information

Article

Unix Domain Sockets Applied in Android Malware Should Not Be Ignored

Xu Jiang *, Dejun Mu and Huixiang Zhang

School of Automation, Northwestern Polytechnical University, Xi'an 710072, China; mudejun@nwpu.edu.cn (D.M.); zhanghuixiang@nwpu.edu.cn (H.Z.)
* Correspondence: jiangxu@mail.nwpu.edu.cn

Received: 8 February 2018; Accepted: 2 March 2018; Published: 4 March 2018

Abstract: Increasingly, malicious Android apps use various methods to steal private user data without their knowledge. Detecting the leakage of private data is the focus of mobile information security. An initial investigation found that none of the existing security analysis systems can track the flow of information through Unix domain sockets to detect the leakage of private data through such sockets, which can result in zero-day exploits in the information security field. In this paper, we conduct the first systematic study on Unix domain sockets as applied in Android apps. Then, we identify scenarios in which such apps can leak private data through Unix domain sockets, which the existing dynamic taint analysis systems do not catch. Based on these insights, we propose and implement JDroid, a taint analysis system that can track information flows through Unix domain sockets effectively to detect such privacy leaks.

Keywords: Android; information flows; Unix domain sockets; private data; malware

1. Introduction

In the second quarter of 2017, Android dominated the smartphone market, garnering an 86.1% share, according to a report by the International Data Corporation. Meanwhile, there are significant numbers of Android apps that enrich Android features. As mobile devices have become integrated into daily life, mobile devices collect increasing amounts of private data. Unfortunately, the Android operating system has been the target of increasing attacks by third-party apps [1–3], forming a widespread, serious challenge because apps also increasingly attempt to steal private data (e.g., IMEI, and location) and send them to remote servers. This has become especially true in the era of big data [4,5]. Hackers use several different methods to steal private data while remaining undetected by existing analysis systems, including inter-process communication (IPC) [6–10].

Since Android is based on a tailored Linux environment, it inherits a subset of the traditional Linux IPCs that differ from Android IPCs [11–14]. Among Linux IPCs implemented within Android, Unix domain sockets are the only one apps can easily make use of. Although Google encourages Android developers to use Android IPCs, some still use Unix domain sockets, known as local sockets [15]. This practice occurs not only because using UNIX domain sockets for IPC is more efficient but also because Android IPCs are unsuitable for communication between the Java language in which most apps are written and native processes/threads [16]. Both the Android software development kit (SDK) and the Android native development kit (NDK) [17] provide APIs for Unix domain sockets. To the best of our knowledge, how malicious apps exploit Unix domain sockets has not yet been systematically studied. After analyzed 2600 apps including 1500 normal apps and 1100 malicious apps, 315 (21%) normal apps and 209 (19%) malicious apps have Unix domain socket related APIs or system calls in code. In addition, APIs for Unix domain sockets can be used in different versions of Android operating systems including the latest version (i.e., Android 8.0) according to the documentation on the official developer website, which is verified through the implement of malware stealing private data based on Unix domain sockets [18]. What is more important, the existing taint analysis systems are unable to detect such leaks.

Motivated by these insights, in this study, we conduct a systematic study on information flows through Unix domain sockets and propose and implement JDroid, an efficient dynamic taint analysis system that tracks information flows through Unix domain sockets. Because the tracking process involves taint propagation at the Java level, the native level and between both through JNI, JDroid reuses some existing modules from TaintDroid [19] and NDroid [20].

To make JDroid effective and efficient, we handle several challenging issues, such as the various approaches to the different types of Unix domain sockets, the differences between Android and Linux, etc. An evaluation using sample apps employs Unix domain sockets to transmit private data and circumvent detection by existing analysis systems and demonstrates the effectiveness of JDroid in detecting private data leakage through Unix domain sockets. We further evaluate and report JDroid's performance.

The rest of this paper is organized as follows. Section 2 introduces the background, related work and describes undetected information leakage scenarios through Unix domain sockets. We detail the design, implementation, and evaluation of JDroid in Sections 3–5. Finally, we report the limitations of JDroid and we conclude the paper in Section 6.

2. Background

2.1. Android App Overview

In an Android system, apps are commonly written in the Java language and compiled into Java bytecode, which is then translated to Dalvik bytecode and stored in .dex and .odex files that execute on the proprietary register-based Dalvik virtual machine (DVM) [21]. Apps may contain both Java and native components; these native components are simply shared libraries loaded dynamically at runtime [22]. The interaction between Java components and native components is well-defined by the Java Native Interface (JNI) specification and supported by the NDK [23]. The lowest level of Android architecture is the customized Linux kernel, which provides the basic architectural model for process scheduling, resource handling, memory management, networking, etc. Consequently, Linux mechanism applied in Android system is worth researching seriously [24].

2.2. Unix Domain Sockets

A Unix domain socket is a data communications endpoint for exchanging data between processes executing on the same host operating system, and such sockets are a standard component of POSIX operating systems [25,26]. The APIs for Unix domain sockets are similar to those of Internet sockets; however, rather than using an underlying network protocol, all communication occurs entirely within the operating system kernel. Traditionally, there are three types of Unix domain socket address namespaces: FILESYSTEM, RESERVED, and ABSTRACT. An address in the FILESYSTEM namespace is associated with a file on the filesystem. RESERVED is in essence a sub-namespace of FILESYSTEM, while ABSTRACT is completely independent of the FILESYSTEM. The protocol family in the Unix domain is AF_UNIX/LOCAL [27].

The socket type specifies the communication semantics. SOCK_STREAM type sockets are full-duplex byte streams that must be in a connected state before any data may be sent or received through it [28]. The diagram in Figure 1 shows the complete client/server interaction [29].

In addition, Unix domain sockets support both unordered and unreliable datagram transmissions (i.e., SOCK_DGRAM) and ordered and reliable datagram transmission (i.e., SOCK_SEQPACKET, which is similar to SOCK_STREAM). As the diagram in Figure 2 shows, there is no flow control between the server and the client [30]. Each datagram message carries its destination address, its return address and a certain amount of data. Compared with SOCK_STREAM type sockets, the server need not create a socket for listening via the "listen" method and then call the "accept" method to wait for a connection.

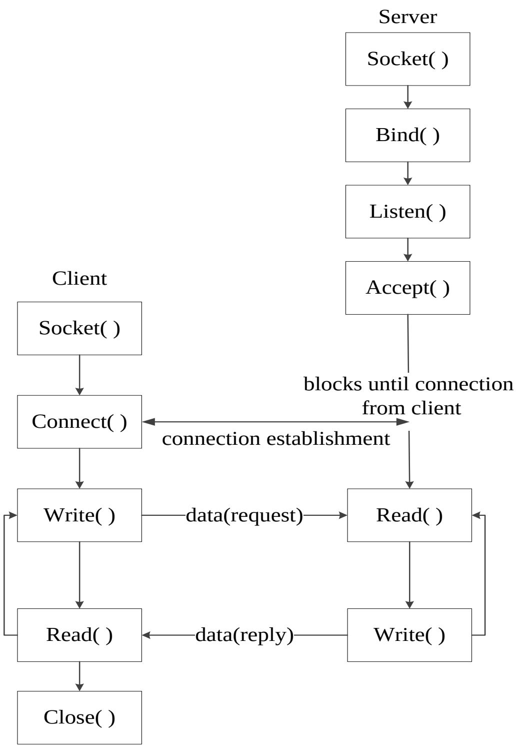


Figure 1. Connection-based sockets interaction.

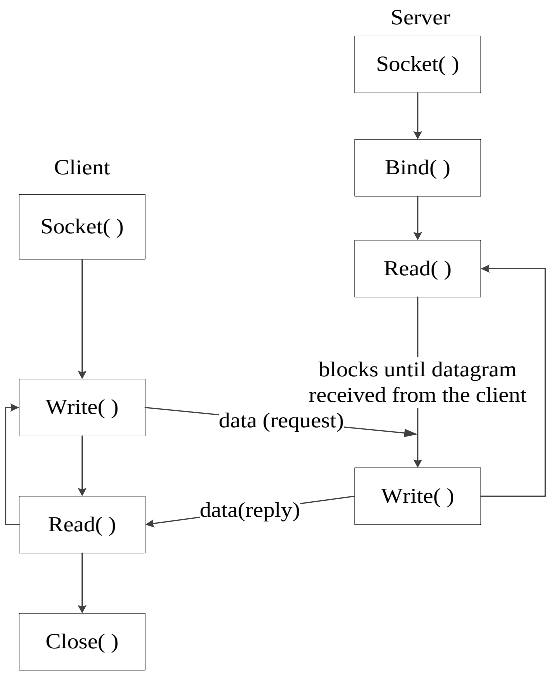


Figure 2. Datagram-oriented sockets interaction.

2.3. Related Work

Currently, the state-of-the-art method for detecting the leakage of private data is called "dynamic taint analysis" and is typically used in information security to enforce information flow policies to preserve data confidentiality and integrity. Tracking information flows allows users to know how a program processes private data [31,32]. TaintDroid is a prominent representative of applications that track information flows in Android dynamic taint analysis systems, with some 2312 citations at the time of this writing. Many existing analysis systems such as Droidbox [33] and AppFence [34] have developed new functionality based on TaintDroid.

By modifying the Android application framework and the DVM, TaintDroid stores a 32-bit bitvector with each variable to encode a taint tag, supporting 32 different taint markers. As with local method variables and arguments, TaintDroid allocates taint tag storage by doubling the size of the stack frame allocation, and taint tags are stored adjacent to class fields and arrays inside the VM interpreter's data structures. TaintDroid propagates the taint tags when the app is running, and monitors whether outgoing data has a taint tag in the Java layer. However, TaintDroid only loads native libraries from the firmware: it does not apply to those included in third-party apps.

Currently, hackers increasingly use native code in their malicious apps to hide the program logic [35–37]. Thus, researchers have begun to pay more attention to the security of such third-party native libraries. Some systems use tools such as ptrace [38], strace [39], and ltrace [40] to collect the system-call sequences made by these libraries and use that information to analyse malicious app behavior. CopperDroid [41] collects and analyses system calls acquired by instrumenting QEMU and app behaviors related to Binder. DroidScope [42] tracks information flows by reconstructing both the OS-level and Java-level semantics simultaneously and seamlessly, although it is less efficient.

NDroid tracks information flows across the boundary between the Java and native layers by instrumenting important JNI-related methods, and it monitors native code by processing each ARM/Thumb instruction. To work seamlessly with TaintDroid, NDroid reuses the modules modified by TaintDroid, and taints added by NDroid follow TaintDroid's format in the DVM. NDroid leverages shadow registers and memory to save the taints during native-layer execution and sets the taints in the DVM stack so it can refer to them when the taints are propagated to the Java layer.

Unfortunately, to the best of our knowledge, none of the existing dynamic taint-tracking approaches consider information leaks through Unix domain sockets.

2.4. Threat Model and Assumptions

Android apps using APIs for Unix domain sockets need only request Internet permission [43], which is so commonly used that users are not wary about granting it. However, once the app has been granted Internet permission, the APIs for Unix domain sockets can be executed in either Java or native code.

In this section, we analyze the scenarios in which private data can be transmitted through Unix domain sockets and then leaked and explain why such leakage cannot be detected by existing systems. We use TaintDroid and NDroid as references, because they are both advanced dynamic taint analysis systems and open source.

An information flow from a source to a sink is the main requirement for leaking private data. We consider the source to be the APIs that can acquire private data and the sink to be any APIs that can send private data out of the Android system (usually the network interface). Dynamic taint analysis systems track how labeled data impact other data in ways that might leak private data [44]. Private data is first identified at the source, where a taint tag indicates the data type assigned. Later, the data will be checked when it gets sent to the sink. Thus, eliminating the taint tags attached by existing systems has raised concerns within the hacker community. To better understand the threat model, we define the client and the server as the sending and receiving ends, respectively, and group the apps that employ Unix domain sockets to transmit private data into three cases, depending on the client and server locations in the Android architecture, as shown in Table 1.

Table 1. The client/server combinations in information flows through Unix domain sockets.

		Client	
		Java	Native
Server	Java	Case 1	Case 3
	Native	Case 3	Case 2

Case 1. As shown in Figure 3a, in this case, the client and the server are both located in the Java layer. First, private data is transmitted from the client to the server, which is acquired by invoking the source. Next, the server processes it and then leaks it through the sink. It is noteworthy that, in this case, the server and the client are created solely by Android APIs using specified names confined to the Linux abstract namespace. TaintDroid and NDroid cannot detect such leakage, because they do not consider taint propagation through Unix domain sockets in the Java layer.

Case 2. As shown in Figure 3b, in this case, the client and the server are both located in the native layer. The app invokes the source to fetch private data, which is then transmitted to the client in the native layer through JNI. The client processes the data and transmits it to the server through Unix domain sockets. Finally, native code can leak the data directly by calling the POSIX socket API.

TaintDroid cannot detect such leaks because it loads only the app, not the third-party native library. Moreover, it does not consider local sockets. NDroid misses such leaks because it cannot track information flow through Unix domain sockets; consequently it receives the information from memory without corresponding taint tags. Most existing analysis systems have the common vulnerability that they ignore POSIX socket APIs as the sink.

Case 3. As shown in Figure 3c, in this case, the client and the server are located in different layers: the direction of information flow could be from the Java layer to the native layer, or vice versa. In the Java layer, the client sends private data to the server in the native layer through Unix domain sockets. Then, Java code fetches private data from the native layer through JNI and leaks it through a Java sink or native code processes, which then leak the data directly through POSIX socket APIs. TaintDroid and NDroid cannot detect such leakage for the same reasons as in Case 2.

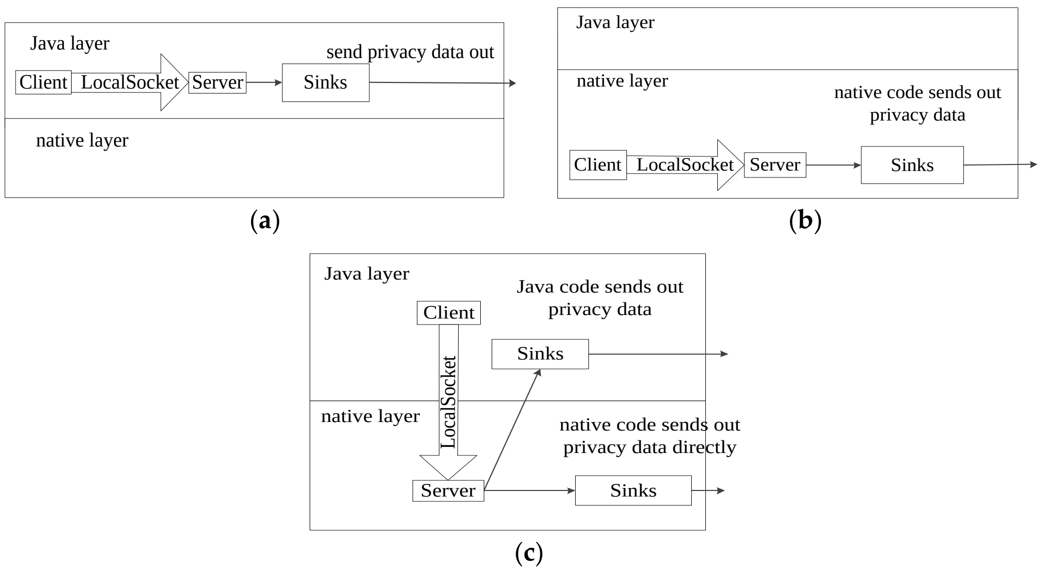


Figure 3. Examples of private data leakage through Unix domain sockets. (**a**) Case 1; (**b**) Case 2; (**c**) Case 3.

3. Design and Implement

Since JDroid must handle the taint propagation in the DVM and across the Java and native layers through JNI, we re-use some modules from TaintDroid and NDroid. Figure 4 illustrates the JDroid architecture. The general goal of JDroid is to track information flows through Unix domain sockets and detect private data leakage.

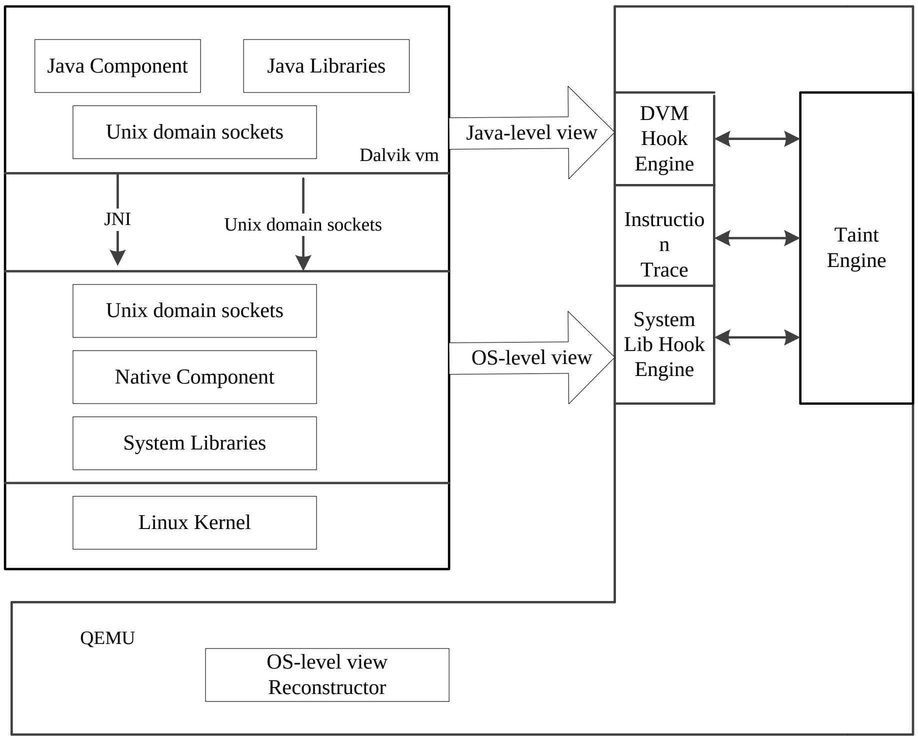


Figure 4. JDroid overview.

3.1. Taint Propagation

Developers can use both Java code and native code to implement communications through Unix domain sockets. For JDroid, a challenging issue is how to correctly ensure taint propagation during the process of private data transmission.

To tackle this issue, JDroid creates a structure called Nodethat records the pathnames of sockets and the corresponding taint tags, and uses a list to store the Node structure. As shown in Figure 5, the Node structure includes taint, the taint tag for sending data; a srcname, which is the pathname of the sending end; and a dstname, the pathname of the receiving end.

```
typedef struct Node{
                    int taint;
                    char srcname[256];
                    char dstname[256];
                    struct node *next;
                                        }Node
```

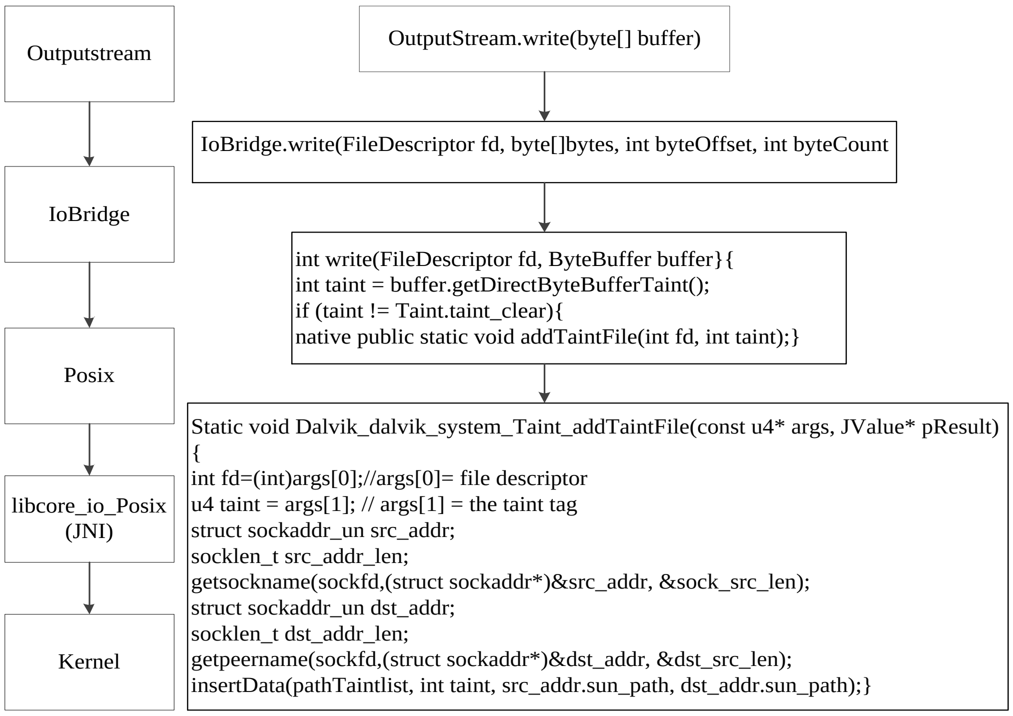
Figure 5. The "Node" structure.

3.2. The Handle inthe Sending End

Prior to sending, each API related to sending data in both the Java and native libraries is tasked with creating a Node.

Connection-based sockets. In the first case, the client is located in the Java layer. Since TaintDroid uses message-level taint tracking that represents the upper bound of the taint tag assigned to variables contained in the message, JDroid also adopts message-level taint tracking and creates a Node to record the relevant information.

Using the Java API "write" as an example, Figure 6 shows how the functions are called and how to model its taint propagation operation. First, the instrumented code in "libcore.io.Posix.write" obtains taint and invokes the native method "addTaintFile" to create the Node and insert the entry in the Node list. Note that the ARM/Thumb procedure call standard defines that the first four parameters are passed in R0 to R3, while the remaining parameters are pushed onto stack, and the return value is placed in R0.



```
OutputStream.write(byte[] buffer)
```

```
IoBridge.write(FileDescriptor fd, byte[]bytes, int byteOffset, int byteCount
```

```
int write(FileDescriptor fd, ByteBuffer buffer){
int taint = buffer.getDirectByteBufferTaint();
if (taint != Taint.taint_clear){
native public static void addTaintFile(int fd, int taint);}
```

```
Static void Dalvik_dalvik_system_Taint_addTaintFile(const u4* args, JValue* pResult)
{
int fd=(int)args[0];//args[0]= file descriptor
u4 taint = args[1]; // args[1] = the taint tag
struct sockaddr_un src_addr;
socklen_t src_addr_len;
getsockname(sockfd,(struct sockaddr*)&src_addr, &sock_src_len);
struct sockaddr_un dst_addr;
socklen_t dst_addr_len;
getpeername(sockfd,(struct sockaddr*)&dst_addr, &dst_src_len);
insertData(pathTaintlist, int taint, src_addr.sun_path, dst_addr.sun_path);}
```

Figure 6. The "write" propagation operation.

In the second case, the client is located in native layer. JDroid adopts NDroid's method to track the propagation of private data through JNI and monitor native code using shadow memory and registers so that the corresponding taint tag always follows the private data. Consequently, JDroid can retrieve a taint tag from memory in the native layer, check whether the data being sent includes a taint tag and create an appropriate entry in the Node list. The rest of the process is identical to that for the Java layer.

Datagram-oriented sockets. Datagram-oriented sockets use datagram communications between one server and several clients. A datagram-oriented socket provides a symmetric data exchange interface without requiring a connection to be established. The sending behavior is implemented by the "sendto" and "sendmsg" methods, as shown in Figure 7.

```
ssize_t sendto(intsockfd, const void *buf, size_tlen, int flags, conststructsockaddr *dest_addr, socklen_taddrlen)

ssize_t sendmsg(intsockfd, conststructmsghdr *msg, int flags)
```

Figure 7. The "sendto" and "sendmsg" methods.

Although Google provides Android APIs (e.g., LocalServerSocket, LocalSocket) for developers to use with Unix domain sockets, they are not available for datagram-oriented sockets except when the client invokes the method "connect" to establish the connection relation. If using "connect" to connect the server, the client can use the method "send" to send the data. Following that path, the handle is the same as in connection-based sockets.

Briefly, the SOCK_DGRAM type of Unix domain socket that does not invoke "connect" must be implemented in native code. Therefore, JDroid hooked the "sendto" and "sendmsg" methods. For "sendto", JDroid parses the second parameter (i.e., buf) to check whether the data being sent has a taint tag and the fifth parameter (i.e., dest_addr) to obtain the dstname. For "sendmsg", it uses the msghdr structure to minimize the number of directly supplied arguments, as shown in Figure 8.

```
struct msghdr {
                  Void *msg_name; //optional address
                  socklen_t msg_namelen; //size of address
                  struct iovec *msg_iov;   // scatter
                  size_t msg_iovlen; // elements in msg_iov
                  void   *msg_control; //ancillary data
                  size_t msg_controllen; //ancillary data buffer len
                  int   msg_flags; // flags on received message };
```

Figure 8. The "msghdr" struct.

The data being sent is pointed to by the elements of themsg.msg_iov array, and dstname is pointed to by msg.msg_name. Further, the Srcname can also be obtained by invoking "getsockname". Based on these data items, JDroid creates a new Node.

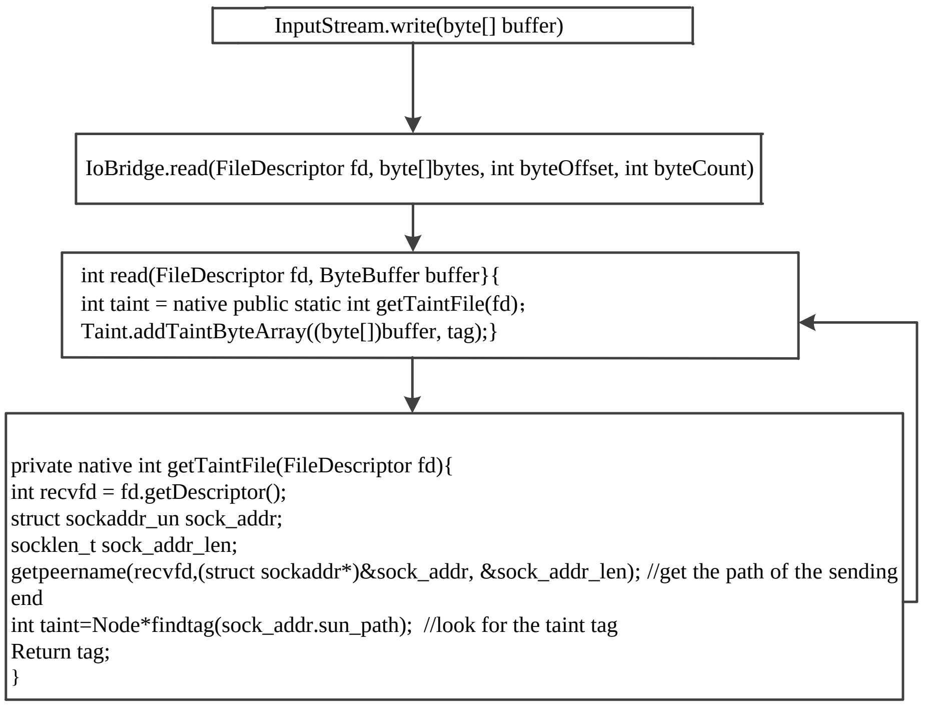
3.3. The Handle in the Receiving End

JDroid initializes the taint tag for tracking an information flow entering the server using two steps. The first step determines which Node should get the taint tag based on the peer's pathname (i.e., srcname) and its own pathname (i.e., dstname). The second step attaches the taint to the received data, which can used to continue tracking taint propagation until it reaches a sink.

Two approaches are required to obtain the peer's pathname because of the differences between connection-based sockets and datagram-oriented sockets.

Connection-based sockets. For communications between connection-based sockets (whether in the Java or native layer), the server looks for the taint tag (i.e., taint) of the corresponding Node based on the methods that JDroid invokes: "getpeername" to obtain the client's pathname and "getsockname" to get itself pathname. If the client does not bind the specified pathname, the return value of "getpeername" may be NULL, differing from Linux, which returns the pathname allocated by kernel. Therefore, JDroid confirms that the corresponding Node is based not only on the pathnames of the client and the server, but also the sequence of a Node in the list based on its creation time. Then, JDroid associates the taint with the received data.

Taking the Java API "read" as an example, JDroid first obtains the taint tag based on the Node, which goes back to the Java layer and was attached to the received data. Figure 9 shows how to model the taint propagation operation.


InputStream.write(byte[] buffer)
IoBridge.read(FileDescriptor fd, byte[]bytes, int byteOffset, int byteCount)

```
int read(FileDescriptor fd, ByteBuffer buffer}{
int taint = native public static int getTaintFile(fd);
Taint.addTaintByteArray((byte[])buffer, tag);}
```

```
private native int getTaintFile(FileDescriptor fd){
int recvfd = fd.getDescriptor();
struct sockaddr_un sock_addr;
socklen_t sock_addr_len;
getpeername(recvfd,(struct sockaddr*)&sock_addr, &sock_addr_len); //get the path of the sending
end
int taint=Node*findtag(sock_addr.sun_path);  //look for the taint tag
Return tag;
}
```

Figure 9. "read" taint operation.

Datagram-oriented sockets. If the client used "connect", the server can employ "recv" to receive the data, which is the same as when using connection-based sockets. In addition, for datagram-oriented sockets, the server calls "recvfrom" or "recvmsg" to initialize the receive behavior on a socket, as shown in Figure 10.

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Figure 10. The "recvfrom" and "recvmsg" methods.

Since the server cannot exploit "getpeername" to get the client's pathname, JDroid parses the function arguments to obtain the srcname and uses that to look for the Node. Then, JDroid associates the taint with the received buffer using shadow memory and registers. Only in this way can JDroid continue to track the data.

For "recvfrom", src_addr represents the srcname, while for "recvmsg", msg.msg_name specifies the srcname. If the src_addrormsg.msg_name are NULL, the sockets are identical to connected sockets and taint propagation procedure is the same as with a connected socket. Taking the "recvfrom" (without a NULL src_addr argument) for example, the handler is shown in Figure 11.

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);
{
    int taint=Node *findtag(src_addr.sun_path);
    taint_pair newTaint (buf, taint);
    taintMap.insert(newTaint);
 }
```

Figure 11. Taint operation for the "recvfrom" method.

4. Experiments

In our experiments, we first used a simple tool, Monkeyrunner, to generate random input to test 2300 apps. However, because Monkeyrunner might not trigger an app's malicious behavior, we found that the Echo.apk leaks private data by following Case 3. From a positive aspect, our results demonstrate that malware developers do not pay much attention to exploiting Unix domain socket. Moreover, we used two proof-of-concept (PoC) apps (one each for Cases 1 and 2) to further evaluate JDroid's ability to track information flows through Unix domain sockets. Finally, we used CaffeineMark to evaluate JDroid's overhead.

The experiments were performed in a virtual machine with 4 GB memory running Ubuntu. The host was an Intel(R) Core(TM) i7 running @ 2.6 GHZ with 16 GB of RAM.

4.1. PoC of Case 1 in Information Leakage

In this PoC, the app uses Java code to accomplish the entire transmission process based on the SOCK_STREAM socket type. The Java code first obtains the device's IMEI with the taint tag (i.e., 0x400) and transmits it to the server from the client. When the client begins to implement the sending behavior, JDroid creates an entry in the Node list that records the taint tag and the client and server pathnames. Before receiving the data, the server invokes "getpeername" and "getsockname" through JNI which then looks for the corresponding Nodeas discussed earlier. When the Node is found, JDroid invokes "Taint.addTaintByteArray" to add the taint to the received data. The main functions in the information flow identified by JDroid are shown in Figure 12. Finally, the received data is sent to the specified server. JDroid not only adds a taint tag to the received data but also tracks the information flow until it reaches the sink.

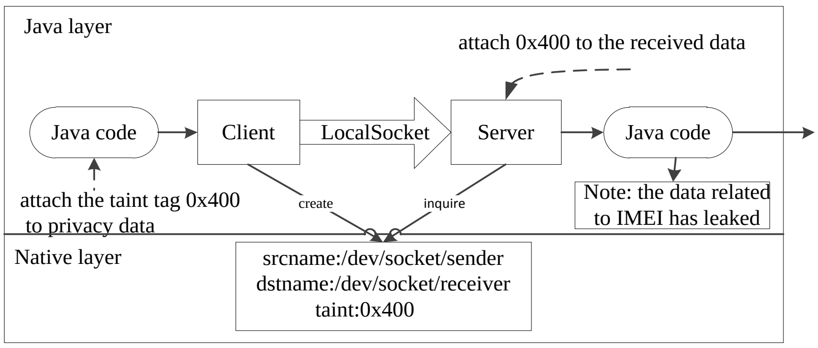

Java layer
attach 0x400 to the received data
Java code
Client
LocalSocket
Server
Java code
attach the taint tag 0x400 to privacy data
create
inquire
Note: the data related to IMEI has leaked
Native layer
srcname:/dev/socket/sender
dstname:/dev/socket/receiver
taint:0x400

Figure 12. PoC for Case 1.

4.2. PoC of Case 2 in Information Leakage

In this PoC, the client and the server are both located in the native layer and based on the SOCK_DGRAM socket type. Similar to the PoC for Case 1, this PoC first fetches private data related to the ICCID with the taint tag 0x1000, which is then transmitted to the native layer through JNI. JDroid traces the information flow through JNI and locates the memory at 0x4a98c9d4 associated with the taint tag 0x1000. Then, the client transmits the data based on the memory at 0x4a98c9d4 to the server using the native library call "sendto". By hooking "sendto", JDroid constructs a Node by parsing the "sendto" parameters (i.e., buf and dest_addr) to populate the taint and dstname and invokes "getsockname" to obtain the srcname. After the server calls "recvfrom" to receive the data, JDroid retrieves the srcname by parsing the fifth parameter of "recvfrom" (i.e., src_addr), and looks for the Node corresponding to srcname and its own pathname (i.e., dstname) and associates thetaint"0x1000" with the received buffer memory at 0x4a98c93c. Finally, that data reaches "sendto". Since "sendto" is a sink, JDroid notices that the ICCID-related data has leaked after checking the parameters, as shown in Figure 13.

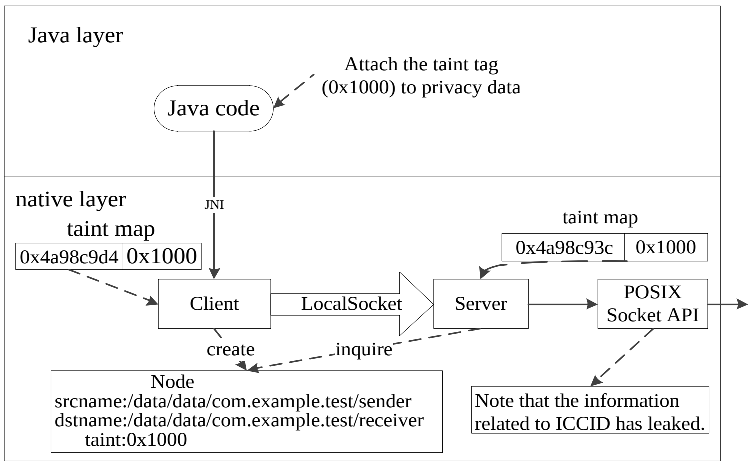

Java layer
Attach the taint tag (0x1000) to privacy data
Java code
JNI
native layer
taint map
0x4a98c9d4 0x1000
taint map
0x4a98c93c 0x1000
Client
LocalSocket
Server
POSIX Socket API
create
inquire
Node
srcname:/data/data/com.example.test/sender
dstname:/data/data/com.example.test/receiver
taint:0x1000
Note that the information related to ICCID has leaked.

Figure 13. PoC for Case 2.

4.3. Echo

JDroid discovered that Echo may send private data related to IMEI to the specified server. The client and the server both use the SOCK_STREAM socket type and the FILESYSTEM namespace. Figure 14, which is an example of Case 3, shows the major functions in the information flow identified by JDroid. First, Java code invokes an Android API (i.e., getDeviceId) to obtain the IMEI that will be sent by the client in the Java layer. This type of parameter is a byte array and its taint tag is “0x400”. Before sending, the instrumented code invokes “getsockname” and “getpeername” to obtain the srcname and dstname respectively. Based on this information, JDroid creates a new Node for the server inquiry.

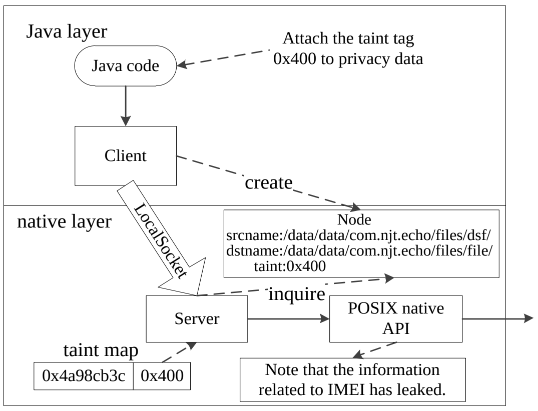

Java layer
Attach the taint tag 0x400 to privacy data
Java code
Client
create
native layer
LocalSocket
Node
srcname:/data/data/com.njt.echo/files/dsf/
dstname:/data/data/com.njt.echo/files/file/
taint:0x400
inquire
Server
POSIX native API
taint map
0x4a98cb3c 0x400
Note that the information related to IMEI has leaked.

Figure 14. Echo.apk.

Before to the server in the native layer begins receiving the data, JDroid looks up the corresponding Node (by searching for srcname and its own pathname (i.e.,dstname). Next, JDroid initializes the taint tag in the native layer based on the found Node. More precisely, it associates the taint tag “0x400” with the memory location at “0x4a98cb3c” and a shadow register. When the data reaches the sink (i.e., “sendto”), JDroid warns that data concerning the IMEI has leaked.

4.4. Compare with Other Analysis Systems

When tested with the two PoC apps and Echo, TaintDroid and NDroid, which are advanced dynamic taint analysis systems currently in use, were unable to detect information leaks through Unix domain sockets.

VirusTotal is a free service that includes 55 of the latest popular analysis systems for analyzing suspicious files and URLs and facilitates quick detection of viruses, worms, trojans, and all types of malware. We employed VirusTotal to analyze our sample apps. None of the included analysis systems detects the malicious behavior, including Qihoo-360, McAfee, and so on. Although the analysis methods of the 55 systems are varied and some include dynamic taint analysis, at a minimum, this result illustrates that we should emphasize this issue—and indicates that hackers may not yet have realized this vulnerability in the existing analysis systems.

Aijiami is another popular service that provides risk assessment for Android apps. We used Aijiami to evaluate the risk of the three sample apps. The results shows that an uncertain risk may exist based on the requested permissions, which are sufficient for leaking private data (i.e., android.permission.INTERNET and android.permission.READ_PHONE_STATE). However, the score of the three sample apps were all above 60, which means that they are considered normal apps.

5. Performance

CaffeineMark is dedicated to measuring the speed of running Java programs and computing the instructions executed per second and uses an internal scoring system for relative comparisons. We used CaffeineMark to evaluate the performance of applications on unmodified Android, and when using TaintDroid and JDroid. Figure 15 shows the execution time results of a Java microbenchmark.

For the sieve microbenchmark, which is how the classic sieve of Eratosthenes finds prime numbers, there is no obvious differences among unmodified Android, TaintDroid, and JDroid. The loop scores achieved by TaintDroid and JDroid are significantly less than that of unmodified Android, which indicates the testing score uses sorting and sequence generation to measure compiler optimization of loops. This result most likely occurs because TaintDroid and JDroid hook the JNI-related methods, which are invoked frequently; thus, they increase the overhead considerably. For the logic microbenchmark, testing the speed with which the virtual machine executes decision-making instructions, JDroid's score was markedly below those of unmodified Android and TaintDroid, because JDroid must perform considerable work when data is transmitted across the different layers. Although JDroid's other indicators are also somewhat lower than the other two systems because of its monitoring of the native layer, this level of overhead is acceptable considering JDroid's improved tracking of the information flow according to the overall microbenchmark that indicates a cumulative score across individual benchmarks.

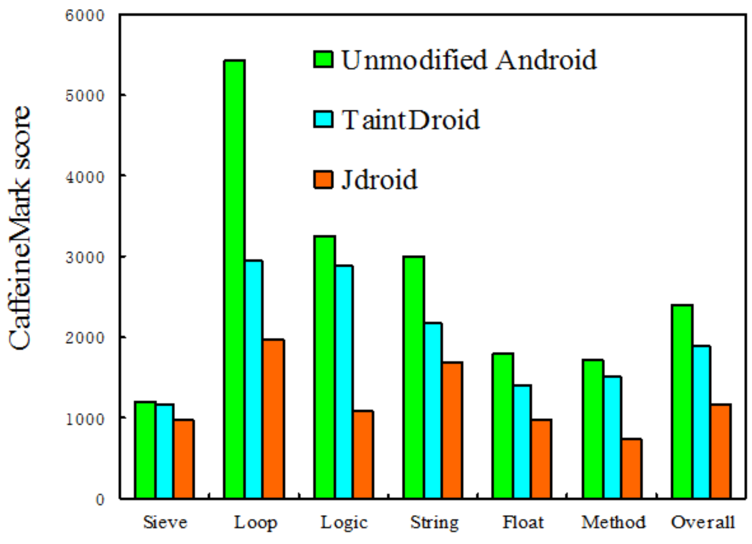


Figure 15. Microenchmark results.

6. Discussion and Conclusions

SinceTaintDroid works only on three Android versions, JDroid based on TaintDroid is also restricted to those same three versions. Thus, future work involves improving JDroid's compatibility, making it applicable to a larger variety of Android versions. In addition, most virtualization-based systems have a common feature: it is difficult to completely emulate the real hardware. For example, JDroid cannot monitor some important data sources (e.g., GPS). Hence, JDroid can generate fake information related to these important sources. Nevertheless, advanced malicious apps can exploit the differences between emulators and real machines to hide by avoiding the triggering of the malicious behavior.

Similar to TaintDroid and NDroid, JDroid does not track control flows. Therefore, there are some anti-taint methods in mobile device environments that can circumvent taint tracking [45]. However, if JDroid were to fully support control-flow tracking, it would suffer from even higher overhead. To ensure information flow control capability, JDroid's trusted computing base (TCB) is comprised of both the kernel and the managed runtime. Additionally, we realized the app with root privilege can modify trusted functions. Therefore, JDroid also incorporates the functions in RootGuard [46], which monitors system calls to detect the malicious behaviors. Furthermore, we will investigate some existing evadable methods to improve JDroid in future work.

In this work, we conducted a systematic study to track information flows through Unix domain sockets in Android apps. We identified a set of scenarios in which private data can be leaked that are not caught by existing dynamic taint analysis systems. Based on these insights, we proposed and implemented JDroid, an efficient dynamic taint analysis system for tracking information flows through Unix domain sockets by solving several challenging issues. The evaluation using sample apps demonstrates that JDroid can effectively track information flows through Unix domain sockets and detect private data leakage associated with the use of such sockets.

Acknowledgments: This work was supported in part by the National Natural Science Foundation of China (No. 61672433), the State Cryptography Foundation of China (MMJJ20170210), and the Science and Technology Foundation of Shenzhen China (201703063000517). The funds that we received cover the costs to publish in open access.

Author Contributions: Xu Jiang is the corresponding author of the paper. Xu Jiang is responsible for all aspects of the paper. Dejun Mu performed part of the experiment. Huixiang Zhang also performed part of the experiment. All authors read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Zhou, Y.; Wang, Z.; Zhou, W.; Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In Proceedings of the 19th Annual Network and Distributed System Security Symposium, San Diego, CA, USA, 7 February 2012; pp. 50–52.
2. Tam, K.; Feizollah, A.; Anuar, N.B.; Salleh, R.; Cavallaro, L. The Evolution of Android Malware and Android Analysis Techniques. *ACM Comput. Surv.* **2017**, *49*, 76. [CrossRef]
3. Mariconti, E.; Onwuzurike, L.; Andriotis, P.; De Cristofaro, E.; Ross, G.; Stringhini, G. MamaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In Proceedings of the NDSS '17: Network and Distributed Systems Security Symposium, San Diego, CA, USA, 26 February–1 March 2017.
4. Tan, J.; Drolia, U.; Martins, R.; Gandhi, R.; Narasimhan, P. Chips: Content-Based heuristics for improving photo privacy for smartphones. In Proceedings of the 7th ACM Conference on Security and Privacy in Wireless & Mobile Networks, London, UK, 23–25 July 2014; pp. 213–218.
5. Gordon, M.I.; Kim, D.; Perkins, J.; Rinard, M. Information-Flow Analysis of Android Applications in DroidSafe. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 8 February 2015.

6. Bugiel, S.; Davi, L.; Dmitrienko, A.; Fischer, T.; Sadeghi, A.R. *Xmandroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*; Technical Report; Technische Universität Darmstadt: Darmstadt, Germany, 2011; p. 4.
7. Bugiel, S.; Davi, L.; Dmitrienko, A.; Fischer, T.; Sadeghi, A.-R.; Shastry, B. Towards taming privilege-escalation attacks on android. In Proceedings of the 19th Network and Distributed System Security Symposium, San Diego, CA, USA, 7 February 2012; p. 19.
8. Dietz, M.; Shekhar, S.; Pisetsky, Y.; Shu, A.; Wallach, D.S. Quire: Lightweight provenance for smart phone operating systems. In Proceedings of the 20th USENIX Conference on Security, San Francisco, CA, USA, 8–12 August 2011; p. 23.
9. Felt, A.P.; Wang, H.J.; Moshchuk, A.; Hanna, S.; Chin, E. Permission re-delegation: Attacks and defenses. In Proceedings of the USENIX Security Symposium, San Francisco, CA, USA, 8–12 August 2011.
10. Fragkaki, E.; Bauer, L.; Jia, L.; Swasey, D. Modeling and enhancing androids permission system. In Proceedings of the 17th European Symposium on Research in Computer Security, Pisa, Italy, 10–12 September 2012; pp. 1–18.
11. Seo, J.; Kim, D.; Cho, D.; Shin, I.; Kim, T. FLEXDROID: Enforcing in-app privilege separation in Android. In Proceedings of the 23rd Network and Distributed System Security Symposium, San Diego, CA, USA, 21–24 February 2016; pp. 21–24.
12. Chin, E.; Felt, A.P.; Greenwood, K.; Wagner, D. Analyzing inter-application communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, MD, USA, 28 June–1 July 2011; pp. 239–252.
13. Davi, L.; Dmitrienko, A.; Sadeghi, A.R.; Winandy, M. Privilege escalation attacks on android. In Proceedings of the 2010 IEEE International Conference on Information Security, Boca Raton, FL, USA, 25–28 October 2010; pp. 346–360.
14. Grace, M.C.; Zhou, Y.; Wang, Z.; Jiang, X. Systematic detection of capability leaks in stock android smartphones. In Proceedings of the 19th Network and Distributed System Security Symposium, San Diego, CA, USA, 7 February 2012; p. 19.
15. Cinar, O.; Grant, A. *Pro Android C++ with the NDK*; Apress: New York, NY, USA, 2012.
16. Ghafari, M.; Gadient, P.; Nierstrasz, O. Security Smells in Android. In Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, Shanghai, China, 17–18 September 2017; pp. 121–130.
17. Li, Y.; Hao, Q.; Ma, X.; Zhang, P.; Fan, Y.; Jiang, J. Design and implementation of Android-based speech storage system. In Proceedings of the Information Technology. Networking, Electronic and Automation Control Conference, Chongqing, China, 20–22 May 2016; pp. 178–181.
18. Shao, Y.; Ott, J.; Jia, Y.J.; Qian, Z.; Mao, Z.M. The Misuse of Android Unix Domain Sockets and Security Implications. In Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 80–91.
19. Enck, W.; Gilbert, P.; Han, S.; Tendulkar, V.; Chun, B.G.; Cox, L.P.; Sheth, A.N. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In Proceedings of the ACM Transactions on Computer Systems, Vancouver, BC, Canada, 4–6 October 2010; Volume 22.
20. Qian, C.; Luo, X.; Shao, Y.; Chan, A.T. On tracking information flows through JNI in android applications. In Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, USA, 23–26 June 2014; pp. 180–191.
21. Backes, M.; Bugiel, S.; Gerling, S.; von Styp-Rekowsky, P. Android security framework: Extensible multi-layered access control on Android. In Proceedings of the Computer Security Applications Conference, New Orleans, LA, USA, 8–12 December 2014; ACM: New York, NY, USA, 2014; pp. 46–55.
22. Allix, K.; Bissyandé, T.F.; Klein, J.; Le Traon, Y. AndroZoo: Collecting Millions of Android Apps for the Research Community. In Proceedings of the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories, Austin, TX, USA, 14–15 May 2016; pp. 468–471.
23. Gu, Y.; Sun, K.; Su, P.; Li, Q.; Lu, Y.; Ying, L.; Feng, D. JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android. In Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, Denver, CO, USA, 26–29 June 2017; pp. 427–438.
24. Sawada, Y.; Arai, Y.; Ootsu, K.; Yokota, T.; Ohkawa, T. Performance of Android Cluster System Allowing Dynamic Node Reconfiguration. *Wirel. Pers. Commun. Int. J.* **2017**, *93*, 1067–1087. [CrossRef]

25. Demertzis, K.; Iliadis, L. Computational intelligence anti-malware framework for android OS. *Vietnam J. Comput. Sci.* **2017**, *4*, 1–15. [CrossRef]
26. Hofmann, M.; Rünger, G. Sustainability through flexibility: Building complex simulation programs for distributed computing systems. *Simul. Model. Pract. Theory* **2015**, *58*, 65–78. [CrossRef]
27. Fox, R.; Kasten, E.; Orji, K.; Bolen, C.; Maurice, C.; Venema, J. Real-time results without real-time systems. *IEEE Trans.Nucl. Sci.* **2015**, *51*, 571–575. [CrossRef]
28. Grace, M.; Zhou, Y.; Zhang, Q.; Zou, S.; Jiang, X. Riskranker: Scalable and accurate zero-day android malware detection. In Proceedings of the 10th International Conference on Mobile Systems, Applications and Services, Low Wood Bay, Lake District, UK, 25–29 June 2012; pp. 281–294.
29. Byun, S.S. TCP over scarce transmission opportunity in cognitive radio networks. *Comput. Netw.* **2016**, *103*, 101–114. [CrossRef]
30. Lin, C.C.; Li, H.; Zhou, X.; Wang, F. Screenmilker: How to Milk Your Android Screen for Secrets. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 24 February 2014.
31. Rafnsson, W. *Tracking Information Flows in Interactive and Object-Oriented Programs*; Chalmers University of Technology: Göteborg, Sweden, 2012.
32. Gries, S.; Hesenius, M.; Gruhn, V. Tracking Information Flow in Cyber-Physical Systems. In Proceedings of the IEEE International Conference on Distributed Computing Systems, Atlanta, GA, USA, 5–8 June 2017; pp. 2589–2590.
33. Lantz, P.; Desnos, A.; Yang, K. DroidBox: An Android Application Sandbox for Dynamic Analysis. Available online: https://code.google.com/p/droidbox (accessed on 3 March 2018).
34. Hornyack, P.; Han, S.; Jung, J.; Schechter, S.; Wetherall, D. These aren't the Droids you're looking for: Retrofitting android to protect data from imperious applications. In Proceedings of the 18th ACM Conference on Computer and Communication Security, Chicago, IL, USA, 17–21 October 2011; pp. 639–652.
35. Egele, M.; Scholte, T.; Kirda, E.; Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.* **2012**, *44*. [CrossRef]
36. Fratantonio, Y.; Bianchi, A.; Robertson, W.; Kirda, E.; Kruegel, C.; Vigna, G. Triggerscope: Towards detecting logic bombs in android applications. In Proceedings of the Security and Privacy Symposium, San Jose, CA, USA, 22–26 May 2016; pp. 377–396.
37. Zhou, Y.; Jiang, X. Dissecting android malware: Characterization and evolution. In Proceedings of the 33rd IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 20–23 May 2012; pp. 95–109.
38. Portokalidis, G.; Homburg, P.; Anagnostakis, K.; Bos, H. Paranoid android: Versatile protection for smartphones. In Proceedings of the 2010 Annual Computer Security Applications Conference, Austin, TX, USA, 6–10 December 2010; pp. 347–356.
39. Burguera, I.; Zurutuza, U.; Nadjm-Tehrani, S. Crowdroid: Behavior-Based malware detection system for android. In Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices, New York, NY, USA, 17 October 2011; pp. 15–26.
40. Spreitzenbarth, M.; Echtler, F.; Hoffmann, J. Mobilesandbox: Having a deeper look into android applications. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, New York, NY, USA, 18–22 March 2013; pp. 1808–1815.
41. Tam, K.; Khan, S.J.; Fattori, A.; Cavallaro, L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In Proceedings of the Network and Distributed System Security Symposium, San Diego, CA, USA, 8–11 February 2015.
42. Yan, L.K.; Yin, H. Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis. In Proceedings of the 21st USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012; pp. 569–584.
43. Johnson, R.; Wang, Z.; Gagnon, C.; Stavrou, A. Analysis of Android Applications' Permissions. In Proceedings of the IEEE Sixth International Conference on Software Security and Reliability Companion, Gaithersburg, MD, USA, 20–22 June 2012; pp. 45–46.

44. Ghandour, W.J.; Akkary, H.; Masri, W. The potential of using dynamic information flow analysis in data value prediction. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Vienna, Austria, 11–15 September 2010; pp. 431–442.
45. Sarwar, G.; Mehani, O.; Boreli, R.; Kaafar, M.A. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In Proceedings of the 10th International Conference on Security and Cryptography, Reykjavík, Iceland, 29–31 July 2013; pp. 461–468.
46. Shao, Y.; Luo, X.; Qian, C. RootGuard: Protecting Rooted Android Phones. *Computer* **2014**, *47*, 32–40. [CrossRef]