP) is a listing of the top directories where MAC/DAC data was mismatched. "Extra Files" (FP) shows the directories where BIGMAC recovered files that did not exist at all on the real running system. Finally, "Missing Files" (FN) are the files that were not available from the raw firmware (*/data* contains installed app data, caches, and settings).



(a) The set of processes recovered by BIGMAC.

(b) The set of processes extracted from a running device.

Figure 4: A comparison of BIGMAC's process recovery and versus running devices. In figure (a), "Correct" (TP) shows how many recovered processes had 100% identical metadata (excluding process IDs) as compared to a running phone. "Different DAC/Cap." (TP/FP) is where a process was matched to a running device process, but parts of the DAC/capabilities were different. "Extra" (FP) shows the processes BIGMAC over-approximated and were not running on the real running system during the snapshot. In figure (b), "Pair Found" (TP) shows how many recovered and real processes were able to be matched via executable and SELinux context. "Missing Native" (FN) are the native daemons that were not instantiated. "Missing App" (FN) are the Zygote-children (applications) that are not currently recovered by BIGMAC.

are completely accurate to a running device, 20 are partially accurate, and 4 are extraneous (not running on the real device). Of the missing processes in column 1b, 55.5% are app processes, which BIGMAC does not instantiate at this time as our focus is on native daemons. To do so would require APK extraction and manifest parsing in order to achieve accurate group membership.[6] For the S7 edge, 20 of the processes are native daemons, which were not instantiated for various reasons, such as already instantiated processes created children.[7] For the Pixel 1 9.0.0 we fare the best with 74.7% of processes having a recovered pair and only 9 missing native processes. For the processes we did recover, in all cases, over 50% of the recovered processes completely matched paired real processes. This would not be possible without parsing Android init files

[6] Android conflates some middleware permissions with UNIX groups, such as the `INTERNET` permission with the `inet` GID.

[7] We do not know when or if a process will fork or exec. Doing so would require binary analysis or an emulator.

to extract service definitions.

5.2 Attack Graph Queries

To explore our graph instantiation, we emit the graph nodes, their attributes, and the edges between them to SWI-Prolog for further analysis. We develop an interactive query interface that allows us to ask attack queries about the graph. The result of a query is a list of all possible paths meeting the parameters of the query. The query parameters include: source node, target node, max path length (cutoff), target capabilities, and object type. To make our analysis concrete, we generate attack graphs from a Samsung S8+ 8.0 and LG G7 9.0 firmware.

Layered Path Reduction To demonstrate the path reduction through including DAC checks, we run `query(untrusted_app,vold,4)` with MAC-only filtering (`query_mac`) and with MAC+DAC filtering. As shown in Table 2, we are able to bring down the number of query results by at least an order of magnitude by providing fine-grained query interfaces with multi-layer filtering. We chose the `vold` process because it is a powerful Android platform daemon responsible for mounting and managing disk volumes. It is also part of Android's formal definition of its TCB [19]. On the S8+, `vold` is directly reachable by nearly 100K unique length-4 paths when just considering the MAC policy and only 14K paths after applying DAC. On LG's firmware, there are considerably fewer paths even on the MAC-only query. This is due to the number of unique files available to be written by processes along the path that can be read by `vold`. A diagram of some of the types of paths that were filtered out by the DAC checking, but passed MAC is shown in Figure 5. Files and directories that would be considered writable by the MAC policy alone are discarded when the DAC policy is applied. In this case, the untrusted_app process is not a member of the `media_rw`. Therefore it cannot affect these directories. It is possible that at runtime the app process could gain access to this group or the DAC permissions on the directories could change. BIGMAC currently works off of snapshots of the security policy state. **While applying DAC information increases the query running time, it provides realistic paths results that would be allowed by the MAC and DAC policy on a real system.** MAC-only paths (effectively pure policy analysis) that are shown by previous work have many more false positives, leading to results not accurate enough to draw security conclusions from.

Analysis of a Privilege Escalation To demonstrate how BIGMAC can discover unintended paths that could be used for privilege escalation, we analyze CVE-2018-9488 [30]. As shown in Figure 6, this flaw let the Zygote process compromise the `crash_dump` binary due to its control over the mount namespace. From here, `crash_dump` has the ability to read and write the memory of `vold` using the ptrace syscall (which was allowed by the MAC policy). We use BIGMAC on the Google 8.1.0 image (before it was patched) to discover this escalation path by with `query(process:zygote,process:vold,4)`. This query returns over 700 paths, all of which involve files, except for one, which involves a domain transition to `crash_dump`.

Was Zygote the only daemon capable of compromising `vold` through the `crash_dump` binary? It was not, as `query(_,transition:crash_dump,1,CAP_SYS_ADMIN)` finds 24 other daemons that, if compromised, could achieve the same escalation. A notable daemon is `installd`, which handles installing untrusted APKs. `installd` is responsible for parsing the complicated APK file format and a exploitable vulnerability via this path could have quickly escalated into a full system compromise. We believe this highlights that the lack of granularity of `CAP_SYS_ADMIN` leads to an ineffective security policy. **Any process with this capability can get arbitrary code execution in any other domain it can transition.** We consider this to be a weakness in the capability security model, and more effort needs to be made to limit the number of processes with this. This analysis shows that BIGMAC could help policy writers determine the impact of a policy misconfiguration while taking into account Linux

Query (S8+)	# Paths	Time (s)
`query_mac_only(ua,vold,4)`	99,448	24.74
`query(ua,vold,4)`	14,417	443.53
Query (G7)	# Paths	Time (s)
`query_mac_only(ua,vold,4)`	7,155	3.03
`query(ua,vold,4)`	1,065	55.38

Table 2: The results of the layered query filtering performed for the `untrusted_app` (abbreviated to `ua`) to `vold` attack surface with a cutoff of four.

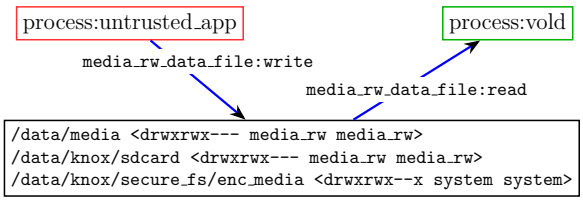


Figure 5: Three paths that were discarded due to DAC filtering on the S8+ image.

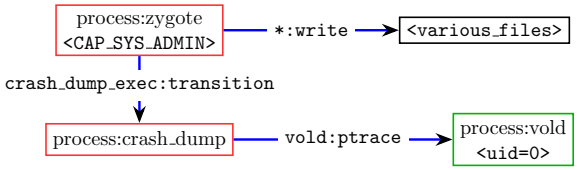


Figure 6: A diagram of the escalation path taken by CVE-2018-9488 to compromise `vold`.

capabilities. This finding is not obvious when analyzing the MAC policy in isolation: Linux capabilities are a crucial part of the overall security model.

Process Strength As an attacker with control over a process and looking to privilege escalate, we would like to see all the possible avenues to write data to objects that can affect other processes. We can easily perform this query for any process by fixing the source node and having a wildcard for the target node. For example: `query(untrusted_app,_,1)` would find all of the objects that can be directly influenced by a process with the untrusted_app label. This query is useful as it takes into account the MAC+DAC policy to find actual writes. With this information, we can identify labels that are *too* strong and should be further compartmentalized. The top three strongest processes are shown in Table 3 for our three firmwares.

For the S8+ and G7 firmwares, `init` and `system_server` are ranked first and second. `init`'s IPC edges consist of property service writes and transitions to new domains. The exception is that it has a socket open to `logd` and `vold`. The rest of its writes are to critical system files, making it file focused. `system_server` is heavily focused on IPC through binder and is also home to hundreds of services. This makes it a large target for attackers considering its key role in mediating application IPC. We argue that a single SELinux label with this much cross-domain write-strength is a risk to the whole system's integrity. For example, while `system_server` does not run as root, many services implicitly trust IPC from it. Using its binder connection to `vold`, it could request any mounting operation. Worse, it has the capability of `CAP_SYS_MODULE` allowing it to load arbitrary kernel modules. **Based upon this finding, we believe `system_server` *must* be refactored into smaller services.** As `system_server` is able to load kernel modules, we argue that it is actually in the system's TCB, yet it is too monolithic to be trustworthy. For the `lpm` process on the S8+ image, it is Samsung's charging daemon. All writable objects are focused on USB and SysFS power management. `hal_usb_default` on the LG G7 is a similar story. It only talks to `system_server` and `hwservice_manager`. All other writes are to USB files and SysFS. This analysis demonstrates that BIGMAC can be used to assist in identifying and triaging over-privileged processes, leading to improved and more modular polices at the MAC, DAC, and CAP layer.

Process Attack Surface Our *query_mac_dac* query interface implements a useful way to study the attack surface of a given process node. We define all possible paths for IPC as an attack surface of a given process. By ignoring the starting node (wildcard) and specifying the target node, we are able to find all the paths leading to the target node. In this case, we focus on `system_server` to further demonstrate how monolithic it is. By limiting the cut-off to be 1, we get all the objects (IPC & files) that the target process can read from. Cut-off 2 finds all possible paths to the target, including the ones found by cut-off 1. This finds all the *writers* of those objects (if any) and eliminates all read-only objects. As a result, the subtraction of these two query response provides us all possible IPC for a target, as shown in Table 4. With these paths, we can look at the shared object between the writer and reader. This is our set of unique IPC objects. With the list of paths, we can perform an "IPC diff" to determine the OEM-specific IPC objects and paths. These are likely to be less audited than upstream AOSP IPC. With this IPC diff, we now know the OEM-specific IPC and can further filter out AOSP IPC paths. For this analysis, we are interested in which IPC paths are writable from an `untrusted_app` (UA). Using a Linux basic text processing and manual inspection, we identified some suspect paths.

Firmware	Process	# Writable	# IPC
S8+	init	2,066	296
	system_server	1,398	458
	lpm	634	8
G7	init	1,233	418
	system_server	573	368
	hal_usb_default	508	19

Table 3: A process strength query where we find all the writable adjacent objects to each process.

Query (S8+)	# Paths	# IPC	Uniq. IPC
`query(_,system_server,1)`	9,853	–	–
`query(_,system_server,2)`	12,681	2,814	716
Query (G7)	# Paths	# IPC	Uniq. IPC
`query(_,system_server,1)`	11,844	–	–
`query(_,system_server,2)`	13,759	1,875	564

Table 4: A combination of BIGMAC queries to find all unique IPC paths and IPC objects in `system_server`.

LG has 11 UA-reachable IPCs. Of those, **we discovered that a UA is allowed to connect to an LG-specific kernel monitoring service as shown in Figure 7**. The MAC nor DAC policy forbid this. This service allows applications to receive information, such as integrity checks, from the running kernel. Further investigation into the service's Binder proxy interface shows that an additional system-app only middleware check is performed at each proxied call. If any of the proxied calls were missing this middleware check, the service would be accessible from any application. Changing the MAC policy to only allow the `system_app` type instead of `untrusted_app` would increase the defense in-depth for this service, in case it exposes a vulnerability in the future. We disclosed this finding to LG, but it is unclear if they will implement the fix as they are deprecating the service going forward.

On the Samsung S8+ image, we discover 58 UA reachable, OEM-unique IPCs. This demonstrates the vast amount of vendor customizations that are made to an already large `system_server`.

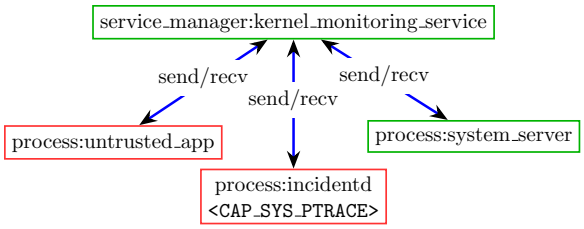


Figure 7: A diagram of the suspicious IPC path discovered on the LG firmware.

Capability Search Our *query_mac_dac_cap* query interface provides a way to find an attack path achieving a certain capability in the ending process node within the path. Some capabilities are more dangerous than others, such as `CAP_DAC_OVERRIDE` and `CAP_SYS_ADMIN`. Possessing one or more of these capabilities increases the strength of an attacker and improves their odds at finding additional escalation paths. In our case, we look at the `untrusted_app` on the S8+ and LG G7 to see how they could achieve additional capabilities in one hop. We limit the cut-off to be 2, focusing on other processes which can directly communicate with the app (e.g., via binder).

We run two queries on both images: `query(untrusted_app,_,2,CAP_SYS_MODULE)` and `query(untrusted_app,_,2,CAP_DAC_OVERRIDE)`. For the LG G7, an app can achieve DAC override via binder by compromising `netd` or `zygote` (both are running as root with all capabilities). For `SYS_MODULE`, it would have to target `system_server` via binder. Letting `untrusted_app` speak to root-level processes via IPC is concerning and indicates an SELinux over-permission. For the Samsung S8+, the paths found are the same, except for three OEM-specific paths to the `hal_iop_default`, `hal_perf_default`, `healthd` processes. All three are running as root with all capabilities. Additionally, in the default AOSP SELinux policy, none are allowed to talk to the `untrusted_app` domain via binder requests. As such, we believe that these direct paths should be eliminated and instead rely on a more trusted proxy for interaction.

External Attack Surface Our *query_mac_dac_cap_ext* interface enables studying external attack surfaces, such as USB, Bluetooth, and modem. By wildcarding the starting node and fixing the target node as a process, we use this query interface to understand the reachability to a certain process from external connections. Fixing the cut-off to be 1 and ignoring the capability option within a query, we find all direct connections between a process and external sources. In our case, we choose the AT distributor process in Samsung S8+ as the target.

As shown in Table 5, the AT distributor process has direct connections with USB, Bluetooth, and Modem. There are 25, 7, and 31 unique device and sysfs files respectively that the process can access and

Query	# Paths
`query(_,at_distributor,1,_,usb)`	29
`query(_,at_distributor,1,_,bluetooth)`	7
`query(_,at_distributor,1,_,modem)`	31
`query(_,at_distributor,1,_,nfc)`	0

Table 5: Different queries on different external attack surface for the at_distributor process.

connect to external devices, including `/dev/mtp/usb`, `/dev/block/platform/soc/7464900.sdhci`, `/dev/mdm`, etc. The AT distributor is mainly responsible for distributing AT commands to different native daemons and applications. Our query demonstrates that while no AT commands likely flow from NFC, USB, Bluetooth, and the modem are areas for further investigation. The large number of paths from the USB and modem demonstrate that native daemons, like the AT distributor, may be prime targets for external exploitation from peripherals.

6 Discussion

6.1 BigMAC for OEMs, Policy Writers, Auditors, and App Developers

BIGMAC, while primarily focused on firmware, will also work from rooted developer devices. An OEM could leverage this by integrating it into their build pipeline for debugging policy misconfigurations on actual devices. It would act as a last-mile check on the "actualized" security policy for the system. For example, imagine there is a rule stating that untrusted applications can never taint a root level process directly without a sanitization step (such as through a trusted process). A static BIGMAC query of the form `query(untrusted_app,_,uid=0,2)` could be used to find all short paths where an app can write to a file that can then be read by a root process. This file can be a regular file, a socket, or service. This list of paths could be compared against a whitelist or cause a failure outright if no paths are to be allowed. By having this query execute after a build, policy misconfigurations could be caught before release, even if their causes were not immediately obvious at the individual MAC/DAC/CAP policy layers. Effectively, BIGMAC could be come a part of Android's compatibility test suite (CTS) which performs many run-time checks already.

Beyond the build pipeline, BIGMAC is extensible and able to give insight into the many policy layers. For example, consider an OEM that has heard about a zero day vulnerability (no public CVE yet) being used in the wild but only knows some of the details, such as "it's triggered from an app" and "it affects system_server". Using BIGMAC, they would be able to query all of the attack paths from an application to this daemon and use it to narrow down on the potential attack

avenue: `query(untrusted_app,system_server,2)`. They would do this by triaging individual paths found. The alternative without BIGMAC would mean manually auditing all of the security policies one by one without a joint perspective.

Policy writers would be able to use BIGMAC to debug permission violations at the MAC+DAC+CAP layers. Additionally, they would be able to use it to determine the attack surface of a process to further focus their fuzzing and hardening efforts. Also, when analyzing CVEs, BIGMAC would aid in finding more semantically similar violations given a query pattern. Security researchers would use BIGMAC to understand how processes interact via IPC and files to narrow down where to look for vulnerabilities.

Auditors could use BIGMAC's firmware extraction capabilities to easily comprehend the running system's security policy (or if they have a physical device with root, extract it dynamically). Previously, auditors would have to manually extract the firmware, decompile the SELinux policy, likely use grep or similar tools to find the relevant types for the process, then have to keep in mind the DAC and CAP policies simultaneously. This burdens the auditor, especially given the large semantic gaps between each policy, and can potentially lead to errors given the amount of manual analysis required. With BIGMAC, their analysis of system daemons will be greatly sped up. Instead of having to reason about what objects these daemons can affect, BIGMAC can print out a report for individual daemons or the links between them automatically. The fully-instantiated graph of BIGMAC boils down the policies to what a running kernel would see instead of a policy writer.

Although BIGMAC is mainly designed for auditors and OEMs, app developers could use BIGMAC to determine why they are getting permission denied errors, assess their application's data and its potential for accidental exposure to other apps via DAC (e.g., when writing files to the SDCard), to aid in porting libraries that rely on file system assumptions (such as permissions) which do not hold on Android.

6.2 Limitations

With only static information available, BIGMAC is limited by the security policy evident in the metadata and configuration files. In reality, a large portion of Android's security model, especially when it comes to IPC communication, relies on a significant amount of access control checks at API boundaries. These checks are a crucial part of middleware permissions and BIGMAC does not not recover a database of these. Previous work has extensively covered middleware permissions [1, 2, 6, 10, 12, 14, 15, 20, 27, 28, 36] and as such we instead focus on developing a fine-grained model of Linux-based security policy. Loss of precision arises from the lack of insight into process behavior. As such, we infer potential IPC objects that exist at runtime, but it is possible that these endpoints are unused or managed by access control differently. Processes are able to create, remove, and change the security state of files at runtime, in addition to creating new instances of themselves at will. These dynamic changes are difficult to capture statically and we avoid making over-permissive assumptions. For example, we may know that a process spawns from init with certain credentials, but it is unclear if it will drop these privileges dynamically. This was the primary source of process DAC/CAP mismatches found during the ground truth evaluation. Processes will start as root temporarily, then `setuid` to a lower privileged Android ID. A potential solution to this issue is to perform binary static analysis using a tool with a powerful Intermediate Language (IL) such as Binary Ninja. [8] Using this IL, we could determine all the cross-references to privilege change functions like `setuid` and attempt to recover the arguments to them. This modification is still not free from the errors inherent with pure static analysis: the dead code problem.

BIGMAC operates on snapshots of a system's access control state. It is possible that a process could modify the DAC state of files under its ownership at runtime. Because BIGMAC relies on static analysis, there is a chance that runtime DAC changes could be missed. However, because of our ability to arbitrarily model DAC permissions, we can provide a conservative worst-case scenario in the case that the owning subjects for processes protected by DAC modify their permissions. Our focus in this paper is on existing permissions found through the static image and while our tool supports these capabilities, we defer further worst-case analysis of runtime behavior to future work.

The greedy approach taken by BIGMAC to creating files from *uevent* configuration files in Section 4.2 can lead to extra files that do not appear on our ground-truth equivalent. The alternative is missing a significant portion of the in-use file contexts. When supporting this feature, we increased the recovery of unique file contexts from 51% to 88% on the Pixel 8.1.0. Given the importance of these device nodes for overall system security, this is a particularly notable gain.

Another aspect of Android's system security model involves the use of `allowxperm` rules to perform IOCTL filtering. Our model does not consider the kernel as part of the attack surface, but with nearly 40% of the current Android exploits targeting the kernel [32], future work should also take these escalation paths into consideration. On the kernel side, new versions of Android use eBPF SECCOMP to create sandbox profiles for untrusted applications. In our model, we only consider Supporting SECCOMP and `allowxperm` would require minor additional engineering work.

7 Related Work

SELinux Policy Analysis Creating SELinux policies is non-trivial; as such, improving policy analysis has been a topic of significant research. One of the earliest tools in this space was Gokyo [25], which originally considered SELinux policies and

[8] https://binary.ninja

subsequently expanded to allow computing of multiple access control spaces [26]. Our work differs in that we consider DAC and Linux capabilities in addition to MAC policies. Hicks et al. released PALMS [23], a Prolog tool for specifying MLS policy in SELinux. We similarly use Prolog for reasoning over policy; while MLS is interesting and a topic of future support for BigMAC, in Android it is primarily enabled in Mobile Device Management (MDM) contexts, which we do not model.

Chen et al. examined the overall protection quality of different SELinux-enabled operating systems by creating VulSAN [5]. VulSAN used logic programming to create attack-graphs to help compare the difficulty of attack scenarios between operating systems. SCIATool integrates different methods of policy analysis together in a framework for querying policies [43] SEGrapher and V3SPA approach the problem of policy analysis visually and tackle many of the issues inherent in drawing and laying out large graphs with many edges [21,29]. Policy visualization gives insight into the relationships within and between policies. V3SPA also provides a visual diffing tool for two policies. Our work does not yet create interactive graphs, but we employ attribute clustering to avoid edge-explosion from attribute expansion. Eatman et al. surveyed existing policy analysis frameworks and determined that a front-end, formally specified, policy language should be created for specifying SELinux rules [11]. Existing tools have mostly employed variants of Prolog or other custom implementations to perform analysis. Effectively, researchers are already abstracting away from SELinux to perform useful analysis.

Mobile Security Policy Analysis The complexities of creating and maintaining SELinux policies has carried over to Android since its introduction of SEAndroid in 2013 [37]. Reshetova et al. performed one of the initial analyses of real SEAndroid policies from actual devices [34]. To achieve this, they developed a tool called SEAndroid Analytics Library (SEAL) to help collect SELinux contexts from running devices for further analysis. Using SEAL requires access to a rooted or developer device, which limits the ability to scale up the analysis to many vendors or devices. Chen et al. analyzed over ten SEAndroid policies for different classes of misconfiguration [4]. They discovered that the combination of DAC and MAC policies could interact in unintended ways, leading to compositional over-privilege, amongst other errors. This work also required running devices running debug builds in order to extract security policies and maxed out at Android 6.0, which was widely available at that time. To help policy writers create new allow rules, EASEAndroid [41] developed a machine learning model to help classify Access Vector Cache (AVC) denials from collected SEAndroid logs as important or spurious. Another approach has been to analyze the source code of SEAndroid policies for common misconfigurations through the tools SELint and SPOKE [33, 40]. Our approach does not assume access to SEAndroid policy source code, as most policies are distributed in a compiled form and considered proprietary. A historical analysis of the SEAndroid policy evolution focusing on policy complexity [24] is also proposed based on Git commit history. There has also been previous work on Android privilege escalations and how to mitigate them using a MAC approach [3], but this was before SEAndroid was implemented and enforced on the platform. Similarly, prior to the development of SEAndroid, Muthukumaran et al. demonstrated a technique for enforcing CW-Lite, a lighter-weight approach to Clark-Wilson integrity [35], on OpenMoko mobile devices running with an SELinux-enhanced kernel [31].

Most mobile policy analysis has been focused on Android, but SandScout [9] and iOracle [8] focus on analyzing Apple iOS Sandbox profiles. SandScout decompiles these profiles from iOS firmware into a Prolog representation and create queries to discover new vulnerabilities used by previously released jailbreaks. iOracle expands upon this work by analyzing the whole iOS security model, including capabilities and UNIX permissions. Their work requires dynamic analysis for data collection, while BIGMAC relies solely on static analysis with domain knowledge. Additionally, BIGMAC incorporates Linux capabilities, DAC, and MAC together in a single graph for the whole-system, which previous iOS policy work does not replicate.

8 Conclusion

We present BIGMAC, a new security policy analysis framework for Android. BIGMAC can rebuild the running system state from firmware images without the need for physical devices. We build an attack-graph in BIGMAC and a logic-based query engine combining MAC, DAC, capabilities, and external attack surface analysis. We can thus find attack paths between processes and from external sources to audit Android security policies.

Acknowledgments

This work was partially supported by the National Science Foundation CNS-1815883, the Semiconductor Research Corporation (SRC), and AFOSR.

References

[1] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 217–228. ACM, 2012.

[2] M. Backes, S. Bugiel, and S. Gerling. Scippa: System-centric IPC Provenance on Android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 36–45, New York, NY, USA, 2014. ACM.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, volume 17, page 19, 2012.

[4] H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the*

33rd Annual Computer Security Applications Conference, ACSAC 2017, pages 553–565, New York, NY, USA, 2017. ACM.

[5] H. Chen, N. Li, and Z. Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS*, pages 11–16, 2009.

[6] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[7] Chris PeBenito. SETools: SELinux Policy Analysis Tools v4, Mar. 2018. https://github.com/SELinuxProject/setools.

[8] L. Deshotels, R. Deaconescu, C. Carabas, I. Manda, W. Enck, M. Chiroiu, N. Li, and A.-R. Sadeghi. iOracle: Automated Evaluation of Access Control Policies in iOS. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 117–131, New York, NY, USA, 2018. ACM.

[9] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi. SandScout: Automatic Detection of Flaws in iOS Sandbox Profiles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 704–716, New York, NY, USA, 2016. ACM.

[10] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, volume 31, page 3, 2011.

[11] A. Eaman, B. Sistany, and A. Felty. Review of Existing Analysis Tools for SELinux Security Policies: Challenges and a Proposed Solution. In E. Aïmeur, U. Ruhi, and M. Weiss, editors, *E-Technologies: Embracing the Internet of Things*, Lecture Notes in Business Information Processing, pages 116–135. Springer International Publishing, 2017.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.*, 32(2):5:1–5:29, June 2014.

[13] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Secur. Privacy*, 7(1):50–57, 2009.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[15] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *USENIX Security Symposium*, page 88, 2011.

[16] Gartner. Market Share Alert: Preliminary, Mobile Phones, Worldwide, 2Q18. https://www.gartner.com/doc/3881811/market-share-alert-preliminary-mobile, July 2018.

[17] Google. Android Init Language, Feb. 2018. https://android.googlesource.com/platform/system/core/+/master/init/README.md.

[18] Google. SELinux for Android 8.0, Feb. 2018. https://source.android.com/security/selinux/images/SELinux_Treble.pdf.

[19] Google. Security Updates and Resources, Aug. 2019. https://source.android.com/security/overview/updates-resources.

[20] S. A. Gorski, B. Andow, A. Nadkarni, S. Manandhar, W. Enck, E. Bodden, and A. 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 '19, pages 25–36, New York, NY, USA, 2019. ACM. Richardson, Texas, USA.

[21] R. Gove. V3SPA: A visual analysis, exploration, and diffing tool for SELinux and SEAndroid security policies. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, Oct. 2016.

[22] S. E. Hallyn and A. G. Morgan. Linux capabilities: making them work. 2008.

[23] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. *ACM Trans. Inf. Syst. Secur.*, 13(3):26:1–26:31, July 2010.

[24] B. Im, A. Chen, and D. S. Wallach. An historical analysis of the seandroid policy evolution. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 629–640. ACM, 2018.

[25] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security Symposium*, 2003.

[26] T. Jaeger, R. Sailer, and X. Zhang. Policy management using access control spaces. *ACM Trans. Info. Sys. Sec.*, 6(3):327–364, Aug. 2003.

[27] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-Time Enforcement of Information-Flow Properties on Android. In J. Crampton, S. Jajodia, and K. Mayes, editors, *Computer Security – ESORICS 2013*, Lecture Notes in Computer Science, pages 775–792. Springer Berlin Heidelberg, 2013.

[28] T. Markmann, D. Gessner, and D. Westhoff. QuantDroid: Quantitative approach towards mitigating privilege escalation on Android. In *2013 IEEE International Conference on Communications (ICC)*, pages 2144–2149, June 2013.

[29] S. Marouf and M. Shehab. SEGrapher: Visualization-based SELinux policy analysis. In *2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*, pages 1–8, Oct. 2011.

[30] MITRE. CVE-2018-9488. https://nvd.nist.gov/vuln/detail/CVE-2018-9488, 2018.

[31] D. Muthukumaran, J. Schiffman, M. Hassan, A. Sawani, V. Rao, and T. Jaeger. Protecting the integrity of trusted applications in mobile phone systems. *Security and Communication Networks*, 4(6):633–650, 2011.

[32] Nick Kralevich. Honey, I Shrunk the Attack Surface - Adventures in Android Security Hardening. Black Hat, July 2017.

[33] E. Reshetova, F. Bonazzi, and N. Asokan. SELint: an SEAndroid policy analysis tool. *arXiv:1608.02339 [cs]*, pages 47–58, 2017. arXiv: 1608.02339.

[34] E. Reshetova, F. Bonazzi, T. Nyman, R. Borgaonkar, and N. Asokan. Characterizing SEAndroid Policies in the Wild. *arXiv:1510.05497 [cs]*, Oct. 2015. arXiv: 1510.05497.

[35] U. Shankar, T. Jaeger, and R. Sailer. Automted Information-Flow Integrity Verification for Security-Critical Applications. In *ISOC Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[36] Y. Shao, J. Ott, Y. J. Jia, Z. Qian, and Z. M. Mao. The Misuse of Android Unix Domain Sockets and Security Implications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 80–91, New York, NY, USA, 2016. ACM.

[37] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, pages 20–38, 2013.

[38] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[39] D. Tian, G. Hernandez, J. Choi, V. Frost, C. Ruales, K. Butler, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, and M. Grace. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 351–366, Washington, D.C., 2018. USENIX Association.

[40] R. Wang, A. M. Azab, W. Enck, N. Li, P. Ning, X. Chen, W. Shen, and Y. Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 612–624, New York, NY, USA, 2017. ACM.

[41] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. Easeandroid: Automatic policy analysis and refinement for security enhanced android via large-scale semi-supervised learning. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 351–366, Washington, D.C., 2015. USENIX Association.

[42] E. Yunis, R. Yokota, and A. Ahmadia. Scalable force directed graph layout algorithms using fast multipole methods. In *2012 11th International Symposium on Parallel and Distributed Computing*, pages 180–187. IEEE, 2012.

[43] G. Zhai, T. Guo, and J. Huang. SCIATool: A Tool for Analyzing SELinux Policies Based on Access Control Spaces, Information Flows and CPNs. In M. Yung, L. Zhu, and Y. Yang, editors, *Trusted Systems*, Lecture Notes in Computer Science, pages 294–309. Springer International Publishing, 2015.

A Appendix

A.1 Implementation Details

Graph Building Once we have a complete policy database saved, we may reload it and begin processing it how we like. In our case, we instantiate attack-graphs, therefore we employ NetworkX for constructing and traversing graphs. We load the saved SEPolicy image, covert it to nodes and labeled edges and also save this for future processing. We also employ Graphviz for visualizing our subject, process, and attack-graphs using the `sfdp` [42] program, which has no trouble laying out large graphs.

For our dataflow graph, we needed to handle a few quirks in our object model. One of these was splitting of character device files into separate read and write objects. Without doing this, processes P_1 and P_2 that can write and read from O_1, $P_1 \longleftrightarrow O_1 \longleftrightarrow P_2$, have an transitive dataflow to each other. This is not correct, therefore we split the object into a read-only and write-only versions: $P_1 \longrightarrow O_{1W} \longleftarrow P_2$, $P_1 \longleftarrow O_{1R} \longrightarrow P_2$. These new nodes are leaf nodes, preventing this unintended flow.

A.2 Tables

USB Attack Surface With our instantiated graph, we tag certain file objects, specifically device nodes relating to USB input and output. We then query our graph searching for all paths from these USB sources to a single or multiple processes. This allows us to get insight into the processes that can be directly affected by USB data. In our case, we chose the `/dev/ttyUSB0` device file which connects with external USB devices. By fixing the cut-off to be 1, we are able to find all processes that are able to directly read from this device file.

As shown in Table 6, a USB connection can directly reach 25 unique processes, such as `DMM-daemon`, `adbd`, `remotedisplay`, etc. Among these 25 processes, 22 have the `CAP_DAC_OVERRIDE` capability, and 22 have the `CAP_SYS_ADMIN` capability. The data demonstrates that USB as an external attack surface interacts with many privileged processes. While some of the daemons have clear reasons to talk with USB, e.g., `adbd`, others might not be obvious, e.g., `sensors`. To reduce the USB attack surface, the first question we need to ask is if it is reasonable to expose a native process to USB connections. We also need to constrain the capabilities within the processes exposed to external USB connections.

Type Reduction With our fully instantiated graph, we are able to distinguish between active and inactive subject domains and file contexts. System-wide security policies are required to describe every possible type transition and access vector by nature. In reality, many of these edges will never be taken on a real system due to its specific configuration. BIGMAC, by design is able to narrow in on the most important domains by finding mappings between them and the underlying filesystem (backing files). If an example of a file is not found, we can prune this type completely. For processes, if a backing executable is not found then, this process is abstract and is not able to be executed on a real system. This effectively prunes the set of types to be considered along with their corresponding access vector rules. In Table 7, we see the effect that BIGMAC has in type reduction from a raw SEPolicy and show that we are able to reduce the set of subject types considered by at a minimum 12% and at a maximum 52%. The reduction varies by OEM and Android version. Some OEMs share a single SEPolicy for all devices and label only the necessary types during the image building, while others will have device-specific policies that are more tailored and compact. For file contexts, not all paths that can be assigned a context will exist all at once or at all, leading to a significant reduction in file types to instantiate.
Pure policy analysis does not consider the true instantiation of a system, which may lead to discovering problems that do not affect real systems due to dead-types.

Query	# Paths
`query(/dev/ttyUSB0,_,1)`	25
`query(/dev/ttyUSB0,_,1,CAP_DAC_OVERRIDE)`	22
`query(/dev/ttyUSB0,_,1,CAP_SYS_ADMIN)`	22

Table 6: Different queries on the starting node `/dev/ttyUSB0`, which can be reached via external USB connections. The cutoff 1 specifies the direction connection between processes and the device file. The number of paths also represents the unique number of processes.

		SEPolicy					Instantiation	
Model	Version	Types	Attributes	Allow	Domains	File Types	Domain Reduction%	File Context Reduction%
Google Pixel	7.1.2	733	29	7,186	114	278 / 477	11.4%	41.7%
	8.0.0	1,093	601	17,520	173	404 / 683	17.3%	40.8%
	9.0.0	1,337	125	18,929	210	401 / 656	15.9%	38.9%
Samsung S8+	7.0.0	2,222	75	16,907	349	497 / 1,646	51.6%	69.8%
	8.0.0	2,409	764	30,482	348	643 / 1,858	48.3%	65.4%

Table 7: A table showing how BIGMAC instantiates from a raw firmware image with an abstract SEPolicy to a concrete set of services and files. Our approach eliminates unused domains by ensuring that all domains have an associated executable. If none is found, we consider it abstract and discard it. This reduces the number of types, domains, and allow rules that need to be considered, yielding a more targetted analysis. The same is true for filesystem contexts. Through parsing Android init.rc files, we are able to recover the set of possible services and not consider transient (oneshot) services. This is far more focused than pure policy analysis as the entire policy may not be used on a specific device model or running system.

Read Access Vectors	`*:read`, `*:ioctl`, `*:unix_read`, `*:search`, `*:recv`, `*:receive`, `*:recv_msg`, `*:recvfrom`, `*:rawip_recv` , `*:tcp_recv`, `*:dccp_recv`, `*:udp_recv`, `*:nlmsg_read`, `*:nlmsg_readpriv`, `binder:call`, `service_manager:{list,find}`
Write Access Vectors	`*:write`, `*:append`, `*:ioctl`, `*:add_name`, `*:unix_write`, `*:enqueue`, `*:send`, `*:send_msg`, `*:sendto`, `*:rawip_send`, `*:tcp_send`, `*:dccp_send`, `*:udp_send`, `*:nlmsg_write`, `binder:call`, `service_manager:{add,find}`, `process:transition`, `process:ptrace`,

Table 8: A mapping of `class:vector` tuples into a read or write-data flow edge type.

Vendor	Model	Version	Build ID	URL
Google	Pixel 1	7.1.2	NJH47	https://dl.google.com/dl/android/aosp/sailfish-ota-njh47f-b1b5d050.zip
Google	Pixel 1	8.1.0	OPM1.171019.011	https://dl.google.com/dl/android/aosp/sailfish-ota-opm1.171019.011-5dca05ea.zip
Google	Pixel 1	9.0.0	PPR2.181005.003	https://dl.google.com/dl/android/aosp/sailfish-ota-ppr2.181005.003-db23e6d5.zip
Samsung	S7 Edge (SM-G935F)	7.0.0	NRD90M	https://androidfilehost.com/?fid=529152257862696441
Samsung	S8+ (SM-G955U)	8.0.0	R16NW	https://www.sammobile.com/samsung/galaxy-s8-plus/firmware/SM-G955U/TMB/download/G955USQS3CRE2/219483/
LG	G7 ThinQ (G710EM)	9.0.0	PKQ1.181105.001	https://lg-firmwares.com/downloads-file/19702/G710EM20b_00_OPEN_EU_OP_0508.kdz

Table 9: Firmware used in our evaluation metadata and download links.