SAUSAGE:
Security Analysis of Unix domain Socket usAGE in Android

Mounir Elgharabawy
Concordia University
Montreal, Quebec, Canada
m_elghar@encs.concordia.ca

Blas Kojusner
University of Florida
Gainesville, Florida, USA
bkojusner@ufl.edu

Mohammad Mannan
Concordia University
Montreal, Quebec, Canada
m.mannan@concordia.ca

Kevin R. B. Butler
University of Florida
Gainesville, Florida, USA
butler@ufl.edu

Byron Williams
University of Florida
Gainesville, Florida, USA
byron@cise.ufl.edu

Amr Youssef
Concordia University
Montreal, Quebec, Canada
amr.youssef@concordia.ca

Abstract—The Android operating system is currently the most popular mobile operating system in the world. Android is based on Linux and therefore inherits its features including its Inter-Process Communication (IPC) mechanisms. These mechanisms are used by processes to communicate with one another and are extensively used in Android. While Android-specific IPC mechanisms have been studied extensively, Unix domain sockets have not been examined comprehensively, despite playing a crucial role in the IPC of highly privileged system daemons. In this paper, we propose SAUSAGE, an efficient novel static analysis framework to study the security properties of these sockets. SAUSAGE considers access control policies implemented in the Android security model, as well as authentication checks implemented by the daemon binaries. It is a fully static analysis framework, specifically designed to analyze Unix domain socket usage in Android system daemons, at scale. We use this framework to analyze 200 Android images across eight popular smartphone vendors spanning Android versions 7-9. As a result, we uncover multiple access control misconfigurations and insecure authentication checks. Our notable findings include a permission bypass in highly privileged Qualcomm system daemons and an unprotected socket that allows an untrusted app to set the scheduling priority of other processes running on the system, despite the implementation of mandatory SELinux policies. Ultimately, the results of our analysis are worrisome; all vendors except the Android Open Source Project (AOSP) have access control issues, allowing an untrusted app to communicate to highly privileged daemons through Unix domain sockets introduced by hardware manufacturer or vendor customization.

1. Introduction

One of the fundamental features any modern operating system provides is Inter-Process Communication (IPC), used extensively by applications to implement inter-functionality between their components. The widely-used Android OS provides a variety of its own IPC mechanisms (e.g., Binder, Intents, Messenger), while also inheriting the traditional IPC mechanisms available in a Linux en-

vironment, in particular, Unix domain sockets. As with any IPC mechanism, privileged processes communicating over these sockets are potentially vulnerable to confused deputy attacks, if they are inadequately protected by the access control policy. In that case, a malicious unprivileged process can control a privileged process to overwrite critical files [29], execute shell commands [6], or gather screenshots and sensitive system logs [27], among other things.

Multiple vulnerabilities and exploits are due to the misuse of Unix domain sockets. For example, CVE-2011-3918 [5] describes an unprotected socket to the Zygote process, which can be leveraged to perform a denial of service attack on a device running AOSP Android 4.0.3. Other examples include the "HTC WeakSauce" exploit, which uses a socket connection to the privileged *dmagent* system daemon to achieve privilege escalation to root [29], and CVE-2013-4777 and CVE-2013-5933 [6], [7], privilege escalation vulnerabilities affecting Motorola devices, due to an unprotected socket to the *init* process. More recently, an information disclosure vulnerability was discovered on Huawei phones, allowing attackers to gather screenshots and kernel and system logs [27]. The vulnerability is exploited by sending commands via an exposed socket to a vendor-customized version of the *debuggerd* daemon. These vulnerabilities demonstrate that unprotected sockets can degrade the security of the system whether they originate in stock (i.e., AOSP), or vendor-customized Android.

However, previous research has primarily focused on Android-specific IPCs, such as Binder [19], [23], [28], [30], [32] and Intents [24], [39], with comparatively little evaluation of traditional Linux IPCs such as Unix domain sockets. While Shao et al. [40] examined the misuse of Unix domain sockets in Android, finding that the inadequate protection of these sockets is a common pitfall in Android, their approach fell short for system daemons, which are arguably much more valuable targets for exploitation. This shortcoming is due to the use of dynamic analysis to avoid challenges such as reasoning about the complex interaction of Android access control layers, extracting firmware images from different vendors

with different formats, and statically analyzing system daemon ARM binaries accurately. As a consequence, their analysis requires access to a running, rooted Android device, and thus only covers two vendors across three Android versions. Also, the approach of Shao et al. makes a cross-vendor analysis infeasible at scale, despite being essential to uncover the misuse of sockets introduced by vendor customization. Furthermore, it fails to uncover inactive sockets, which can be created in response to an event or configuration change.

To address this gap, we propose SAUSAGE, a static analysis framework to identify valid socket connections that untrusted apps can establish to system daemons on an Android device, without the need of a running device, enabling large-scale analysis. Given an Android firmware image, our framework analyzes access control policies and performs static binary analysis on daemon binaries to discover socket addresses that an untrusted app can connect to and any authentication checks implemented in the binary. We overcome the challenge of reasoning about Android access control policies by using a version of the BIGMAC [26] SELinux policy analysis tool; we extend its functionality to enable socket creation within the init boot simulation step. Since all Android IPC is governed by MAC (and sometimes by DAC), we believe that our approach may serve as a good base for future work examining the security of Android IPC mechanisms (especially statically). We also implement our own binary analysis component. The SAUSAGE framework extracts the system's SELinux policy, system daemon binaries and init RC files from an Android firmware image. It analyzes the SELinux policy to determine which system daemons an untrusted app can communicate with. By using inter-procedural data-flow analysis, it then detects socket addresses, their access control credentials, and any authentication checks in the system daemon binaries with high accuracy.

We used our framework to analyze 200 Android firmware images, spanning eight different vendors and Android versions 7-9. SAUSAGE fully analyzes a firmware image in around 14 minutes, making it scalable as a cross-vendor Android firmware analysis tool, without requiring vendor-specific devices. The results of our analysis are worrisome; all vendors except AOSP have access control issues that allow an untrusted app to communicate to highly privileged daemons. These include HTC *dmagent* that has been previously exploited in "HTC WeakSauce," and Samsung's Professional Audio service, which allows any app to set its process scheduling priority. We also identify insecure authentication practices used by these daemons, such as checks based on an app's process name, which can be trivially spoofed. Additionally, we demonstrate that our approach can uncover Unix domain sockets that would have been missed by the dynamic analysis approach used by Shao et al. [40]

Contributions.

1) We propose an access control-aware, fully static framework to analyze Unix domain socket usage in Android system daemons, compatible with Android versions 7.0 and above. Using this framework, we conduct an analysis of 200 Android factory images spanning versions 7.0-9.0 across eight different vendors, including prominent players such as Samsung and Xiaomi, which account for over a third of the mobile vendor market share worldwide [15], and others such as Asus, Motorola, HTC, and AOSP, to find system daemon sockets accessible to untrusted apps.[1] We compare our results to the ground truth from three running devices and find that our framework achieves 100% accuracy in detecting socket addresses and their MAC and DAC credentials.

2) We use a novel methodology based on a version of BIGMAC (that we modified at the init boot simulation step) to reason about Android's complex interaction of access control layers through static analysis. We will publish the source of the binary analysis module to the community, as well as the modified version of BIGMAC we use as part of our framework.

3) We find multiple instances of unprotected Unix domain sockets to root processes that could lead to exploits such as HTC WeakSauce [29]. Our approach detects Unix domain sockets that are created under certain conditions and would not have been detected through past approaches relying on dynamic analysis alone.

Notable Findings.

1) We found two highly privileged Qualcomm system daemons, *cnd* and *dpmd*, where a faulty authentication mechanism is used to authenticate the peer, relying only on its process name. However, the process name can be easily spoofed in both cases. Through static analysis, we infer that this allows clients to get/set network settings such as WiFi AP, WiFi P2P, and Default Network settings, by sending the appropriate command over the cnd socket.

2) In 25 Samsung Android 7.0-7.1 images, we found that the daemon apaservice listens over a socket that can be used to request changing the scheduling priority for any process to any priority. This can result in DoS of the Samsung audio subsystem. Additionally, the daemon is vulnerable to buffer overflow, allowing an untrusted app to execute code in the daemon's security context. This vulnerability was assigned CVE-2021-25461.

3) There are multiple instances of overly permissive SELinux policies in seven of the eight vendors we analyzed. These policies allow socket communication between untrusted apps and highly privileged system daemons, weakening the system's overall security posture. Examples of these daemons include *dumpstate* and *rild* in HTC, and *cnd* in most vendors.

4) We discovered multiple instances of vendor customization of AOSP binaries that expose additional unprotected sockets. One example is HTC *rild* where two custom sockets were added that are configured to be accessible to an untrusted app.

---

1. We use the term *untrusted app* to reference third-party applications a user can install.

We have contacted Samsung and Qualcomm as part of our responsible disclosure process with details of the vulnerabilities and proof-of-concepts. We detail this process in Section 7.

Outline. The remainder of the paper is structured as follows: Section 2 provides the necessary background on how Unix domain sockets work and their role in the Android security model. Section 3 details key related work. Section 4 develops the design of SAUSAGE and Section 5 its implementation. In Section 6 we evaluate SAUSAGE against multiple Android firmware images and demonstrate the scalability and impact it can have when discovering accessible sockets. In Section 7 we present case studies on the results obtained with SAUSAGE. We discuss our limitations in Section 8, and finally conclude in Section 9.

2. Background

In this section, we provide background about how Unix domain sockets work and where they fit in the Android security model.

2.1. Android Security Model

The Android security model implements various layers. Apps run in sandboxes defined by the creation of a unique Linux UID for each application at install time. Processes can only communicate with other applications via enforced mandatory access control (MAC). This is a feature implemented in Security-Enhanced Linux (SELinux) through the modified SEAndroid framework. Interested readers can find a comprehensive discussion of the Android security model in [33]. In this section, we explain the key concepts behind it that are relevant to our work.

Discretionary Access Control on Android (DAC). DAC is an access control model that is used by Linux. It is implemented in Android by using a fixed set of user and group IDs for system-related purposes and limiting a range of user IDs for dynamically installed applications. Android limits the number of processes that can run as root, therefore a highly-privileged process would typically run under the system UID, or another UID specific to execute the role intended for the process. This avoids granting more permissions than necessary to a process, which can inherently avoid security issues. Untrusted apps are assigned a unique UID from a specified range of IDs that are available. This prevents third-party applications from having more access than necessary on any other files that are not included in their installation.

SELinux. Security Enhanced Linux (SELinux) provides a framework for enforcing Mandatory Access Control (MAC) on Linux. It was introduced into the Android platform in 2013 through the SEAndroid framework [43]. SELinux contains a set of rules that are based on file labels which contain information such as user, role, type, and level. These rules determine what types and actions a process has access to and are structured to group items together based on their accessibility. In Android, SELinux is not only restricted to access control for files, but it also manages access control for IPC mechanisms, such as Binder and Unix domain sockets. Thus, for processes to

TABLE 1. SECURITY ENFORCEMENT CORRESPONDING TO ANDROID UNIX DOMAIN SOCKET NAMESPACES

Namespace	Security Enforcement	
	SELinux	File Permissions
RESERVED	✓	✓
FILESYSTEM	✓	✓
ABSTRACT	✓	✗

communicate with one another, the communication must be explicitly allowed by the SELinux policy.

SELinux policies are developed by vendors by combining the core AOSP policy with device-specific customization. Policies compare rules that guide the SELinux security engine, including types for file objects and domains for processes. The SELinux policy uses roles to limit the domains that can be accessed and has user identities to specify the roles that users can have. New rules can be added into the policy which is then preprocessed and built into the policy.conf file.

Supplementary Groups. Adding a supplementary group ID to an application will grant it all the privileges of the specified group. The groups for an application are assigned within the manifest file. An example of some supplementary groups would be the Bluetooth group or the Internet group. The permissions of a supplementary group can be enforced at the kernel level or at the Android Framework level depending on the functionality granted to the group [18].

Middleware Permissions. The Android middleware layer contains a reference monitor that mediates inter-component communication [22]. Middleware permissions grant apps access to resources and services that are provided by the Android operating system rather than the Linux kernel.

2.2. Unix Domain Sockets

A Unix domain socket is a communications endpoint for exchanging data between processes on the same host operating system. It can also be referred to as an inter-process communication socket. The main difference between Unix domain sockets and Internet sockets is that a Unix domain socket is an IPC where all communication occurs strictly within the operating system kernel. Internet sockets use an underlying network protocol for communication. Unix domain sockets also have what is called a namespace, or a unique identifier to an object of a certain kind, that is used to label the socket types. There are three types of Unix domain socket namespaces in Android, as can be seen in Table 1.

FILESYSTEM. Sockets with this namespace are associated with a file on the file system and are created by a process that needs them. Once a socket file is created it will be protected by the discretionary access control system, or the DAC, as well as the mandatory access control, or the MAC. Only processes with the proper read and write permissions can communicate with these socket files.

RESERVED. This namespace is introduced in Android and falls under the FILESYSTEM namespace and thus inherits its access control properties. The socket files are created by init and are located under /dev/socket.

The name indicates that these socket files are reserved for system use. The socket file descriptors are made available to their owner service daemon through an environment variable named `ANDROID_SOCKET_<addr>` where `<addr>` is the address of the socket in the RESERVED namespace.

ABSTRACT. These sockets allow a program to bind a Unix domain socket to a name without the name being created in the filesystem. The socket's name begins with a null byte which removes the need to create a filesystem path name for the socket.

There are three prerequisites for a process to be allowed to establish a connection to a FILESYSTEM socket. First, the connecting process must be allowed to communicate to the server process through the Unix domain socket IPC by SELinux. Second, the connecting process must be allowed to write to the socket file, based on its file context in SELinux. Third, the connecting process must have the appropriate UID or GID to write to the socket file, depending on the socket file's DAC file permissions. On the other hand, only the first prerequisite is needed in the case of ABSTRACT sockets since file-based access control policies are not applicable to them. As a result, ABSTRACT sockets are the least secure of the three namespaces. Furthermore, Unix domain sockets can only be bound by one process. Filesystem MAC and DAC can restrict the creation of sockets under certain directories to a set of processes, preventing untrusted apps from binding sockets used by system daemons. This does not apply to ABSTRACT sockets, however, allowing a malicious app to DoS the system daemon by occupying an ABSTRACT address if the app manages to bind the socket address before the daemon. Fortunately, daemons started by init are always started before apps, and in most cases, their sockets are bound on initialization and stay bound for the daemon's entire lifecycle.

3. Related Work

Android IPC security has long been the focus of a large body of research. Most of this research, however, centered on either the Binder IPC interface [19], [23], [28], [30], [32], or on Android Intents [24], [39]. Furthermore, the majority of these analyses are concerned with Android application security rather than Android framework security. Iannillo et al. [28] designed *Chizpurfle*, a grey-box fuzzer for system services that discovers vendor-specific system service methods exposed through Binder IPC and runs a fuzzing campaign on the identified methods. However, it heavily relies on analyzing Java reflection by design, so it is not compatible with native system services. Liu et al. [32] overcame this limitation with, FANS, which is capable of finding vulnerabilities in native system services, but is limited to examining the Binder IPC mechanism. While, these works clearly demonstrate their effectiveness finding vulnerabilities in system services, there is no clearly defined threat model that accounts for the multi-layered Android security model. Thus, some of the reported vulnerabilities from these works may require chaining with other privilege escalation exploits to interact with privileged, but vulnerable, Binder interfaces. Additionally, the chaining may be prevented by commonly deployed access control policies.

Shao et al. [40] conducted the first study of Unix domain socket usage by both Android apps and system daemons. To perform their analysis, they developed SInspector, which identifies Unix domain socket addresses and detects authentication checks. SInspector exclusively utilizes static techniques for apps, allowing for large scale analysis of apps. However, for system daemons, the tool needs to be run on a live, rooted Android system. The reasons for this limitation include the difficulty of unpacking Android factory images from different vendors and the complexity of security enforcement for sockets, which involves the interplay of SEAndroid and DAC file permissions. However, with the advent of new open-source tools such as the Android image unpacking library [44] and BIGMAC [26], it is now feasible to tackle these challenges and develop a fully static large-scale cross-vendor analysis framework for Unix domain socket usage in system daemons. We use these tools to overcome the challenges faced by SInspector that caused it to resort to dynamic analysis for system daemons.

A separate growing body of work examines Android OS security from an access control perspective [26], [31], [37], [46], [47]. Android access control, especially SEAndroid, plays an essential role in securing IPC. All Android IPC mechanisms are protected by the SEAndroid policy and some are protected by DAC. Lee et al. proposed PolyScope [31], a tool to vet Android filesystem access control policies. They define three possible patterns of integrity violations in access control policies and rely on AOSP documentation as well as the integrity walls method [45] to categorize process into different integrity levels. However, PolyScope requires a rooted phone, precluding the possibility of using it in a static analysis framework. Hernandez et al. [26] proposed BIGMAC that combines DAC and MAC to construct an attack graph representing allowed data-flows between subjects and objects in a running system. BIGMAC succeeds to recover the running system's security state purely though static analysis with high accuracy. This makes it an ideal candidate as a base on which we can bootstrap more in-depth analysis. Although BIGMAC serves to abstract away the complexity of the Android security model, which was recently detailed by [33], it cannot detect DAC checks that occur dynamically in a running process.

4. Design

An overview of the architecture of the tool can be seen in Figure 1, we describe these steps in further detail below. The tool begins with the firmware image extraction. The filesystem is unpacked and extracted to acquire the SELinux policy, daemon binaries, and the init RC scripts. The SELinux policy is analyzed using a modified version of BIGMAC [26] to query the processes an untrusted app can communicate with via sockets. The results from the analysis are then compiled into a list that is separately filtered with the daemon binaries and the relevant service definitions we have been able to extract. Once filtered with the service binaries, the tool conducts binary analysis to verify the socket address and find any security checks that may be in the binaries. Running in parallel is the Init RC Service Definition Analysis which will extract socket addresses and file permissions from the init RC files. The
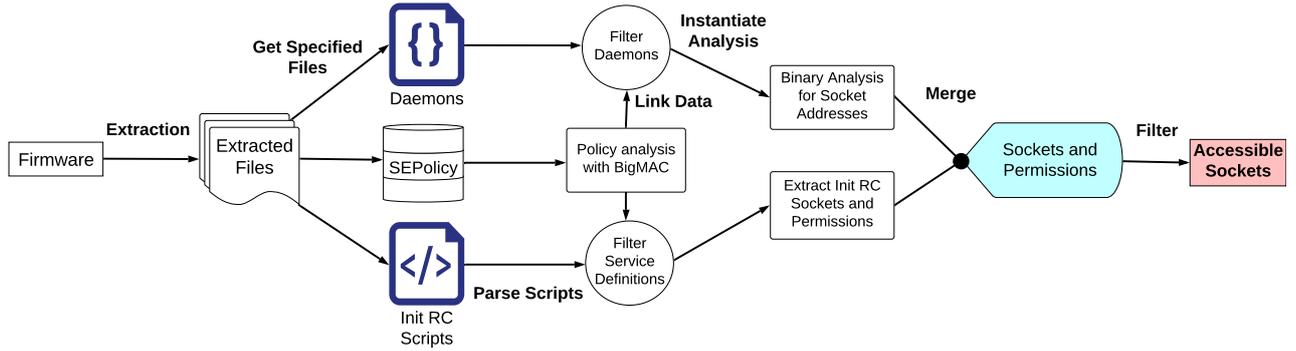
Figure 1. SAUSAGE Framework Architecture. Following the unpacking of an image, the SELinux policy is analyzed to find the processes an untrusted app can communicate with via Unix domain sockets. The processes' binaries and their RC service definitions are retrieved from the extracted filesystem and analyzed independently to find the socket addresses that the process uses, their access control configurations, and any authentication checks in the binary code. The list of socket addresses is filtered down to the ones an untrusted app can access based on their access control permissions.

output of the binary analysis is combined with the output of the Init RC Service Definition Analysis and compiled into a list that can be used to filter for accessible sockets.

4.1. Threat Model

Unix domain sockets can only be accessed from processes that have proper permissions when checked by the MAC and DAC. Our threat model specifically focuses on untrusted apps, labeled `untrusted_app` in SELinux. In our threat model, an app is allowed to obtain middleware permissions that can be granted to any `untrusted_app`. Middleware permissions are not part of the MAC or DAC, so they are not directly considered when a socket is being accessed. However, the DAC supplementary groups assigned to an untrusted app depend on the permissions it has. Thus, we assign the following supplementary group to an untrusted app in our threat model:

The `android.permission.INTERNET` permission, which corresponds to the `inet` group, allows the `untrusted_app` to perform network operations, such as opening network sockets. The `android.permission.BLUETOOTH_ADMIN` permission, which corresponds to `net_bt_admin`, allows applications to discover and pair Bluetooth devices. Similarly, the Bluetooth permission labeled `android.permission.BLUETOOTH` permission, corresponding to `net_bt`, allows applications to connect to paired Bluetooth devices. The final permission is the external storage permission, `android.permission.MANAGE_EXTERNAL_STORAGE`, which belongs to the `external_storage` group, allows an application a broad access to external storage in Scoped storage, a feature in Android allowing an application to only have access to their application directory on external storage plus any media created by the app [13].

4.2. Image Extraction

The initial step of our tool is the firmware image extraction from a repository of firmware images that we have collected. A majority of the images have either been downloaded directly from vendor specific websites, such

as the AOSP firmware images website [2], or from third-party websites such as firmwarepanda [3]. The image packing format varies by vendor, e.g., HTC required an RUU Decrypt tool to account for the `Rom Update Utility` format, while Samsung required `LZ4` decompression. Thus, we rely on existing unified tools that support every Android vendor and version to avoid developing our own from scratch. We use a modified version of the ATextract tool [44] to extract the files needed to analyze the SELinux policies and native daemons from the firmware images. For Android versions higher than 8.0, and for vendors that are not supported by the modified version of the ATextract tool, we use the newly released unpacking tool developed by Possemato et al. [37]. The tool was modified to restructure the extracted image in the right format to be interpreted by our analysis pipeline. Our analysis pipeline collects DAC/MAC/CAP metadata, init RC files, daemons, shared libraries and SELinux policy files to be used by BIGMAC and our binary analysis component.

4.3. SELinux Policy Analysis

Following extraction, our framework reasons about the system's access control policies in order to determine which processes an untrusted app can connect to through Unix domain sockets. The Android security model is based on the complex interaction of multiple security layers, including SEAndroid policies, Linux filesystem permissions and Linux capabilities. Thus, we use a version of BIGMAC that we modified, to extract and recreate the security state of the running system. BIGMAC is a fine-grained SELinux policy static analysis tool [26]. It first goes through the filesystem and extracts files' DAC file permissions, SELinux labels and Linux capabilities. Then, it parses the system's init scripts and simulates commands that affect the filesystem (e.g., `mkdir`, `chmod`), as well as `service` commands which execute service binaries. Performing boot emulation is required to create files in the `/sys`, `/dev` and `/data` directories, which would not be present in a static firmware image. We extended BIGMAC's init boot simulation step to parse the `socket` option of `service` commands in order to create the socket files in the /dev/socket/ directory. For a full discus-

sion of BIGMAC, we refer readers to the original paper by Hernandez et al. [26].

Once the modified version of BIGMAC finishes analyzing an image and generating an attack graph, it provides a query engine that can be used to find all the objects an untrusted app can write to. We can filter the resultant list of objects to only include IPC objects of the "socket" type. Since each IPC object holds a reference to its owner process, we extract the file path of the process's binary executable. The next steps of the analysis are then performed on these binaries.

4.4. Socket Address Extraction

Once we have the set of binary files for processes that an untrusted app can communicate with through Unix domain sockets, we can start to extract the socket addresses that these processes are listening over. We employ two methods for each one of these service binaries: init RC parsing and static binary analysis. These two methods are complementary to each other; parsing init RC files guarantees all RESERVED socket addresses will be recovered, and static binary analysis will recover all three types of socket addresses that the binary might be listening over.

4.4.1. Init RC Service Definitions. For each one of these binaries, there exists one or more service definitions in the Android system's init RC files. These service definitions can have options that configure how and when `init` runs these files. One of these options, `socket`,[2] creates a socket file for the service in the RESERVED namespace, creates a file descriptor for this socket and binds it to the created socket file, and saves the socket's file descriptor as an environment variable[3] for later retrieval by the service process. Therefore, it is straightforward to retrieve all RESERVED socket addresses from service definitions, by finding and parsing the `socket` options. However, this method does not capture socket addresses in other namespaces.

4.4.2. Static Binary Analysis. In the binary analysis module, we first construct the Control Flow Graph (CFG) of the binary and all externally linked objects, and identify all defined functions. We then perform an inter-procedural dataflow analysis starting at the entry point of every function that calls the `bind` system call in the binary. At the `bind` callsite, we extract the value of the address argument. The address is checked to determine whether or not it is a Unix domain socket address. If it is, we detect the namespace that the address belongs to by checking the first character of that address. If it is a null byte, then it belongs to the ABSTRACT namespace and no further analysis is needed. If the address starts with a directory separator ('/'), then it belongs to the FILESYSTEM namespace, and we attempt to determine the permissions the socket

file is created with. This is done by detecting all preceding `umask`, `seteuid` and `setegid` system calls in the binary and extracting their arguments. The same process is carried out for subsequent `chmod`, `fchmod`, `chown`, and `fchown` calls. Additionally, the static binary analysis module detects RESERVED socket addresses by performing the same type of dataflow analysis for functions that call `getenv`. If the requested environment variable name starts with the "ANDROID_SOCKET_" prefix, the rest of the environment variable name is saved as a RESERVED socket address.

4.5. Peer Credential Check Extraction

The `getsockopt` system call [4], when invoked with a file descriptor `sockfd`, retrieves the value of various options for the socket pointed to by `sockfd`. The option retrieved is specified by the `optname` argument, which is an integer corresponding to a valid socket option. The retrieved option is stored in the pointer specified by the `optval` argument. In our case, we are mainly interested in `getsockopt` calls where the `optname` is specified as `SO_PEERCRED` (0x11). In this case, the connected peer's credentials are stored at the `optval` pointer in a `ucred` struct. This struct contains three member variables: the process ID (PID), user ID (UID) and group ID (GID) of the connected peer.

Using the same CFG used in the Socket Address Extraction step, we perform another dataflow analysis of every function that invokes the `getsockopt` system call. First, we check the value of the option name (`optname`) argument. If the function is called with the `SO_PEERCRED` optname, we track all subsequent uses of the returned credentials in the function and record which credentials are being used. We use the same categorization used in [40]; UID- and GID-based checks are considered secure, while PID checks are considered weak. Additionally, we attempt to detect and categorize uses of these credentials. We have identified two types of uses: (1) *Integer comparisons:* We detect whenever a credential is used in a comparison instruction and record the operand if it is a constant integer, or "UNDEFINED" if it is not. (2) *Function arguments:* We detect whenever a credential is used as a function argument and record the function address and name (if it was not stripped). With this usage information, we can determine exactly what credentials a connected peer needs to be able to communicate with the process through a given socket, thus greatly aiding in further interpretation of the analysis result.

4.6. File Permission Analysis

Following the detection of all socket addresses and their filesystem permissions (if any), we check whether an untrusted app with all possible permission-mapped GIDs can access the socket file for each FILESYSTEM or RESERVED socket address. We use a modified version of BIGMAC to reason about the MAC policy and determine whether an untrusted app has access to a socket file. We also inspect the socket file's permission bits, UID and GID to determine access w.r.t. the DAC policy.

---

2. The `socket` option follows the syntax: `socket <name> <type> <perm> [ <user> [ <group> [ <seclabel> ] ] ]` where `<name>` is the address of the socket, `<perm>`, `<user>` and `<group>` are its credentials on the filesystem, and `<seclabel>` is its SELinux label.

3. This environment variable's name is formatted as `ANDROID_SOCKET_<address>` where `<address>` is the address of the socket in the reserved namespace.

TABLE 2. ANDROID-SPECIFIC BIND APIs

Function	Namespace	Library
FrameworkListener	RESERVED	libsysutils.so
SocketListener	RESERVED	libsysutils.so
android_get_control_socket	RESERVED	libcutils.so
socket_local_server_bind	Any	libcutils.so
socket_local_server	Any	libcutils.so

5. Implementation

In this section, we discuss the implementation of the three most crucial parts of the SAUSAGE architecture, which includes the extension of BIGMAC to better serve our use case, the static analysis of daemon binaries using angr [42], and the final step of filtering and categorizing accessible sockets.

5.1. BigMAC Query

The first step following successful extraction of a firmware image is the SELinux policy analysis. We implemented an easy-to-use API that exposes the most crucial functionalities of BIGMAC to the developer. This API facilitates running the whole workflow of BIGMAC in a single call, and implements an interface to the prologue engine that facilitates query operations on the generated Attack Graph. We use this API by specifying the path of the extracted SELinux policy and running the attack graph instantiation. Using this attack graph, we run the query: `query(untrusted_app, _, 1)` to retrieve all nodes in the graph that an untrusted app can write/connect to. We then filter only socket objects from the resultant list. Since socket objects are `IPCNodes`, they hold a reference to the owning process. Thus, we can retrieve all the processes, and their executable binaries, that an untrusted app can connect to through a Unix domain socket.

Additionally, we extended BIGMAC's init boot simulation step to handle `socket` options under service definitions in init RC files. On encountering a `socket` entry under a service definition, BIGMAC now creates the corresponding file in the simulated filesystem as part of the boot process with the specified permissions. Additionally, it assigns the correct SELinux context to the socket file based on the extracted filesystem contexts. This addition is essential as BIGMAC removes filesystem contexts that do not have a backing file from the attack graph which would have prevented us from querying whether an untrusted app has access to these socket files.

5.2. Static Binary Analysis

Our static binary analysis implementation contains three modules (about 2K LoC). The Socket Address Extraction module extracts socket addresses that the binary is listening over by analyzing `bind` call sites. The DAC Check Extraction module detects and analyzes DAC checks by performing data flow analysis after `getsockopt` calls with the `SO_PEERCRED` argument as discussed in Section 4.5. Each daemon binary an untrusted app can connect to is statically analyzed to retrieve all the Unix domain socket addresses it listens on and the

permissions they are created with, as well as detect any hardcoded DAC checks in the binary.

We implement our static binary analysis using the angr framework [42]. We first generate the CFG of the analyzed binary during which function prologues are detected and stored in the angr project's knowledge base. The dataflow analysis is implemented by angr's intra-procedural `ReachingDefinitions` analysis [1] and is used in both socket address extraction and DAC check extraction. To make the analysis inter-procedural, we implement a `FunctionHandler` which handles function calls by performing the `ReachingDefinitions` analysis recursively based on [36].

Socket Address Extraction. First, we find all call sites to the `bind` system call. This is done by finding the `bind` function node in the CFG and listing all of its predecessor nodes where the connecting edge is of type `Ijk_Call`, signifying a function call. For each one of these nodes, we find the function that it belongs to, and perform an inter-procedural data flow analysis on that function. The `FunctionHandler` also simulates common libc string manipulation functions, such as `strcpy`, `sprintf` and others, in order to capture dynamically constructed socket addresses at the `bind` callsite. These string manipulation handlers are implemented in order to avoid inaccuracies caused by the complex control flow structures associated with string operations. Additionally, Android provides additional utility APIs for system daemons to create, bind, and listen over local sockets. These functions are found in `libcutils.so` and `libsysutils.so` and are detailed in Table 2. We perform the same analysis at the call sites of these functions to recover socket addresses passed to these utility functions.

DAC Check Extraction. The same dataflow analysis implementation is used for DAC check extraction. We analyze call sites of the `getsockopt` system call, and check its arguments. If the `optname` argument is set to `SO_PEERCRED`, we track usages of the `ucred` struct, stored in the pointer `optval`, by tainting its member variables. We record any variables used and attempt to identify the type of usage as discussed in Section 4.5.

5.3. File Permission Analysis

We add additional functionality to BIGMAC allowing us to add previously undetected files, e.g., files created dynamically by a running process. Following the recovery of FILESYSTEM socket addresses and their DAC metadata, we insert these filenames along with their metadata into BIGMAC's recovered filesystem, and rerun BIGMAC's workflow. This will assign the correct SELinux labels to these files in an automated manner, which allows us to determine whether a socket file is accessible through simple queries of the Prolog engine.

6. Evaluation

In this section, we first present our findings on accessible sockets, and security downgrade; we then report on the performance measurements and ground truth comparison. The number of firmware images analyzed per vendor and Android version can be found in Figure 2. Accessible
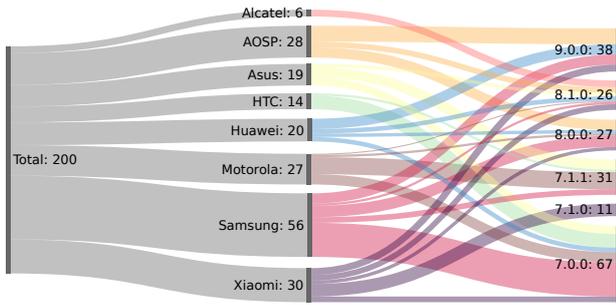
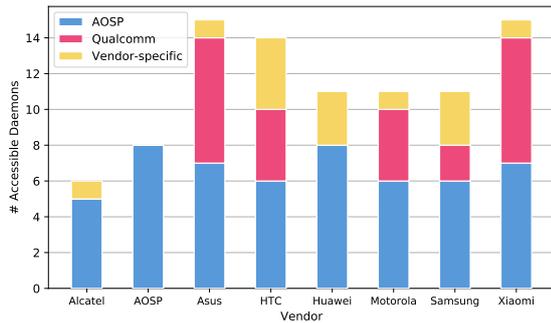Figure 2. Distribution of Android versions and vendors in our corpus

Figure 3. Accessible system daemons by vendor in our corpus

sockets are the sockets that we have identified to be accessible for an untrusted app without any prerequisites (following our threat model). For security downgrade, we consider daemons that exist in AOSP Android but have weaker security protections due to vendor customization of access control policies and customization of the daemons themselves.

6.1. Accessible Sockets

Figure 3 displays the number of accessible system daemons by vendor. In this figure, we count daemons that are accessible to an untrusted app by SELinux and for which we have found at least one socket. We divide them into three categories: AOSP daemons, Qualcomm daemons and vendor-specific daemons. We count AOSP daemons with additional vendor-specific sockets as vendor-specific daemons. From these daemons, we identified 28 unique socket addresses that an untrusted app is allowed to connect to by the access control policy in at least one of the firmware images analyzed. We present 17 of these socket addresses and their daemons in Table 3. We omit the remaining 11 sockets from the table as they are intended to be accessible to untrusted apps as part of the Android framework. We also specify any authentication checks performed on the client after a connection is established, as well as the list of vendors these socket addresses were detected on.[4] In the following, we discuss each daemon's functionality and its accessible sockets. Information about proprietary daemons' functionality is not publicly available. Therefore, we infer their functionality based on static

---

4. We only specify the vendors with firmware images where the system daemon that uses these sockets is *accessible*. If the daemon exists in other vendors' firmware images, we do not include it.

analysis of the binaries, as well as whatever information we can find online.

6.1.1. AOSP Daemons. AOSP daemons are available in all Android systems as part of AOSP Android. The AOSP version of these daemons' source code is made available by AOSP, although vendors might make proprietary customizations to them in their own distributions of Android. These daemons purposefully expose sockets for communication with an untrusted app to implement various functionalities. These sockets were detected consistently across our dataset, confirming our tool's reliability at detecting accessible sockets. Since these sockets are intended to be accessed by untrusted apps, we omit these accessible sockets from Table 3 and briefly discuss the daemons' functionality instead.

The *logd* daemon is a centralized logger implementing all logging operations in Android [10]. It utilizes three sockets, all of which are accessible to an untrusted app: logd, logdr, and logdw. The *netd* daemon is responsible for managing network interface configurations [10]. In AOSP Android, the *netd* daemon utilizes four socket addresses: netd, dnsproxyd, mdns and fwmarkd. Of these sockets, dnsproxyd and fwmarkd are accessible to an untrusted app with the INTERNET permission. The *surfaceflinger* daemon's main functionality is to compose and render multiple display surfaces onto the display [10]. In Android 8.0+ images, we found three accessible RESERVED socket addresses from the *surfaceflinger* daemon binary located in the pdx/system/vr/display/ directory and include client, manager, and vsync. The *tombstoned* daemon was added in Android 8.0 and it plays a role in capturing crash data from a system and storing it for further analysis. The *traced* daemon is part of an open-source solution developed by Perfetto [16] and used in Android for system profiling, app tracing and trace analysis. The *perfd* daemon collects information to keep track of performance on the system.

6.1.2. Qualcomm Daemons. Qualcomm provides a wide range of hardware and peripherals on Android devices, such as the processor and the Mobile Station on Modem (MSM) system on chip (SoC). For interoperability between these peripherals and the operating system, Qualcomm implements daemons that bridge the communication between these devices and the rest of the Android framework. These daemons were found to be accessible across multiple vendors' images in our dataset. In AOSP however, these daemons exist, but none of them are accessible to an untrusted app. This discrepancy indicates that access control policies placed by AOSP were relaxed by other vendors where these daemons were found to be accessible.

cnd. The *cnd* daemon manages Qualcomm Connectivity Engine which chooses between 3G/4G and Wi-Fi networks based on their performance for the specific application a user is using [9]. It is a proprietary daemon, therefore its exact functionality is not publicly known. In Android versions prior to 8.0, it uses the cnd RESERVED socket and a FILESYSTEM socket located in /dev/socket/nims, both of which are accessible to an untrusted app with the INTERNET permission. The /dev/socket/nims socket was also found to be world-

TABLE 3. SOCKET ADDRESSES AN UNTRUSTED APP CAN CONNECT TO, THEIR SYSTEM DAEMONS, AUTHENTICATION CHECKS THEY IMPLEMENT AND THE VENDORS WHERE THEY WERE FOUND TO BE ACCESSIBLE

Address	Namespace	System Daemon	Auth. Checks	Vendor
/dev/socket/nims	FILESYSTEM	cnd	None	Asus, HTC, Motorola, Xiaomi
cnd	RESERVED	cnd	GID or AppName	Asus, HTC, Motorola, Xiaomi
qvrservice	RESERVED	qvrservice	None	Asus, HTC, Samsung, Xiaomi
qvrservice_camera	RESERVED	qvrservice	None	Xiaomi
seempdw	RESERVED	seempd	None	Asus, HTC, Xiaomi
@fmhal_sock	ABSTRACT	fmhal_service	UID	Asus, Motorola, Xiaomi
@qcom.dun.server	ABSTRACT	dun-server	None	Asus, Xiaomi
@cand.socket.ctrl	ABSTRACT	cand	None	HTC
@cand.socket.msg	ABSTRACT	cand	None	HTC
dmagent	RESERVED	dmagent	None	HTC
cfiat	RESERVED	rild	UID	HTC
kipc	RESERVED	rild	UID	HTC
/dev/socket/dpmwrapper	FILESYSTEM	dpmd	None	HTC, Xiaomi
@btloggersock	ABSTRACT	bt_logger	None	Motorola, Xiaomi
@dev/socket/jack/set.priority	ABSTRACT	apaservice	None	Samsung
napproxyd	RESERVED	netd	None	Samsung
tcm	RESERVED	dpmd	None	Xiaomi, Asus
dpmwrapper	RESERVED	dpmd	None	Xiaomi, Asus

accessible in some HTC, Motorola and Xiaomi images. It is unclear whether this is a result of misconfiguration or a change of the socket's functionality.

qvrservice. The *qvrservice* daemon is a proprietary daemon that manages Qualcomm VR service. It exposes a world-accessible RESERVED socket qvrservice. On Xiaomi Android 9.0 images, it also exposes the qvr_camera RESERVED socket with the same permissions.

seempd. The *seempd* daemon is part of the Qualcomm Trusted Execution Environment stack (QTEE [38]). It exposes a world-writable DGRAM socket with address seempdw.

dpmd. The *dpmd* is a daemon which stands for Data Port Mapper, and is part of the QTI DPM Framework [35]. It is unclear what functionality it provides. This daemon utilizes two sockets: a RESERVED socket with address dpmd and a FILESYSTEM socket with address /dev/socket/dpmwrapper. The dpmd socket is configured to be inaccessible to apps. In Android 8.0 images, it uses an additional RESERVED socket named tcm, and the FILESYSTEM socket /dev/socket/dpmwrapper was changed to a RESERVED socket "dpmwrapper". The dpmwrapper and tcm RESERVED sockets require the `INTERNET` permission to be able to connect.

dun-server. *dun-server* is a daemon that implements and manages Dial-Up Networking over Bluetooth [41]. It listens over the @qcom.dun.server ABSTRACT socket address. It contains no authentication check, allowing any client to connect to dun-server over this socket. Additionally, manual static analysis of the binary reveals that this socket is bound and closed repeatedly by *dun-server*. Since this is an ABSTRACT socket, any process can create one with the same name as long as that address is not being used. Therefore, a malicious process can bind to this socket address before *dun-server* re-binds it, denying *dun-server* from using it and disrupting its workflow.

fmhal_service. *fmhal_service* is an open-source daemon that manages FM radio on supported systems [25]. It exposes an ABSTRACT socket with address @fmhal_sock. Since this socket is ABSTRACT, it is accessible by default. Thus, any app can establish a connection to it. However, when a client connects, a UID check is performed

to ensure that the client's UID is one of root, system or bluetooth.

bt_logger. *bt_logger* is an open-source daemon that has the ability to log Bluetooth traffic [20]. It exposes an ABSTRACT socket with address @btloggersock with no authentication check, allowing any client to start/stop Bluetooth snooping.

6.1.3. Vendor-specific Daemons. Vendor-specific daemons are daemons that are developed by the Android device manufacturer and bundled with their operating system distribution. A daemon is classified as vendor-specific if it is present in the Android images of a single vendor. In our results, two out of three accessible vendor-specific daemons run as root, presenting valuable targets for exploitation. The third daemon provides an interface to a function available only to processes running privileged UIDs.

dmagent. HTC *dmagent* is a proprietary daemon that manages the Open Media Alliance Device Management protocol. dmagent runs with UID root and listens over the RESERVED socket address dmagent. The socket is configured to be accessible to applications with the `INTERNET` permission. This socket has been previously used to issue copy file command to the daemon which acts to copy files as root from arbitrary source to arbitrary destination [29]. Our analysis shows that this socket remains unprotected by access control policies or authentication checks, making it a prime target for exploitation of a root process.

cand. HTC *cand* is a proprietary daemon which runs as root and listens over two ABSTRACT socket addresses: @cand.socket.ctrl and @cand.socket.msg. Through static analysis, we determined that the daemon serves as an interface to communicate over the CANBus, although the specific use case is not clear. Nevertheless, since it runs as root and listens over unprotected ABSTRACT sockets, it may present another security risk much like dmagent.

apaservice. The *apaservice* daemon is part of the Samsung Android Professional Audio framework [34]. It runs with a UID of "jack." We found that it creates and listens over one ABSTRACT socket address

@dev/socket/jack/set.priority which is used as an interface through which it calls `android::requestPriority` with the parameters it receives. According to AOSP source code [8], this functionality should only be exposed to processes with the audioserver, cameraserver and bluetooth UIDs.

6.2. Downgraded Security

A system daemon is considered to have downgraded security if the vendor relaxes SELinux rules that would have prevented communication between the daemon and an untrusted app in AOSP. To find these instances of downgraded security, we go over the list of daemons an untrusted app can communicate with for each vendor. We then try to find each daemon in that list and its service definition in the corresponding AOSP Android version. If the service exists in AOSP and is enabled, but is not accessible by an untrusted app, then we flag it as a security downgrade.

HTC dumpstate. The *dumpstate* system daemon is an AOSP system daemon that can generate logs that are used to collect details of device-specific issues; an untrusted app is disallowed to communicate with this daemon in AOSP. HTC relaxed this restriction and added two extra sockets to the daemon: htc_dk and htc_dlk. Untrusted app access to these sockets is not allowed by both the MAC and DAC policies. However, as per the comments of the `file_contexts` file, the htc_dlk socket sends kernel log messages to a system app. This is a bad security practice as kernel logs can hold sensitive information, and pre-installed apps packaged with vendor-customized firmware have been shown to be insecure [21].

HTC rild. *rild* is the Radio Interface Layer daemon in Android [11], [12]. It provides an abstraction layer between Android telephony services layer and the radio hardware layer and handles all telephony operations such as call handling, SMS, and others. In Android versions prior to 8.0, *rild* utilizes three sockets: rild, rild-debug and sap_uim_socket1. In AOSP Android, communication with rild using Unix domain sockets is not allowed for untrusted apps by the SELinux policy. In HTC images, our SELinux policy analysis showed that the policy was relaxed and an untrusted app was allowed to communicate with *rild*. Furthermore, we detected two vendor-specific sockets, kipc and cfiat, both of which grant read and write access to an untrusted app with the `INTERNET` permission. These socket addresses have only been detected on the HTC firmware images we analyzed, and their file contexts are labeled htc_cfiat_socket and htc_kipc_socket, confirming that they originate from an HTC-specific vendor customization of *rild*. We detected a UID-based authentication check in the HTC *rild* binary, leaving these sockets potentially protected only by a single post-connection DAC check. Therefore, if a malicious app changes its UID through a privilege escalation exploit, it can gain access to these sockets, which would not have been possible if the SELinux policy had not been relaxed.

cnd. The *cnd* daemon can be available in AOSP firmware images. In all of the AOSP and Samsung images tested, an untrusted app does not have access to this daemon. On the other hand, Asus, HTC, Motorola and Xiaomi firmware images allow an untrusted app to communicate with this daemon. In this case, the daemon exposes two sockets: cnd and /dev/socket/nims that are accessible to an untrusted app, one of which has no authentication checks. Both require the app to have the `INTERNET` permission.

Asus mm-qcamera-daemon. In Asus Android images, *mm-qcamera-daemon* is allowed to communicate with an untrusted app. It contains a socket named at address /data/misc/camera/cam_socket in the FILESYSTEM namespace. The socket itself is inaccessible by both the MAC and DAC policies.

6.3. Abstract Sockets

Our analysis aims to find sockets accessible to untrusted apps. However, we report on the ABSTRACT sockets in our results that are vulnerable to DoS attacks by untrusted apps. Such sockets become vulnerable to DoS if the daemon that owns the socket closes the socket at any point in its operation, or if the daemon exits for any reason. We detect the first case through manual static analysis using Ghidra [17]. This is done by detecting `close` calls on the socket that was previously bound to an ABSTRACT address. If `close` is called anywhere outside of a final cleanup of the daemon's resources during termination, then we flag it as vulnerable to DoS. As for the second case, we detect daemons that are started or stopped by property triggers. These triggers are defined in init RC files and are used to start/stop a service daemon when the specified system property changes, depending on the value of the system property. A malicious app can cause DoS of the vulnerable daemon in either of these cases by repeatedly attempting to bind the ABSTRACT socket that the daemon would usually bind. As a result, if the daemon is stopped, restarted, or closes the ABSTRACT socket, the malicious app will be successful in binding it. Consequently, the daemon will fail to bind this ABSTRACT socket again, disrupting IPC between the daemon and other processes that rely on it.

Four of the ABSTRACT sockets that we detected are vulnerable to DoS, as they match the criteria defined above: @dev/socket/jack/set.priority, @fmhal_service, @qcom.dun.server and @btloggersock. @dev/socket/jack/set.priority is bound by *apaservice* and is triggered by a service call to `IAPAService::StartJackd`. Therefore, a malicious app can occupy this ABSTRACT socket address before another app makes the service call to `IAPAService::StartJackd`. When the service call is made, *apaservice* would fail to bind the socket. However, this failure is handled gracefully by *apaservice* so that its other functionalities would remain unaffected. @fmhal_service and @btloggersock are bound by their system daemons on initialization, but the daemons themselves are triggered by a property trigger in init RC files. A malicious app can bind either of these addresses while the corresponding property is not set. When the property is set and the daemons are started, they fail to bind the socket address and terminate due to the resulting bind error. @qcom.dun.server is bound and closed repeatedly by *dun-server* after every connection. Exploiting this DoS vulnerability would require a race-

condition, where a malicious app attempts to bind the @qcom.dun.server address before *dun-server* re-binds it.

Additionally, by applying our framework to find accessible sockets for the `system_app` context, we found two ABSTRACT sockets, @com.mtk.aee.aed and @aee:rttd created by the *aee_aed* daemon in the Alcatel Android 8.1 firmware. AEE stands for Android Exception Enhancement, and the daemon serves to collect and log backtraces of crashes on the system. We found that the daemon restarts on any configuration change, e.g., triggered by a system app, allowing a malicious app to occupy this ABSTRACT socket address and resulting in DoS of the daemon. This can stop the daemon from logging crash information on the system, potentially hiding evidence of exploit attempts.

6.4. Ground Truth Evaluation

To confirm the correctness of our framework in detecting sockets and their access control properties, we ran a ground truth evaluation on Samsung devices running Android 7.0 and 8.0, and a Motorola device running Android 7.1.1. The test was carried out by an app we developed which obtains the permissions we listed in our threat model. The app runs the `netstat -xl` command to list all the listening Unix domain sockets and their addresses. The app then tries to connect to each socket address and displays a table containing the socket addresses and the result of the connection attempt. However, this app is not part of our framework and only serves to collect ground truth data for evaluation.

On all devices, the app successfully connected to the fwmarkd, dnsproxyd, logd, logdr RESERVED socket addresses. On the Motorola device, the app also reported a connection to the perfd RESERVED socket address. All of these socket addresses were accurately detected by our framework as accessible sockets. For the Samsung device, our analysis detected an ABSTRACT socket owned by the *apaservice* daemon at @dev/socket/jack/set.priority which was not found on the running device. We later discover that the socket is created only after a Samsung-specific service is activated through a Binder call and discuss the details in Section 7. This demonstrates the effectiveness of our approach compared to dynamic analysis which may not detect sockets created conditionally or in response to a trigger. Note that our ground truth evaluation app is a simplified implementation of the *Connection Tester* dynamic analysis module used in [40]. Thus, we claim that our approach achieves a better socket detection rate due to the higher coverage inherent to static analysis. The accessible sockets detected by the testing app can be found in Table 4 in Appendix B.

6.5. Performance Evaluation

We ran our analysis on a PC with the Intel(R) Core(TM) i7-8700 CPU @3.20GHz and 16 GB RAM. On average, each firmware was analyzed in 770 seconds (approx. 14 minutes). Figure 4 shows a box plot of the time taken for each firmware image. The static binary analysis takes a majority of that time at an average of 736 seconds (approx. 14 minutes), resulting in a large variance for each image, depending on how many binaries are being
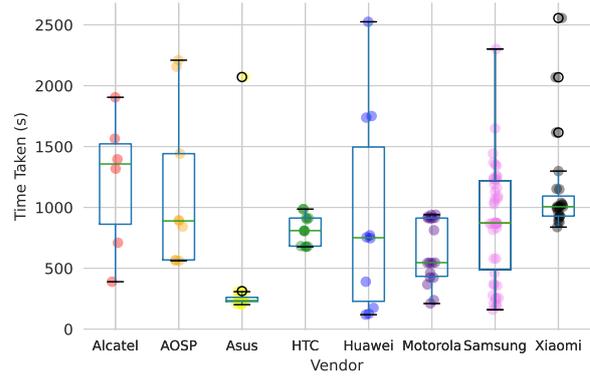

Figure 4. Analysis time for firmware images in our corpus

analyzed, i.e., how many service daemons are accessible to an untrusted app, and how complex their binaries are. The remaining time is used in the initial and final steps in the instantiation and querying of BIGMAC. Currently, our prototype does not implement obvious optimizations, such as parallelization of the binary analysis step for each binary. We leave these engineering tasks as future improvements to our framework. Based on monitoring the memory usage for our analysis, parallelization would have resulted in a 3x speedup on the same PC.

7. Case Studies and Responsible Disclosure

In this section, we focus on two interesting cases from our results that translate to vulnerabilities that a malicious app can potentially exploit. We discuss Samsung *apaservice* daemon's FILESYSTEM socket and how it can be used to request a priority for any process ID or thread ID, a functionality only available to processes with the audioserver, cameraserver or bluetooth UID. We discuss another potential permission bypass exploit in Qualcomm's *cnd* and *dpmd* daemons, both of which implement an identical check based on the connecting process's name.

Samsung apaservice. Our analysis indicates that Samsung's *apaservice* daemon creates an ABSTRACT socket with address @dev/socket/jack/set.priority. In our ground truth evaluation, this socket was not found at first on the running device. We analyzed the *apaservice* binary to determine the reason, and found that this socket is only created after calling `APAService::startJackd`. This method is exposed by the service `com.samsung.android.jam.IAPAService`. After calling this method using the `service call` command, the socket was created and appeared in the `netstat` output. This demonstrates the effectiveness of our analysis in uncovering sockets which are created only under certain conditions, as compared to dynamic analysis.

The socket at @dev/socket/jack/set.priority is a DGRAM socket under *apaservice* that accepts messages from any client, with no DAC check after receiving a message. It receives messages of the format "*4<pid>,<tid>,<priority>," where <pid> is a process ID, `tid` is a thread ID, and <priority> is

the requested priority. These values are then passed to the function `android::requestPriority`, which requests the SchedulingPolicyService to assign a priority to the requested process ID and thread ID. Although this request is typically available to system processes running under the audioserver, cameraserver and bluetooth UIDs, an untrusted app can set its own priority, or the priority of any other process through this socket, effectively bypassing authentication checks.

Additionally, through manual analysis, we discovered that the function that handles messages received over this socket, `android::APAService::handlePriorityMessage`, is vulnerable to buffer overflow. By sending the correct preamble, "4*", followed by 25 bytes of data, the *apaservice* daemon crashes due to stack corruption. The backtrace logs show that the return address was successfully overwritten. The impact of this buffer overflow vulnerability can range from DoS of *apaservice* to privilege escalation to the more privileged "jack" UID (recall that this UID is used by *apaservice*) in the less restrictive `apaservice` SELinux context. We developed a Proof-of-Concept (PoC) for this buffer overflow that crashes the daemon, achieving DoS. Achieving local code execution would require bypassing the buffer overflow protections compiled into the daemon binary. *Apaservice* exists in Samsung Android up to version 8.1. Within our analysis, we first noted *apaservice* in Samsung Android 7.0. This indicates the lifetime of this vulnerability would, at least, be from the release of Samsung Android 7.0 on Aug. 22, 2016 through when the vulnerability was reported on Sept. 10, 2021 by us.

Qualcomm cnd and dpmd. In 18 Xiaomi and four HTC images analyzed, we found that *dpmd* implements an insecure authentication check after a connection is established. The *cnd* daemon implements an identical authentication check in 14 HTC images and seven Motorola images. Both of these daemons are started by init with root UID, but later drop to system UID through a `setuid` call. The authentication mechanism works as follows: after a connection with a new client is established, the client's DAC credentials are retrieved using a `getsockopt` call. First, the GID is compared against a list of GIDs that are allowed by the daemon. If the GID does not match any of the allowed GIDs, then the PID is used to retrieve the connecting client's process name by reading the `/proc/<pid>/comm` file for that process. In Unix systems, this file exists for every process and contains the process name. The process name is then compared to a list of allowed process names, and access is granted if a match is found. This is however an insecure check, as any app can change its own process name dynamically, even if a different process has the same name. Therefore, a malicious app can bypass this check trivially by changing its name to that of an allowed process.

In the case of *dpmd*, this check is implemented for an inaccessible RESERVED socket of the same name, "dpmd," and an untrusted app is not allowed to connect to by both MAC and DAC. On the other hand, *cnd* implements this check on its "cnd" RESERVED socket, which is accessible to an untrusted app. Through static analysis, we infer that this socket allows clients to get/set network settings such as WiFiAP, WiFi P2P, and Default Network settings, by sending the appropriate command over the cnd socket.

Responsible Disclosure. We sent detailed reports of these vulnerabilities to Samsung and Qualcomm, including the aforementioned PoC for Samsung. Samsung acknowledged the vulnerability in *apaservice* and patched it in in SMR Sep-2021 Release 1. The vulnerability was assigned CVE-2021-25461. They also rewarded our findings through their bug bounty program on Bugcrowd. Qualcomm's response to our disclosure was that the unprotected *cnd* socket is deprecated starting Android 8.0. Thus, they will not be patching the affected systems, despite around 180 million users relying on the affected Android versions (7.0-7.1) [14]. As for *dpmd* daemon, they mention that the daemon now uses a more secure UID check although we still see the same issue up to Android 9.0 firmware in our dataset.

8. Limitations

Our analysis approach faces certain limitations. Firmware unpacking and extraction presents the only obstacle to expanding our analysis to more recent Android versions and a wider variety of vendors. Extending the current open-source toolset for Android image extraction requires significant engineering effort, but can pave the way for similar large-scale analyses. Additionally, we discuss the limitations inherent in our static binary analysis approach that make the analysis of statically-linked stripped binaries difficult.

Firmware Unpacking and Extraction. Extracting and unpacking Android images is not trivial as the format of factory images can vary greatly between different Android vendors and versions. Multiple tools have been developed that facilitate the unpacking process or different stages of it. However, to our knowledge, there is no freely available unified factory image unpacking tool that can unpack any firmware image across different versions and vendors, except for the one we used [37], [44]. These tools are outdated, however, and only support Android versions 5-9 for AOSP, and 5-8 for other vendors. Furthermore, within these versions the unpacking success rate is not perfect, and some filesystems may not be recovered. This limits the operable dataset we can use in our analysis, and as a result extracted firmware might have missing daemons or SELinux policy files.

Static Binary Analysis. Our implementation of the static binary analysis relies on detecting Android bind APIs and string manipulation functions by their symbol name. This does not pose a problem in the case of dynamically-linked binaries since external symbols are persevered for linking. However, this becomes problematic in statically-linked stripped binaries. In our analysis, we encountered three cases of statically-linked stripped binaries which we ultimately skipped, namely: *mcDriverDaemon*, *debuggerd* and *adbd*. Additionally, we assume that if a `bind` call exists in the binary with a Unix domain socket address parameter, then that socket is bound on initialization of the daemon. We do not perform reachability analysis to avoid the problem of inaccurately resolving indirect jumps. Additionally, SAUSAGE's implementation contains over-approximations that could produce similar duplicate results (e.g. `@dev/socket/some_socket` vs.

`/dev/socket/some_socket`) in some cases where the socket address is constructed in multiple string manipulation steps.

Manual Analysis. SAUSAGE's output is a report of all socket addresses detected by our analysis, and any DAC checks or file permissions assigned to them. A manual reverse engineering/analysis process needs to be carried out in order to evaluate whether these sockets result in a vulnerability. For instance, in the case of ABSTRACT sockets, the analyst would examine the binary's code to determine if the socket is closed and re-bound in a loop, which would lead to the socket being vulnerable to DoS. This is a limitation since it requires manual effort to probe the results for vulnerabilities.

9. Conclusion

In this work, we present SAUSAGE, a static analysis framework to evaluate the security of Unix domain sockets used in Android. Our approach combines fine-grained access control policy analysis with static binary analysis techniques to comprehensively detect exposed IPC sockets available to an untrusted app. We use this framework to analyze 200 Android images from different vendors and Android versions, and uncover vulnerabilities and access control misconfigurations, such as permission bypass and denial of service. Some of these sockets would not have been discovered by previous work relying on dynamic analysis. We will open-source our static binary analysis module to make it available for the community upon publishing. The source code for our static binary analysis module is published at https://github.com/mounir-khaled/SAUSAGE.

Acknowledgements

References

[1] angr - Analysis and Coordination - angr 9.0.7912 documentation. https://angr.io/api-doc/angr.html#angr.analyses.reaching_definitions.reaching_definitions.ReachingDefinitionsAnalysis.

[2] Factory Images for Nexus and Pixel Devices — Google Play services. https://developers.google.com/android/images.

[3] Firmware Panda – Database of Android Stock ROM Firmware for all Android Devices. https://firmwarepanda.com/.

[4] getsockopt(2) - linux manual page. https://man7.org/linux/man-pages/man2/getsockopt.2.html.

[5] NVD - CVE-2011-3918. https://nvd.nist.gov/vuln/detail/CVE-2011-3918.

[6] NVD - CVE-2013-4777. https://nvd.nist.gov/vuln/detail/CVE-2013-4777.

[7] NVD - CVE-2013-5933. https://nvd.nist.gov/vuln/detail/CVE-2013-5933.

[8] SchedulingPolicyService.java. https://android.googlesource.com/platform/frameworks/base/+/master/services/core/java/com/android/server/os/SchedulingPolicyService.java.

[9] Qualcomm's CnE Brings "Smarts" to 3g/4g Wi-Fi Seamless Interworking. https://www.qualcomm.com/news/onq/2013/07/02/qualcomms-cne-bringing-smarts-3g4g-wi-fi-seamless-interworking, 2013.

[10] *Daemons*, volume 1, pages 126–129, 137–140, 144. Jonathan Levin, 1st edition, 2015.

[11] Radio Layer Interface — Android Open Source Project. https://wladimir-tm4pda.github.io/porting/telephony.html, Sep 2015.

[12] RIL Refactoring — Android Open Source Project. https://source.android.com/devices/tech/connect/ril, Sep 2020.

[13] Manifest.permission — Android Developers. https://developer.android.com/reference/android/Manifest.permission, Jun 2021.

[14] Mobile & tablet android version market share worldwide. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2021.

[15] Mobile Vendor Market Share Worldwide — StatCounter Global Stats. https://gs.statcounter.com/vendor-market-share/mobile/worldwide, May 2021.

[16] perfetto — Android Developers. https://developer.android.com/studio/command-line/perfetto, May 2021.

[17] N. S. Agency. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/, 2019.

[18] A. B. Ayed. A literature review on Android permission system. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 4(4), 2015.

[19] C. Cao, N. Gao, P. Liu, and J. Xiang. Towards Analyzing the Input Validation Vulnerabilities Associated with Android System Services. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, page 361–370, New York, NY, USA, 2015. Association for Computing Machinery.

[20] Codeaurora. bt_logger/src/bt_logger.c · Codeaurora / platform_vendor_qcom-opensource_bluetooth. https://gitlab.com/Codeaurora/platform_vendor_qcom-opensource_bluetooth/-/blob/db31ce2e09daeb551320b4f9d20e56000123edd0/bt_logger/src/bt_logger.c, 2017.

[21] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2379–2396. USENIX Association, Aug. 2020.

[22] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security Privacy*, 7(1):50–57, 2009.

[23] G. Gong. Fuzzing Android System Services by Binder Call to Escalate Privilege. BlackHat USA, 2015.

[24] S. Groß, A. Tiwari, and C. Hammer. PIAnalyzer: A Precise Approach for Pending Intent Vulnerability Analysis. In J. Lopez, J. Zhou, and M. Soriano, editors, *Computer Security*, pages 41–59, Cham, 2018. Springer International Publishing.

[25] S. Gupta. fmhalservice·Codeaurora/platform_vendor_qcom-opensource_fm. https://gitlab.com/Codeaurora/platform_vendor_qcom-opensource_fm/-/tree/44d045c660bcfd9063a77cdfce3694c8f5b9dbef/fmhalService, 2017.

[26] G. Hernandez, D. Tian, A. S. Yadav, B. J. Williams, and K. R. B. Butler. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *USENIX Security Symposium*, 2020.

[27] Huawei. Security advisory - information disclosure vulnerability in aee extension of mtk platform, Aug 2017.

[28] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru. Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–11, 2017.

[29] jcase. [Root] WeakSauce APK - 1.0.1. https://forum.xda-developers.com/t/root-weaksauce-apk-1-0-1.2699089/, Mar 2014.

[30] W. Kai, Z. Yuqing, L. Qixu, and F. Dan. A fuzzing test for dynamic vulnerability detection on Android Binder mechanism. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 709–710, 2015.

[31] Y.-T. Lee, W. Enck, H. Chen, H. Vijayakumar, N. Li, Z. Qian, D. Wang, G. Petracca, and T. Jaeger. PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[32] B.-Z. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge. FANS: Fuzzing Android Native System Services via Automated Interface Analysis. In *USENIX Security Symposium*, 2020.

[33] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich. The Android Platform Security Model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3), Apr. 2021.

[34] D. Morse. Samsung releases Professional Audio SDK for Galaxy devices • DJWORX. https://djworx.com/samsung-professional-audio-sdk-galaxy/, Oct 2014.

[35] V. Oltean. Kang Netmgr QMI DPM (Data Port Mapper) from Xiaomi Daisy. https://review.lineageos.org/c/LineageOS/android_device_lenovo_YTX703-common/+/239742/8, Feb 2019.

[36] Pamplemousse. Handle function calls during static analysis in angr. https://blog.xaviermaso.com/2021/02/25/Handle-function-calls-during-static-analysis-with-angr.html, Feb 2021.

[37] A. Possemato, S. Aonzo, D. Balzarotti, and Y. Fratantonio. Trust, but verify: A longitudinal analysis of Android OEM compliance and customization. In *42nd IEEE Symposium on Security and Privacy*, 2021.

[38] I. Qualcomm Technologies. Mobile Security Solutions: Secure Mobile Technology. https://www.qualcomm.com/products/features/mobile-security-solutions, May 2021.

[39] R. Sasnauskas and J. Regehr. Intent Fuzzer: Crafting Intents of Death. WODA+PERTEA 2014, page 1–5, New York, NY, USA, 2014. Association for Computing Machinery.

[40] 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, page 80–91, New York, NY, USA, 2016. Association for Computing Machinery.

[41] N. Shivpure. src/org/codeaurora/bluetooth/dun · Codeaurora / platform_vendor_qcom-opensource_bluetooth. https://gitlab.com/Codeaurora/platform_vendor_qcom-opensource_bluetooth/-/tree/db31ce2e09daeb551320b4f9d20e56000123edd0/src/org/codeaurora/bluetooth/dun, 2017.

[42] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

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

[44] 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, Baltimore, MD, 2018. USENIX Association.

[45] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. ASIACCS '12, page 75–76, New York, NY, USA, 2012. Association for Computing Machinery.

[46] 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., Aug. 2015. USENIX Association.

[47] D. Yu, G. Yang, G. Meng, X. Gong, X. Zhang, X. Xiang, X. Wang, Y. Jiang, K. Chen, W. Zou, and et al. SEPAL: Towards a Large-scale Analysis of SEAndroid Policy Customization. *Proceedings of the Web Conference 2021*, Apr 2021.

Appendix

1. BigMAC

BIGMAC is a fine-grained SEPolicy static analysis tool [26]. In this section we discuss the functionality the process behind how BIGMAC works and what results can be generated. BIGMAC first walks the filesystem and extracts files' DAC file permissions, SELinux labels and Linux capabilities. Then, it parses the system's init scripts and simulates commands that affect the filesystem (e.g., `mkdir`, `chmod`, etc.) as well as `service` commands which execute service binaries. Performing boot emulation is required to create files in the `/sys`, `/dev` and `/data` directories, which would not be present in a static firmware image.

After the boot process is emulated, BIGMAC begins the Backing File Recovery step, where it assigns the appropriate SELinux file types and domains to all files in the extracted filesystem. This is done by decompiling the extracted binary SEPolicy file to a multi-edge directional graph via the Access Vector rules (AVrules). Afterwards, it correlates policy types to actual files on the initialized filesystem. File objects are straightforward to correlate since their SELinux policy types are captured in the extraction step. For process subjects, Type Enforcement rules related to process transitions are inverted and processed allowing for the correlation of subject types and their executable binary backing files.

Using the full set of subject nodes, BIGMAC constructs a dataflow graph which simplifies the SELinux policy's access vectors into a read/write model. The dataflow graph captures all data flows allowed by AVRules for all subjects and objects by considering vectors that imply a read or a write. In the dataflow graph, objects can be one of two types: file objects and Inter-Process Communication (IPC) objects. As discussed in the previous step, file objects can contain multiple backing files, each with its own MAC/DAC/CAP metadata. On the other hand, IPC objects typically do not have any backing files and are tagged with the underlying AVClass. For instance, all classes that derive from the `socket` class are tagged as IPC objects.

The recovered subject nodes are also used in the Process Inflation step. For each subject node, BIGMAC attempts to match the subject node to a service definition by comparing the subject's backing file with the binary file in the service definition. If the service is enabled and is not a one-shot (transient) process, the service's defined security options are assigned to a new process. This process is then inserted into a concrete process tree.

The final step in the process is the Attack Graph Instantiation. In this step, all file objects within the dataflow graph are expanded, such that each file corresponds to one node in the graph, encompassing all of this file's MAC/DAC/CAP attributes. All of the edges to and from the original file object are duplicated for each individual node. This expanded graph is overlaid onto the concrete process tree, whereby, for each process in the process tree, all in- and out-edges in the corresponding subject in the dataflow graph are copied to the process tree. In the resultant graph, concrete processes have concrete edges to all the objects they can read from or write to. BIGMAC

TABLE 4. ACCESSIBLE SOCKETS DETECTED BY THE TESTING APP OR SAUSAGE ON EACH OF THE THREE TEST DEVICES. ✓ AND ✗ INDICATE WHETHER THE SOCKET WAS DETECTED BY THE CORRESPONDING TOOL.

Socket	Testing App	SAUSAGE
Motorola 7.1.1 NPIS26.48-43-2		
dnsproxyd	✓	✓
fwmarkd	✓	✓
logd	✓	✓
logdr	✓	✓
logdw	✓	✓
perfd	✓	✓
Samsung 7.0.0 NRD90M G920FXXU5EQJ1		
dnsproxyd	✓	✓
fwmarkd	✓	✓
logd	✓	✓
logdr	✓	✓
logdw	✓	✓
/dev/socket/jack/set.priority	✗	✓
@dev/socket/jack/set.priority	✗	✓
Samsung 8.0.0 R16NW G930UUES4CRH2		
dnsproxyd	✓	✓
napproxyd	✓	✓
fwmarkd	✓	✓
logd	✓	✓
logdr	✓	✓
logdw	✓	✓
pdx/system/vr/display/client	✗	✓
pdx/system/vr/display/manager	✗	✓
pdx/system/vr/display/vsync	✗	✓
/dev/socket/jack/set.priority	✗	✓
@dev/socket/jack/set.priority	✗	✓

then uses this graph to generate Prolog facts that can be used for dataflow paths in that graph.

2. Ground Truth Evaluation Results

Table 4 shows all accessible sockets detected by either SAUSAGE or the testing app we used in the ground truth evaluation, as well as the version information of the Android devices tested. We detect more sockets than the ground truth testing app as our static analysis approach allows the detection of inactive sockets that could be created under certain conditions or configurations. The ABSTRACT socket @dev/socket/jack/set.priority was detected twice by SAUSAGE, once as a false-positive FILESYSTEM socket and the second time as a true-positive ABSTRACT socket. This false positive result is due to the peculiar construction of the socket address itself in the *apaservice* binary. The address string is first set to "/dev/socket/jack/set.priority." Afterwards, the '/' character at the 0th index is replaced by a null byte. This effectively changes the address from a FILESYSTEM to an ABSTRACT socket address. However, due to the imprecision of the static analysis, both addresses are reported by SAUSAGE.