Cybersecurity

RESEARCH

Open Access

AppChainer: investigating the chainability among payloads in android applications

Xiaobo Xiang[1,2], Yue Jiang[1,2], Qingli Guo[1,2*], Xiu Zhang[1,2], Xiaorui Gong[1,2] and Baoxu Liu[1,2]

Abstract

Statistics show that more than 80 applications are installed on each android smartphone. Vulnerability research on Android applications is of critical importance. Recently, academic researchers mainly focus on single bug patterns, while few of them investigate the relations between multiple bugs. Industrial researchers proposed a series of logic exploit chains leveraging multiple logic bugs. However, there is no general model to evaluate the chaining abilities between bugs. This paper presents a formal model to elucidate the relations between multiple bugs in Android applications. To prove the effectiveness of the model, we design and implement a prototype system named AppChainer. AppChainer automatically identifies attack surfaces of Android applications and investigates whether the payloads entering these attack surfaces are "chainable". Experimental results on 2138 popular Android applications show that AppChainer is effective in identifying and chaining attacker-controllable payloads. It identifies 14467 chainable payloads and constructs 5458 chains both inside a single application and among various applications. The time cost and resource consumption of AppChainer are also acceptable. For each application, the average analysis time is 317 s, and the average memory consumed is 2368 MB. Compared with the most relevant work Jandroid, the experiment results on our custom DroidChainBench show that AppChainer outperforms Jandroid at the precision rate and performs equally with Jandroid at the recall rate.

Keywords Android security, Vulnerability exploit, Payload chain

Introduction

The variety and quantity of Android applications are fast growing nowadays. Statistics (buildfile 2022) show that more than 80 applications are installed on each smartphone on average, exposing numerous attack surfaces to malicious attackers. If application vulnerabilities are successfully exploited by an attacker, both the user privacy and system resources would be seriously threatened.

Existing academic research related to Android vulnerability mainly focuses on single bug patterns or the methodology for discovering and exploiting bugs. The well-researched bug patterns include permission-related bugs (Au et al. 2012; Bagheri et al. 2018, 2015; Demissie et al. 2020), attack families related to Intents (Groß et al. 2018; El-Zawawy et al. 2021; Gao et al. 2018a) or other IPC mechanisms (Elgharabawy et al. 2022), insecure deep links (Aldoseri and Oswald 2022), and dangerous file operations (Zhang et al. 2019). Methods such as fuzz testing (Yang et al. 2014; Ye et al. 2013; Choi et al. 2018), symbolic execution (Gao et al. 2018b; Luo et al. 2019), taint analysis (Min et al. 2019; Arzt et al. 2014), and machine learning (Garg and Baliyan 2020) are conducted for both vulnerability discovery and exploitation. However, few works investigate the relationship between bugs.

Recently, security researchers from the industry proposed a series of attacks that chain multiple logic bugs together. Such attacks are able to execute arbitrary code (Plaskett and Loureiro 2018; Geshev and Miller 2018),

*Correspondence:
Qingli Guo
guoqingli@iie.ac.cn
[1] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China

steal private files (f-secure Lab 2019), and break the application sandbox (Dawn Security Lab 2022). Despite the existence of effective exploit chains, experienced researchers widely regard the exploit chain building procedures as "case-by-case" tasks (Plaskett and Loureiro 2018). A formal model for explaining whether and why two bugs can be chained together is missing. To our best knowledge, Jandroid (f-secure Lab 2019) is the only tool developed for building the logic bug exploit chain towards the Android platform. Jandroid searches specific code patterns defined in a template and chains the identified code snippets together. However, the definition of its templates depends on expert experience and lacks universality. In addition, the data controllability of the identified code snippets is unknown because it does not track the information flow of the attacker-controllable inputs.

Motivated by these existing logic bug exploit chains and bug patterns, this paper proposes an exploit chain model to elucidate the "chainability" of multiple functionalities from the angle of payloads. The term *Payload* represents the attacker-controllable input entering an attack surface in Android applications. The term *Gadget* denotes a set of functional code snippets that can be triggered or controlled by a *Payload*. We model the input requirements and output abilities of each triggered gadget to measure the IO property of a payload. For two payloads ready to be received and processed by corresponding attack surfaces, if the output of one payload fits the input of another payload, then the two payloads can be chained together.

Practically, we design and implement a prototype system named AppChainer based on the chain model. AppChainer can automatically identify an application's potential attack surfaces, gather the gadgets triggerable by payloads entering these attack surfaces, and investigate the chainability among the identified payloads.

The effectiveness of the model and performance of AppChainer are evaluated on 2138 popular applications downloaded from Google Play. Each popular application in our test suite has more than one million installations. AppChainer identifies 14467 payloads composing 5458 chains both inside a single application and among various applications. The chained payloads can trigger more functionalities than single payloads and thus increase the attacker's attack abilities. The average analysis time and consumed memory for these applications are 317 s and 2368MB, respectively.

The precision and recall rates are evaluated on the DroidChainBench. Compared with the most relevant work Jandroid, the experiment results on our custom DroidChainBench show that AppChainer outperforms Jandroid at the precision rate (100%> 77.78%) and performs equally with Jandroid at the recall rate (87.5% = 87.5%).

To sum up, the contributions include:

(1) This paper proposes a model to formally explain the logic bug exploit chain from the angle of payloads and their IO properties.
(2) Based on the model, we design and implement a prototype system named AppChainer. It automatically identifies chainable payloads inside one or among various applications.
(3) Experimental results on a number of popular Android applications show that AppChainer is effective in finding chainable payloads and constructing exploit chains. The cost of time and memory are also acceptable.

Motivation

This work is initially inspired by the exploit chain (Geshev and Miller 2018) proposed in a mobile pwn2own competition. The chain glued 11 bugs or features across six applications together to launch remote code execution attack on a Samsung S8 device. In this section, we briefly introduct the workflow and key bugs involed in the motivating chain, then we summarize three key observations to prove the feasibility of conducting exploit chain research on the Android platform.

Motivating example

In the following text, we briefly retrospect the two "intent-proxy" bugs leveraged in the exploit chain (Geshev and Miller 2018).

Intent-proxy bugs, also named second order permission re-delegation attacks in Demissie's research (Demissie and Ceccato (2020)), are able to delegate part of the attacker's inputs to other components via the application's internal Intent. The two bugs are the main clue of the whole chain and glue all other bugs or features together.

We illustrate the exploit chain's workflow in Fig. 1. The two key intent-proxy bugs are highlighted in red. For simplicity, we tag the payloads in the chain so that we can better explain their relationships in later sections.

The first intent-proxy bug lies in the Samsung Vending application. The application implements a browsable activity that declares a *Browsable* category intent-filter element, allowing an attacker sends a malicious web URI intent to it. Once Samsung Vending's browsable activity receives the web Intent, it extracts two parameters named *id* and *url* from the input URI and then constructs a new Intent leveraging the two parameters. The *id* parameter is used to set the destination of the Intent via *Intent.*

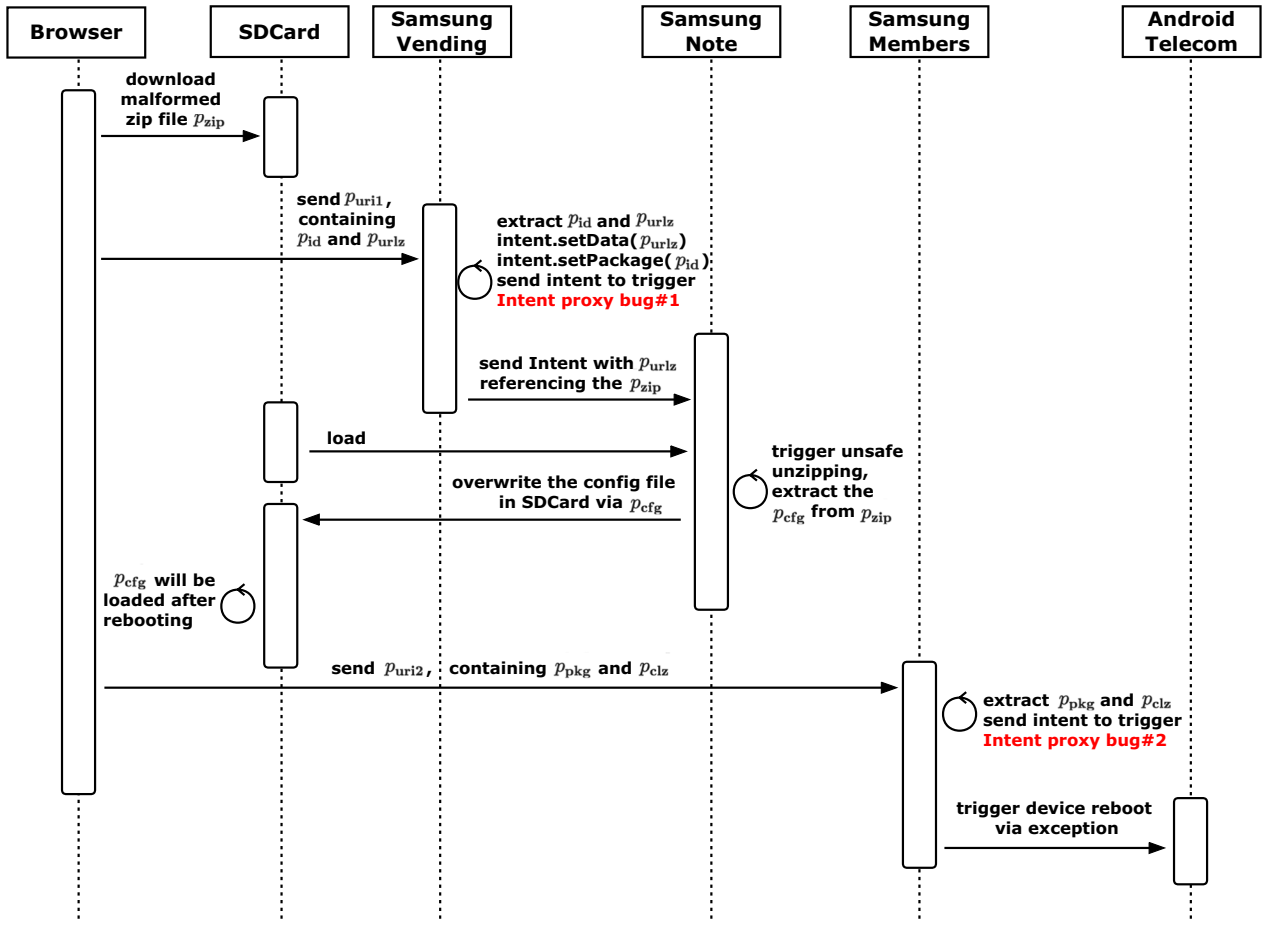

Fig. 1 The workflow of the exploit chain targeting Samsung S8

setPackage(id) method. The *url* is used as the argument of the *Intent.setData(url)* method. In the exploitation, the constructed intent is sent to a specified application to handle the attacker-controllable data URI. We hereby symbolize the attacker-controllable payloads involved in this stage. We name the web URI requested from the web browser as p_{uri1}, the extracted *id* parameter as p_{id}, and the *url* parameter as p_{urlz}.

The second intent-proxy bug lies in the Samsung Members application. This application's browsable activity extracts both the *packageName* and *className* from the web URI. The controllable package name and class name allow an attacker to launch any components under the context of Samsung Members. Similarly, we name the URI payload from the browser as p_{uri2}, the *packageName* parameter as p_{pkg}, and the *className* as p_{clz}.

The two intent-proxy bugs chains all the bugs and features in the whole chain. The first one was used to help trigger an unsafe unzipping bug in the Samsung Note application. A malformed zip file p_{zip} referenced by p_{urlz} was downloaded to the external storage beforehand. Using p_{cfg} extracted from p_{zip}, a configuration file stored in the SDCard was overwritten. The configuration file would be loaded by a snippet of leftover debug code and perform further code execution attacks. However, the leftover debug code is executed only after the device reboots. This condition is met by the second intent-proxy bug, which triggers a null object exception in the Android Telecom application and reboots the device.

Key observations

From the motivating example and some other exploit chains (Plaskett and Loureiro 2018; f-secure Lab 2019; Dawn Security Lab 2022), we summarize three key observations. Based on these observations, we found Android is a proper platform on which we conduct the exploit chain research.

Multiple attack surfaces on android devices

From the adversary's perspective, applications and their functionalities that receive attacker-controllable inputs are all attack surfaces (Sherman 2014). An android

device has various attack entrances due to a large number of installed applications and functionalities provided by them. These attack entrances include both the local attack surfaces such as local IPC mechanisms, and remote attack surfaces such as deep links sent from the browser.

The exploit chain introduced in section Motivating Example involves both remote and local attack surfaces. The chain was developed intentionally for a pwn2own competition, which requires compromising a target device remotely with very few user interactions (Initiative 2022). Thus the researchers launch an attack from a remote direction. Remote attack surfaces receive payloads in the form of web URIs and remote files. For example, researchers prepare p_{uri1} and p_{uri2} in their web server, waiting to be accessed from the target device's browser. In addition, the well-constructed p_{zip} was also downloaded in advance from their remote server. As for local attack surfaces, various inter-process, inter-component, and inter-application communication mechanisms are used in the chain, involving up to six unique applications and their exported components.

The diversity and quantity of attack surfaces allow an attacker to send various payloads to exploit the system from different directions and provide abundant raw materials for building a chain.

Measurable capabilities during exploiting

Exploiting an Android application written in memory-safe languages such as Java or Kotlin is less flexible than exploiting a memory-unsafe C/C++ program. On the Android platform, the attackers' ability relies more on the target application's internal functionalities and the controllability of their inputs. There are fewer chances for an attacker to behave beyond the application's invoked APIs, compared to the control flow hijacking and memory manipulation attacks in memory-unsafe programs written in C/C++.

In the motivating example, the attacker's ability depends completely on the inner behavior of the target applications. The exploit chains the functionalities that can be triggered or controlled. Although there are "arbitrary" component launching attacks involved in the chain, the limited flexibility also depends on the functionality of invoked API sequence, the controllability of each phase, and finite target components available on the device. In addition, the amount and category of APIs invoked by an application are finite, thus the capabilities that can be gained by an attacker are also finite and measurable.

The inflexibility for exploiting a memory-safe Android application leads to a finite and measurable set of an attacker's capabilities, making it feasible to measure a payload's behavior and capability set.

Dependencies between payloads

There are both data and control dependencies between two payloads from different directions.

An example of data dependency is the load-after-store operations in an application. Once the program receives the payload, part of the attacker-controllable data extracted from the payload may be stored in a temporary or persistent data entity, and later be loaded by functionalities triggered by another payload. In the motivating example, p_{zip} is downloaded in the SDCard beforehand, but it won't be loaded to assist the exploit until a proper p_{uriz} referencing p_{zip} was sent to trigger the unsafe unzipping functionality in the program.

The control dependency is revealed in "pending attack surfaces". A pending attack surface cannot receive the attacker's payload directly unless another payload drives the program's control flow and triggers certain events to turn the pending attack surface into a ready state. In the motivating example, the leftover debug code is in a pending state, it does not proactively load p_{cfg}, until payload p_{uri2} produces an event to reboot the system and transforms the leftover debug code into a ready state. Therefore, the p_{uri2} has control dependency with p_{cfg}.

Two payloads with data or control dependencies can be chained together to trigger more functionalities or enlarge the ready attack surfaces in the target program.

Logic exploit chain model

Based on the three key observations, we propose a general model of the Android application exploit chain.

The model involves three important roles—payload, gadget, and shared data entity.

The relations among these roles are shown in Fig. 2. In a scenario where multiple inputs can be sent to the target in different directions, an attacker carefully prepares and sends the *payloads* to an application's attack surfaces either simultaneously or in proper order. The payloads flow into the system from different directions, and trigger functional code snippets—*gadgets* in applications. These gadgets take input from and send output to the *shared data entity*. The entities chain multiple payloads together. As shown in the figure, the shared data entity are modeled inside one application as the internal data entity, or among multiple applications in the form of external data entity.

In the rest of this section, we introduce the threat model of our research, describe the concept of *Gadget*, define the IO properties of each gadget, and finally, explain the chaining rules for the payloads.

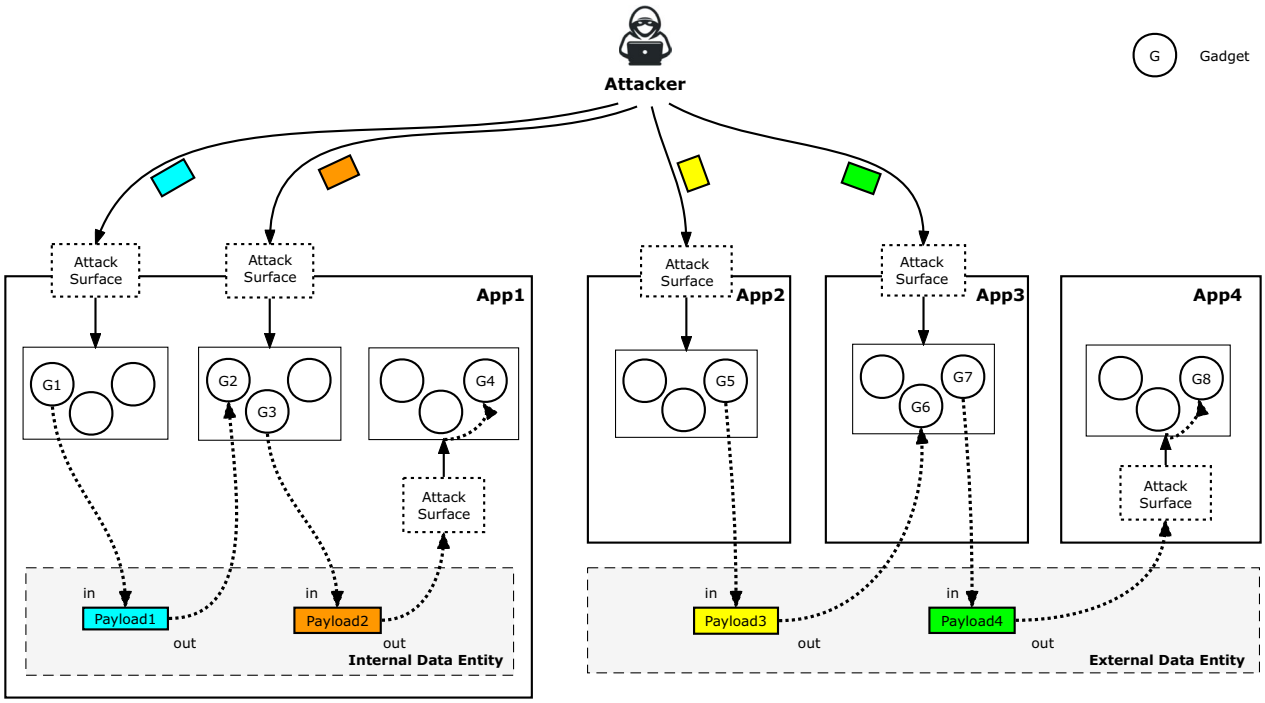

Fig. 2 Relations of multiple roles in the model

Threat model

Research scope

This work aims to find chainable payloads in Android applications, including third-party user applications, platform applications, and system applications. All types of applications share the same attack entrances and vulnerability patterns. In this paper, we only use popular applications in Google Play as a test suite to evaluate our model.

This work focuses on the applications implemented in Java or Kotlin, because the APIs of these languages are officially documented and are adopted by most Application developers. Functionalities implemented with C/C++ or hybrid applications based on HTML5 are not involved in this work, because describing the behaviors of C/C++ or hybrid applications is another challenge.

We do not detect specific bug patterns. Instead, we put more emphasis on the relationship of functionalities. We blur the boundary between an application's bug and feature, as essentially they are all considered sequences of APIs calls. According to several empirical studies (Linares-Vásquez et al. 2017) on Android application bugs and the OWASP mobile top ten risks (Owasp 2022), most bugs are brought by insecure usage of APIs.

Attack assumptions

The threat model assumes that an adversary can launch attacks both locally and remotely.

In a locally launched attack, it is assumed that a malicious application has been installed on the target device by the attacker. The malicious application can interact with other applications via various IPC mechanisms. In addition, the malicious application is granted with necessary permissions, including external storage access permission for allowing attackers to read and write files in the SDCard. Specifically, we assume the device user may interact with the application's operable UI widgets, such as clicking a button, inputting text in an edit box, etc. These user interactions facilitate producing necessary events and the execution of more pending functions in the application.

In a remotely launched attack, it is assumed that the user can be guided to click a malicious URL embedded in SMS, email, or instant message applications, so that unprotected browsable activities can be triggered by an attacker. We also assume an attacker can perform a "man-in-the-middle" attack and hijack plain HTTP streams to tamper requests or responses.

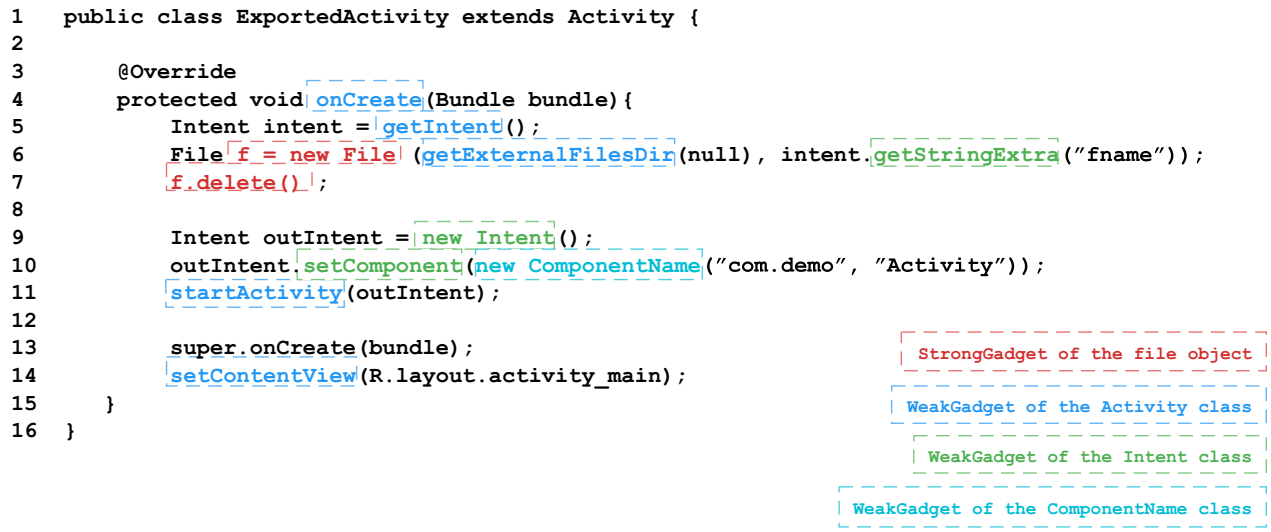

Fig. 3 An example of strong gadget and weak gadget

Attack aims

The aim of the threat model is to compromise user privacy and access system sources via chained payloads. Specifically, the adversary tries to trigger more functionalities with less privileges requested and less user interactions.

Gadgets definition

In this paper, a *gadget* is a snippet of code in an Android application that can be triggered by a payload or handles the data input of a payload. The term *Gadget* is borrowed from the Return Oriented Programming technique (Buchanan et al. 2008) in memory corruption exploits, where gadgets are snippets of code invoked for manipulating memory and registers.

According to whether the input parameters can be controlled by an attacker, we divide gadgets into two categories: weak gadgets and strong gadgets.

The input parameters of a *Weak gadget* have no association with the payload, the payload only triggers the execution of these gadgets. Weak gadgets are often leveraged to produce events or set intermediate states in a chain. The form of a weak gadget is a cluster of invoked APIs in the same class.

In comparison, part of a *Strong gadget*'s input parameter is extracted from the payload. Leveraging strong gadgets, the attacker can conduct more flexible manipulation of the applications' behaviors by assigning varied values as the input. The form of a strong gadget is a cluster of invoked APIs belonging to an object, rather than just of the same class.

The scope of APIs in gadgets includes but is not limited to Android development APIs, Java language APIs and other third-party library APIs. The invoking sequences of these APIs are carefully sliced from the whole program so that the functionality of a gadget is explicable.

Figure 3 presents a code snippet to illustrate the form of both strong and weak gadgets, and how the code is sliced. The *ExportedActivity* in line 1, as the name implies, is exported and allows another application to send Intents to it and launch it. In line 6, the application extracts data from the intent payload and uses it as the parameter of the *File.$init* method. Thus *File.$init* and the API of the same object *File.delete* can be triggered by the intent payload. In addition, at least one of their parameters is attacker-controllable. Therefore, the group of two APIs belonging to the *file* object is marked as a strong gadget. In comparison, some API calls in the *onCreate* method are just triggerable but have no data relations with the intent payload. After classifying these APIs according to their class name, we get three weak gadgets. The *getIntent* (line 5), *getExternalFilesDir* (line 6), *startActivity* (line 11), *onCreate*(line 13), *setContentView* (line 14) of the Activity class; the *$init* (line 9), *getStringExtra* (line 6), *setComponent* (line 11) of Intent class; the *$init* (line 10) of ComponentName class.

Formally, we use $G(p)$ to denote the set of gadgets triggered by payload p. The $G_w(p)$ and $G_s(p)$ are used to denote the set of weak gadgets and strong gadgets separately. $G(p)$ is the union of weak gadget set and strong gadget set:

$$G(p) = G_{\mathrm{w}}(p) \cup G_{\mathrm{s}}(p)$$

Gadgets IO property

We model the input and output (I/O) properties for a gadget to help explain the dependencies between payloads. We divide the I/O properties of a gadget into three categories—*DataIO*, *ControlIO*, and *DataControlIO*. These properties respectively reveal the *DataDependency*, *ControlDependency*, and both dependencies for two payloads.

DataIO indicates that the gadget takes data input from or outputs its results to an intermediate data entity—probably a persistent or temporary place for storing data. DataIO only involves strong gadgets due to the attacker-controllable data input requirements. A strong gadget's DataIn propagates the attacker-controllable data input to the gadget's parameters or member variables. After the DataIn is manipulated or propagated inside the gadget, attacker-controllable data are outputted to another data entity. In our model, there are four kinds of functional gadgets with DataOut ability. (1) the static value assignment statements outputting data to a temporary data entity. (2) the file writing operations, including both writing data into external storage and an application's private directories. (3) the shared preference editing operations writing key-value style information into configuration files. (4) the database operations writing structured data into a persistent database.

ControlIO of a gadget reflects the control requirements to be met for triggering its execution, and the abilities to awaken other code snippets that are not executed. As Android applications are event-driven, there are multiple execution entries in the form of callback methods instead of the sole main method. In our model, ControlIn is a set of user interaction events and some component lifecycle events for triggering these callback methods. The ControlOut abilities are revealed in the gadget's specific APIs used to trigger the execution of other components, e.g., the *startActivity()* API used to awaken another Activity component, and the *sendBroadcast()* API used to send a broadcast to another Broadcast Receiver component. The model takes all of the four kinds of basic components into consideration, thus their corresponding Inter-Component Communication (ICC) methods' ControlOut abilities are modeled.

DataControlIO is the combination of DataIO and ControlIO. A gadget with a nonempty DataControlOut property extracts data from the payload, and then sends the extracted data to other components to trigger it. e.g., a gadget extracts a value from the payload, uses it as the parameter of *Intent.setData()* or *Intent.putExtra()* methods, and then sends the Intent to awaken other components.

Payload dependency

Two payloads p_1 and p_2 are chainable if at least one gadget triggered by payload p_1 is data or control dependent on the gadgets of payload p_2.

Data dependencies are uncovered from the DataIO properties of multiple gadgets. We use $In_{\mathrm{d}}(g)$ and $Out_{\mathrm{d}}(g)$ to denote the DataIn and DataOut set of a gadget g. Payload p_1 and payload p_2 have data dependency if the DataOut in one payload fits the DataIn of another payload. The fitness means that the two values are originated from the same taint source. For example, in a situation where a gadget g_1 in p_1 outputs data $Out_{\mathrm{d}}(g1)$ to a data entity, and then another gadget g_2 triggered by payload p_2 takes input from the same data entity, p_1 and p_2 are data dependent. Formally, the requirements for data dependency between p_1 and p_2 are shown in the following conditional expressions. Note that the $ts(v)$ is the taint source of value v.

$$\begin{aligned}
&\exists g_1 \in G_{\mathrm{s}}(p_1), g_2 \in G_{\mathrm{s}}(p_2)\\
&\exists v_1 \in Out_{\mathrm{d}}(g_1), v_2 \in In_{\mathrm{d}}(g_2)\\
&p_1 \neq p_2, ts(v_1) = ts(v_2)
\end{aligned}$$

Control dependencies are associated with the ControlIO properties of multiple gadgets. The dependency exists among both strong strong gadgets and weak gadgets. Formally, we define p_1 and p_2 have control dependency if p_1 can trigger the execution of gadget g_2 in p_2. This requires there are gadgets with ControlOut ability in p_1, and the ability matches the event requirements for awakening the execution of gadget g_2. We express the dependency as follows.

$$\begin{aligned}
&\exists g_1 \in G(p_1), g_2 \in G(p_2)\\
&Out_{\mathrm{c}}(g_1) = In_{\mathrm{c}}(g_2)\\
&p_1 \neq p_2\\
&In_{\mathrm{d}}(g2) \neq \emptyset
\end{aligned}$$

It is possible that both data and control dependencies exist between two payloads. This requires the above conditions are satisfied simultaneously. Two dependent payloads can be chained together by attackers to either gather deeper controllability of the target program or enlarge the attack surfaces.

Design

Based on the exploit chain model, we design a prototype system named AppChainer to automatically chain the payloads towards various applications.

In this section, we first give an overview of AppChainer to introduce its key modules and overall workflow, then

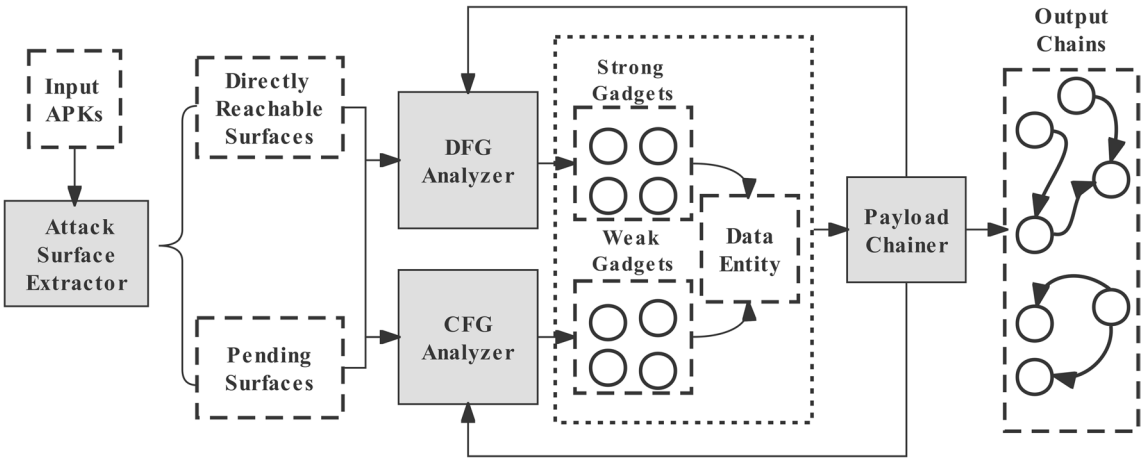

Fig. 4 Architecture of the prototype system AppChainer

explain the details of each module in the latter part of the section.

System overview

AppChainer takes APK files and several configuration files as input, and outputs possible chainable payloads in an application or among multiple applications.

As shown in Fig. 4, AppChainer is composed of four modules: Attack Surface Extractor, Control Flow Graph (CFG) Analyzer, Data Flow Graph (DFG) Analyzer, and Payload Chainer. The modules are denoted in the figure with grey boxes.

The workflow of AppChainer is divided into four steps. Attack Surface Extractor identifies attack surfaces under our threat model, i.e., both the local and remote entrances for receiving payloads in an application. CFG Analyzer gathers *weak gadgets* for each payload, and DFG Analyzer gathers *strong gadgets*, respectively. Payload Chainer searches the data and control dependencies between gadgets of various payloads, calculates the chainability of the payloads, and finally outputs the results.

Attack surface extractor

Attack Surface Extractor takes APK files as inputs, and extracts two sets of attack surfaces as outputs. In this section, we first analyze our observations on the state of an attack surface, then divide the attack surfaces into two categories according to whether or not an attack surface is directly reachable. In line with the attack surface classification, the concepts of two payload categories are also derived. At last, we introduce the attack surface identification methods.

Despite the large number of attack surfaces, an attacker can only proactively reach a part of them, such as sending his payload to Browsable activities or unprotected exported components. While other part of attack surfaces are not in a ready state unless their corresponding code snippets are executed and enabled, e.g., the external file-loading attack surfaces. One cannot directly put a file in the external storage and let the application load it immediately unless the program is told to do, or the file observer service (Google 2022a) is executing in the background.

Based on the above observation, we divide the attack surface into two categories: *directly reachable attack surfaces* and *pending attack surfaces*. Directly reachable attack surfaces, including browsable activities and unprotected exported components, start executing and handling the attack controllable inputs once receiving the payload. While pending attack surfaces are in a pending state unless certain events are sent, triggering the execution of its code snippets.

Corresponding to the classification of attack surfaces, payloads are classified into two categories. We name the payloads entering these directly reachable attack surfaces as *immediate payloads* because they can take effect in an immediate way. Relatively, we name the payloads entering the pending attack surfaces as *pending payloads*.

To extract directly reachable attack surfaces, the Attack Surface Extractor first decompiles the APK file to obtain the Android manifest file, from which the browsable activities and exported components are identified. Among the exported components, AppChainer pays special attention to the intent-filter attribute that changes the default value of the *android::exported* attribute (Chen et al. 2016). In addition, AppChainer resolves the permission requirements of a component to exclude the protected components. If a component is protected by a self-defined system or signature-level permission, then

we ignore it because our threat model does not assume that an attacker has the ability to break through the sandbox restriction.

To identify pending attack surfaces, we provide a default potential attack surface list containing behaviors such as file operations, shared preference and database operations, etc. We also provide a flexible way for researchers to extend APIs that can be marked as attack entries. The payloads entering these components are pending until they meet another payload or event capable of arousing them, i.e., has control dependency with them.

The Attack Surface Extractor outputs the list of directly reachable attack surfaces and the list of pending attack surfaces.

CFG analyzer

The CFG Analyzer takes the two identified attack surface sets as its inputs, builds an interprocedural control flow graph (iCFG), performs analysis on the graph to identify gadgets, and eventually outputs the weak gadget set $G_w(p)$ of each payload p.

The starting points of constructing iCFGs vary according to the types of attack surfaces. Directly reachable attack surfaces have explicit entries, thus the CFG Analyzer constructs iCFGs on their entry methods. Specifically, each component has an enumerable set of entry methods, e.g., the lifecycle methods (*onCreate*, *onResume*, etc.) in an activity, the *onStartCommand* and *onBind* methods in a service, *onReceive* method in a broadcast receiver, and database operating methods in a content provider. In addition, callback functions handling the user interaction events are also marked as entry points, e.g., the *onClick* method of a button.

Pending attack surfaces may be located in the middle of a call graph. Thus the CFG Analyzer performs backward analysis besides the forward analysis for constructing a complete control flow graph. The backward analysis process terminates when it reaches the lifecycle methods of a component or user interaction callback methods of a widget. Different from the directly reachable components, the result component may not be an exported component and relies on other payloads to start it.

Weak gadgets are identified via static CFG analysis. The recognizer follows the iCFG to collect invoked API methods and then sort them into clusters sliced by distinct classes. For each invoked API method, the analyzer checks every parameter of it. Backward constant analysis is performed to record the deterministic parameters of each API method. The method names and their constant parameters are leveraged to reason the IO properties for later chaining. The sliced weak gadgets are then stored in a database file, associated with a unique payload id. The detail of its algorithm will be described in section Weak Gadget Identification.

DFG analyzer

DFG Analyzer also takes the two attack surface lists as its input. It uses taint analysis to identify the strong gadget set $G_s(p)$ of a given payload p.

DFG Analyzer reuses the iCFG constructed in the CFG Analyzer stage. It marks the payload receiving methods as taint sources and then propagates the taints through the iCFG. Once the invoked API's parameters are attacker-controllable data or tainted by the payload, then we slice the cluster of APIs as a strong gadget according to its object. During slicing, we use alias analysis on the same object to guarantee that the invoked APIs are completely sliced in the strong gadget. Similar to weak gadget identification, constant analysis of each parameter is also performed. Finally, the sliced strong gadgets and their parameter information that reveals which parameter is attacker-controllable are stored in another table of the database.

We made modifications to traditional taint analysis techniques to better fit the strong gadget identification scenario. In traditional taint analysis, taint sources and taint sinks are predefined. The aim of a traditional taint solver is to confirm whether there are reachable paths between the sources and sinks. However, in the procedure of identifying strong gadgets, we care more about the set of tainted intermediate nodes in the data flow graph, rather than the results indicating whether a source can reach a sink. Thus we do not define a specific taint sink and instead focus on how to handle the taint propagation procedure. The modified taint analysis techniques will be shown in section FlowDroid Integration, the detail of the algorithm will be described in section Strong Gadget Identification.

Payloads chainer

With the iCFGs and two databases generated by the two analyzers, the Payloads Chainer then investigates the data and control dependencies within these payloads and picks out the payload combinations that can be chained together to trigger new functionalities.

The payload chainer uses an iterative strategy to chain as many payloads as possible. It starts with all the immediate payloads, analyzing the DataOut and ControlOut properties of each gadget. Then searches proper code points that consume the data or control output event. Once one or more consumers are matched, the analyzer invokes the CFG or DFG analyzers to propagate its data or enlarge its control impact. After all the immediate payloads are analyzed, the chainer chains the payloads

that have data or control convergence. The chaining algorithm will be explained in Sect. Chaining Rules.

The number of chainable payloads is not limited to two. In some large applications, the dependencies may exist in more than two payloads and constitute a graph. Eventually, the payload chainer outputs the set of potentially chainable payloads both in the form of payload pairs and payload graphs.

Implementation

AppChainer is built based on FlowDroid, a well-known static analysis tool for Android applications. In this section, we first introduce the modifications we made in FlowDroid and how we use FlowDroid in AppChainer, then explain the algorithms of the analyzers and chainer.

FlowDroid integration

FlowDroid is a precise context, flow, field, and object-sensitive taint analysis framework for Android applications. It receives an input file defining the taint sources and sinks, implements an IFDS Solver by describing the taint analysis problems, and eventually confirms whether there are paths between sources and sinks by solving the described reachability problem.

We integrate AppChainer with FlowDroid to conduct basic tasks such as iCFG generation, taint propagation, alias analysis, etc. Despite the popularity and mightiness of FlowDroid, it cannot be applied directly to strong gadget identification. The reasons are as follows: (1) In the gadgets identification process, the taint sources are not fixed and sinks are not finite, making it infeasible to prepare an explicit set of sources and sinks. (2) Static values, which are important data load and store points for chaining various payloads, cannot be marked as independent sources or sinks. In the rest of this section, we describe these problems in detail and give our solutions for them.

Handling taint source and sink

Taint analysis is capable of solving integrity or confidentiality problems (Lerch et al. 2014). FlowDroid is more often used to solve confidentiality problems such as information leakage, where sources are the generation points of sensitive information, and sinks are the exit points that transfer data out of the system. To implement AppChainer, we are facing a non-standard integrity problem and confronted with two challenges: (1) Dynamic sources. (2) Infinite Sinks.

In the integrity problem, taint sources are attack entries that receive attacker-controllable data, and sinks are the strong gadgets we need to gather. There are two reasons the taint sources are not fixed in AppChainer. Firstly, not all occurrences of an API can be marked as attack entrances. For example, when solving an information leakage problem, researchers can mark the *Location.getLatitude()* and *Location.getLongitude()* as sources to identify whether the location information is leaked, while in gadgets extraction, one cannot simply mark a specific API such as *Intent.getStringExtra()* as a source, because not all Intents are controllable by an attacker. Secondly, As shown in the Payload Chainer's algorithm, the DataOut property of a strong gadget would be marked as a new source and added to DFG Analyzer's working queue, thus we should handle a dynamic set of sources.

Taint sinks are also not enumerable in AppChainer. Before the analyzer executes, we do not know what kinds of APIs a payload will flow through. Thus it is not applicable to define a set of APIs as the sink. Extra solutions should be conducted for solving the infinite sink set problem.

To solve the dynamic source problem, we provide both a statically predefined source template and an on-demand source injection interface. In the statically predefined source template, we collect both local and remote APIs that are possible for receiving a payload, including operations on files, shared preferences, databases, and network streams. The template file is in the form of an XML file and is extensible for researchers to add third-party potential attack surface APIs into it. These items will be loaded by the attack surface extractor to exclude the not controllable entries. The on-demand source injection is performed in the gadgets chaining stage. Whenever a new taint is propagated to a temporary or permanent data entity, new sources will be added as a source seed, then the DFG analyzer will be invoked to perform further propagation analysis.

To deal with the infinite sink set problem, we first patch the FlowDroid to make the empty sink be allowed. Then we implement a Taint Propagation Handler to get the taint flow paths in FlowDroid's solving procedure. FlowDroid's IFDS implementation involves four types of flow functions. Among them, the call flow functions are responsible for processing a call site, If an incoming taint abstraction shows as the parameter of the invoked method, the customized handler gets informed and stores the gadget in the database's corresponding table, according to the type of the payload.

Handling static value

In FlowDroid, the declaration of taint sources should be in the form of an API method. While in AppChainer, the DataOut of a gadget may be stored in a static field. The static field will be treated as a new taint source, but this is not supported by a vanilla FlowDroid.

As a solution, we implement the on-demand taint analysis feature for FlowDroid and add specific logic for

handling the static values. Once a tainted value is propagated to a static field, we perform a backward analysis to find the definition of the static value. For each load point of that static value, we analyze whether it is reachable based on the immediate payloads' iCFGs. For the triggerable load points, the chainer invokes the DFG Analyzer to propagate the tainted value further, and marks the involved payloads as chainable. For the pending ones, we just record them and do not conduct immediate analysis on them until these load points are awakened by other payloads, i.e., their ControlIn requirements are satisfied.

Weak gadget identification

CFG Analyzer follows the constructed iCFG to collect invoked API methods and then sort them into clusters reasoned by distinct classes. The algorithm of weak gadget identification is shown in Alg. 1.

In line 3, the analyzer traverses the iCFG to get every statement in the intermediate representations. The *getStmts()* method handles the loop of the iCFG and uses a visited tag for each procedure. It returns a set of interprocedural unique statements. In line 4, the analyzer determines whether the statement is an invoke expression. In lines 5-12, the class name and method name are extracted from the invoke expression and temporarily stored in the res data structure. In lines 13-15, for each invoked methods, the analyzer checks every parameter of it. Backward constant analysis is performed to record the deterministic parameters of each API method. For efficiency, we do not perform alias analysis here, as the data dependencies are transparent in the CFG analysis stage.

Algorithm 1 Weak Gadget Identification Algorithm

```
INPUT: iCFG   ▷ the iCFG that contains correlated statements
OUTPUT: res   ▷ the weak gadgets result map sorted by classes
 1: function CFGANALYZE(iCFG)
 2:     res ← {}
 3:     for all stmt ∈ iCFG.getStmts() do
 4:         if stmtinstanceOfInvokeExpr then
 5:             methodname = stmt.getMethodName()
 6:             classname = split(methodname)
 7:             if classname ∈ res then
 8:                 res[classname].methods.add(methodName)
 9:             else
10:                 res[classname].methods = new set()
11:                 res[classname].methods.add(methodName)
12:             end if
13:             for i = 0 to stmt.getNumOperands() − 1 do
14:                 v = ConstAnalyze(stmt.getParam(i))
15:                 while v instanceOf Constant do
16:                     res[classname].params.push((i, value))
17:                 end while
18:             end for
19:         end if
20:     end for
21:     return res
22: end function
```

Strong gadget identification

The algorithm for strong gadget identification is more complex than weak gadget identification. It uses taint propagation techniques as its core, aided by constant analysis and alias analysis to extract strong gadgets along the iCFG.

The input of this algorithm is an attack entry corresponding to a payload. In this work, attack entries are in the form of the return value of an API call or a static value. The output is the result of identified strong gadgets.

In line 2, we initialize a ResultSaver utility class to save the identified strong gadgets and their corresponding extra information in a database. The ResultSaver sorts each identified strong gadget into clusters by distinct objects, or distinct classes if the target API call is static. There are two member functions in the ResultSaver, as shown in line 9 and line 12. The *saveGadget()* get the class name and method name from the stmt, and add the invoked API into a proper map structure, while the *saveExtraInfo()* saves the results of const analysis of each parameter in the invoked API method. For simplicity, the implementation detail of the ResultSaver is not expanded in the algorithm.

In lines 3-4, we gather all the tainted nodes of a given payload and then process each taint abstraction. In lines 5-6, the algorithm gets the statement of the taint abstraction, and determines whether it is a invoke statement for APIs. In lines 7-9, the ResultSaver put the current API call into the proper position in the database. In lines 10-13, each operand of an API call site is processed. Const analysis is performed on each operand to trace the data flow backward to check whether the operand or part of the operand is a constant value. The const analysis result is also saved into a database as extra information of the API call, and will be used in the latter gadgets chaining phase.

Algorithm 2 Strong Gadget Identification Algorithm

```
INPUT: entry                          ▷ the entry of a payload
OUTPUT: res          ▷ the strong gadget result of the payload
 1: function DFGANALYSIS(entry)
 2:     res ← ResultSaver()
 3:     taints = propagate(entry)
 4:     for all abs ∈ taints do
 5:         stmt = abs.getStmt()
 6:         if stmt.isInvokeExpr() then
 7:             meName = stmt.getCalledMethod()
 8:             clName = split(methodName)
 9:             res.saveGadget(entry, meName, clName, stmt)
10:             for op ∈ stmt.getOperands() do
11:                 cInfo = ConstAnalyze(operand)
12:                 res.saveExtraInfo(entry, cInfo, op)
13:             end for
14:         end if
15:     end for
16:     return res
17: end function
```

Chaining rules

With the gadgets database outputted by the gadget analyzer, the payload chainer leverages the chaining algorithm to identify the set of potentially chainable payloads.

The algorithm of the payload chainer is shown in Alg. 3. It first reasons the DataOut and ControlOut properties of every collected gadget (lines 4 and 15). As illustrated in section Gadgets IO Property, four kinds of DataOut related to different data entities and four kinds of ControlOut related to ICC are modeled.

The load points of the corresponding outputs are inferred (lines 5 and 16), determined by a matching strategy including both fuzzy searching and exact searching. If the load points are not empty, CFG and DFG analyzers are performed on these inferred points, as shown in lines 7 and 18.

The chainer chains p' with p if there are data or control dependencies between them, as shown in lines 10 and 21. In lines 11 and 22, the chainer iteratively adds the dependent payload p' into the work queue for future processing tasks. The iteration allows the system to dig as many payloads as possible in a chain.

Algorithm 3 Payload Chainer Algorithm

```
INPUT: ip                          ▷ an intermediate payload for chaining
OUTPUT: res                        ▷ items that can be chained with the inpu
 1: function GETCHAINS(entry)
 2:     res ← {}
 3:     for all g ∈ p.gadgets() do
 4:         if g.hasDataOut() then
 5:             loadPoints = inferLoad(g.dataOut)
 6:             for all load ∈ loadPoints do
 7:                 DFGAnalyze(dest)
 8:                 p' = getPayload(load)
 9:                 if p'.isImmediate() then
10:                     res.dataDep.add(p')
11:                     addWork(p')
12:                 end if
13:             end for
14:         end if
15:         if g.hasControlOut() then
16:             dests = inferDest(g.ctrlOut)
17:             for all dest ∈ dests do
18:                 CFGAnalyze(dest)
19:                 p' = getPayload(dest)
20:                 if p'.strongGadgets() ≠ ϕ then
21:                     res.ctrlDep.add(p')
22:                     addWork(p')
23:                 end if
24:             end for
25:         end if
26:     end for
27:     return res
28: end function
```

Evaluation

In this section, we will introduce the environment of our experiments, explain how we collect the APK test suite and construct the benchmark, and give the evaluations of (1) the effectiveness of the ability to discover attack surfaces, (2) the effectiveness of identifying payloads in real-world applications, (3) the precision and recall rates compared with Jandroid, (4) the performance of AppChainer.

Environment setup

We execute the AppChainer in a machine running Ubuntu 20.04 system with an Intel Core i7-6700 3.40GHz processor and 40 G RAM.

We use popular applications downloaded from Google Play as the test suite. A total of 2138 applications with more than one million installations are analyzed by the system without reporting errors. As the FlowDroid engine fails to analyze some large applications under our hardware facilities, the application sizes are on the lower side. The average size of the 2138 applications is 37.20 MB. There are only 27 applications larger than 100MB.

We set the memory warning limit to 32GB to provide sufficient memory for analysis. The timeout limit for the data flow analyzer (the most time-consuming phase) is set to 10 min. Applications that cannot be analyzed successfully under the current execution configuration and computation environment will be discarded.

Test set collection

We evaluate the AppChainer on two test sets. The effectiveness and performance of AppChainer are evaluated over a large number of applications from the Google Play Store. The precision and recall rates are evaluated on our custom benchmark applications named DroidChainBench.

Popular application collection

We randomly collect popular applications from Google Play Store. The metadata of an applications is gathered based on the google-play-scraper (JoMingYu 2022) project on GitHub. The project provides a searching API, returning at most 30 recommended results according to a given keyword. With the returned *appId* field, we construct a deep link referencing the application and send it to the gms (Google Mobile Service) application. The gms handles the link and displays a page showing the detail of the application. The application will be installed on the device after clicking the install button. We eventually collect the APK files via the ADB command. All of the actions are performed automatically via a python script based on uiautomator.

We generate random words or phrases as the keywords for searching, so that the categories, developers and functionalities of applications are random. From the metadata of an application, the "installs" field reveals its popularity, we exclude the less popular applications with less than one million installs.

Table 1 The excerpt results of applications identified payloads entering identified attack surfaces, sorted by the number of total payloads

id	Package name	Version	Immediate payload	Pending payloads				Total
				File	sp	db	Intent	
1	com.***.***.**ad	1.46.10	35	368	393	178	147	1121
2	com.***.***.**id	5.54.0	54	248	153	61	431	947
3	com.****.***.**ok	4.2224.2	34	237	204	94	163	732
4	com.***.**le	8.61.0.100(1.3.262439.0)	13	281	161	89	82	626
5	com.***.**ms	1416/1.0.0.2022314401	15	218	66	58	133	490
6	com.***go	5.63.3	7	152	174	78	43	454
7	com.***.**ip	17.8.0.0	7	146	67	44	173	437
8	tv.***.***.**pp	13.3.1	8	130	80	119	22	359
9	com.***.**er	4.9.20	15	136	116	45	34	346
10	com.**.**op	3.22.06	8	167	61	23	37	296
11	cn.**er	12.1.2.Prime	12	147	28	57	13	257
12	com.**do	5.15.4.2	36	62	53	23	35	209
13	com.***.**sh	1.7.3	7	58	35	49	58	207
14	com.**ze	4.84.0.2	14	49	59	6	78	206
15	com.**tv	4.25.1	6	106	68	8	15	203

For ethical reasons, package names are blurred with the symbol "*"

DroidChainBench

As there is no suitable benchmark to evaluate the precision and recall rates of exploit chains in Android, we design the *DroidChainBench* specifically for the Android exploit chain model.[1] The benchmark, now in its version 1.0, consists of 10 hand-crafted test cases that perform reading and writing operations on data entities such as files, static values, intents, shared preferences and databases.

DroidChainBench implements a handful of exported components, which meets the attack assumption that an adversary has multiple attack entrances to send their payloads to the components in these applications. The benchmark implements all kinds of data entities in our model, revealing both data and control dependencies among various payloads. We also add two false chains to test the precision of AppChainer, two external shared data entity operations to test payload chaining among multiple applications, and the permission definition and request test case that exposes the deficiency of AppChainer. As the payload chainer inspects and chains payloads two by two at a time, thus the identification result on two applications also covers the scenario across more than two applications.

Besides, we do not apply functional tests such as the callback, jni, or lifecycle mechanisms that influence the precision of taint analysis, because these aspects are tested by the FlowDroid engine already. We only focus on whether the payloads can be successfully identified and chained under the taint analysis technique in the present.

Attack surface evaluation

The number of extracted attack surfaces reflects whether there are abundant material payloads available for chaining.

For an application, we identify both the immediate payloads that can be received by directly reachable attack surfaces, and pending payloads waiting to be accepted by pending attack surfaces. For the pending payloads, we calculate four kinds of payload form, including file reading, shared preference getting, database querying, and intent receiving operations.

We rank the applications by the number of total potential attack surfaces, the excerpt results of identified attack surfaces are shown in Table 1.

Chainable payloads evaluation

We investigate the control and data dependencies between payloads both inside one sole application and among various applications. The dependencies between payloads determine whether the available payloads are capable of being chained together, and how many payloads can be involved in each chain.

[1] The source code of the DroidChainBench will be uploaded to https://github.com/xiangxiaobo/DroidChainBench.

Table 2 The number of involved payloads in the 5458 chains

#Involved payloads	2	3	4	5	6	7
#Chain	3036	1730	409	150	112	21

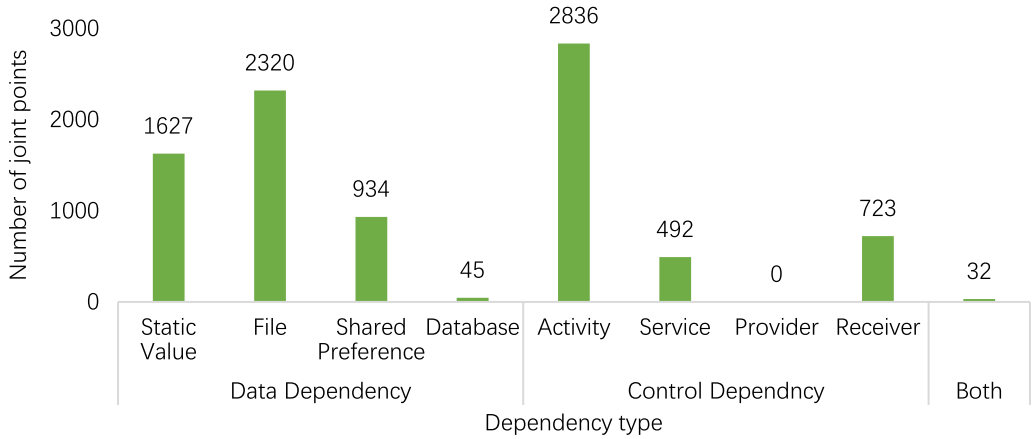

Fig. 5 The distribution of dependency types

AppChainer effectively identifies 5458 chains consisting of 14467 payloads. Table. 2 shows the number of chains corresponding to the number of payloads. The most complex chain identified by AppChainer contains up to seven various payloads, and 21 such chains are identified. The long chain is the result of the iterative analysis strategy in the payloads chainer algorithm.

To answer the question about how two payloads are chained together, we investigate the type of disjoint points in the identified payload chains. According to the type of dependencies, we classify each chain's joint points into 9 categories. Among the 5458 chains, there are a total of 9009 joint points. Figure 5 shows the number of joint points for each dependency category. As in the figure, there are no joint points generated according to the control dependency between Providers, because we found few exported providers in the test APKs. Only a few joint points are generated based on both data and control dependencies.

Specifically, we calculate the number of payloads in the form of web Intents that can be chained with other payloads. Web Intent, which is the only IPC mechanism that can be triggered remotely, is security-essential (Liu et al. 2017) for an Android application. Among the 14467 identified payloads, 532 of them are web Intents that can be sent to exported Browsable activities via the browser, revealing the possibilities for remote attackers.

We also calculate the number of triggered gadgets in each chain. Compared with a single entry payload, in the chain, the number of strong gadgets increases by 87%, weak gadgets increases by 162% respectively.

Table 3 DroidChainBench test result

No.	AppName	AppChainer	Jandroid
1	FileEntityChain	⊞	⊞
2	SpEntityChain	⊞	⊞
3	StaticValueEntityChain	⊞	⊠
4	DbEntityChain	⊞	⊞
5	IntentChain	⊞	⊞
6	IntentSetDataChain	⊞	⊞
7	FalseFileChain	⊟	⊕
8	FalseIntentChain	⊟	⊕
9	PermDef & PermUse	⊖	⊞
10	ExtFileUse & ExtFileWrite	⊞	⊞

⊞ = detected (precisely)
⊟ = not detected (precisely)
⊠ = not supported
⊖ = false negative
⊕ = false positive

Precision and recall

We compare the precision and recall rates of AppChainer with Jandroid on the DroidChainBench. The test result is shown in Table 3. Experimental results show that AppChainer and Jandroid can both detect the chains involving files, databases, shared preferences, and intents.

The precision rate of AppChainer is 100%. The identified chains are all correct. In comparison, the precision rate of Jandroid is 77.78%. It identifies nine results, two of which are false postives due to the lack of a dataflow track.

The recall rate of AppChainer is 87.5%. It detects seven results out of eight. In the PermDef application, we define a signature-level permission. The PermUse application has the same signature as the PermDef. It applies the application and is able to access the protected component in PermDef. AppChainer fails to chain payloads in the permission-related test case, because currently we do not model the Android Application permission and SELinux rules. In contrast, Jandroid can detect the chain with a template that simply does not check permissions or with a sophisticated template that finely models the permission rules during manifest searching. The recall rate of Jandroid is also 87.5%. It fails to analyze the static value entity test case because it does not support identifying operations on static values, because its code-searching rules are designed for APIs.

During the comparison experiment, we find it is complicated to construct the searching rules in ".template" and ".link" files in Jandroid. In total, we write 1406 lines of template code for Jandroid in the benchmark. While in AppChainer, we do not need to write extra code for each of the chains in our model, the gadgets are output to the database and chained by the chainer automatically.

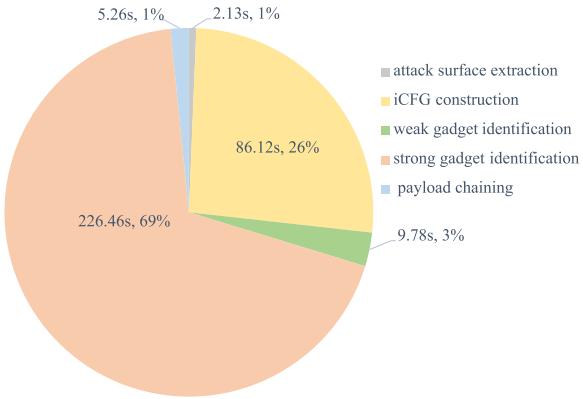

Fig. 6 The average execution time of each stage

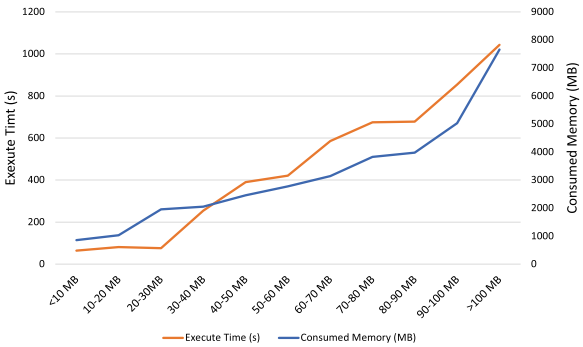

Fig. 7 The growing trends of memory and runtime overhead with the application sizes increase

Performance evaluation

By default, FlowDroid uses the most precise configurations to identify as many data flows as possible. The higher precision brings higher performance overhead. To lower the overhead, we compromise the precision by limiting the length of the access path to 2 and using the flow-insensitive alias analysis option instead of the default flow-sensitive way. Meanwhile, we turn off the static value analyzer because we already handled the static value propagation problem, as explained in section FlowDroid Integration. We also turned off the callback analysis because we regard message transmission or thread creation callback methods as functional gadgets. Their functionalities are shown in their corresponding ControlOut or DataOut properties and will be used as a rule in the gadget chainer module.

With these configurations, we evaluate the execution time of every single stage and the maximum memory consumption of the whole task. For the 2138 applications in our test suite, the average execution time is 317.73 s, the average memory consumption is 2368MB. Figure 6 shows the average execution time of every stage of the whole task. As the chainer invokes the strong and weak gadget identification module during its execution, we strip the consumed identification time in the chaining stage, adding it to the strong or weak gadget identification stage separately.

In general, the memory and runtime overhead increase with the size of APK files, as shown in Fig. 7.

Related work

As there are no existing chain building methodologies to our best knowledge, we introduce the relevant works of AppChainer in two aspects: the recent logic bug exploit chains, and single bug patterns proposed by researchers.

Logic bug exploit chains

Except for the Samsung exploit chain explained in section Motivation, we also investigate other industrial logic exploit chains in Android platforms.

Mystique exploit chain (Dawn Security Lab 2022) leverages a SEPolicy misconfiguration bug in recent Android versions, which allows a system app to overwrite another application's original private files and break the Android application sandbox. By Chaining with other bugs, this

exploit chain can eventually trojanize third-party applications on the target device.

Huawei Mate 9 Pro was pwned by Plaskett and Loureiro Plaskett and Loureiro (2018) via chaining two bugs in HiApp and four bugs or features in the Huawei Reader application. They eventually managed to create a bind shell of the target device.

Xiaomi Mi9 (f-secure Lab 2019) was pwned at mobile Pwn2Own 2019 using two different exploit chains, launching the attacks from the web browser and NFC tag, respectively. Researchers eventually achieved remote file theft on the victim's device.

Android application bug patterns

Academic research in Android vulnerability discovery and exploitation proposed several harmful vulnerability patterns.

CHEX (Lu et al. 2012) is a static analysis tool to find component hijack vulnerabilities in Android applications. The component hijacks they generalized include permission leakage, unauthorized data access, intent spoofing, etc.

Permission re-delegations are investigated in Android applications (Demissie et al. 2020), the Android framework (Felt et al. 2011), and system services (Gorski III and Enck 2019; key 2022). Researchers find potential deputies that bypass permission checks.

The Next Intent Vulnerability, where an Intent is embedded in another Intent and is delegated to another component, allows an attacker to invoke and send data to a private component of the victim app. It is first used to build an attack in Wang et al. (2013) and is well modeled and scrutinized by researchers in El-Zawawy et al. (2021).

Similar to the Next Intent Vulnerability, a second order permission re-delegation (Demissie and Ceccato 2020) pattern extracts data from the received Intent, creates a new Intent containing the extracted data, and sends it to other components under its privilege context.

The security threats brought by insecure usage of PendingIntents (Google 2022b) are researched by PIAnalyzer (Groß et al. 2018) and PITracker (Zhang et al. 2022). In the worst case, an attacker may gain SYSTEM privileges to perform the most sensitive operations, e.g., deleting a user's data on the device.

Choi and Kim (2018) investigate general remote code injection attacks. In this research, several file-overwriting vulnerabilities are listed, e.g., unsafe zip extraction, unsafe Content-Disposition Implementation, etc. In our model, the file overwrite is an important DataOut property of a gadget.

Dynamic code loading problem and its security implications are researched in StaDyna (Zhauniarovich et al. 2015), Dydroid (Qu et al. 2017), StaDART (Ahmad et al. 2020), and so on. A dynamic code loading functionality is regarded as a strong gadget under our model if the loaded code is controllable by the attacker. This bug pattern can be potentially leveraged to execute arbitrary code by an attacker in the context of the affected application.

Future research plans

AppChainer is only a small step towards modeling the chain building and breaking the "case by case" prejudice. There are lots of challenges left for us and the whole community to solve. We hereby give some directions for future works.

Firstly, during measuring a gadget in this work, we only modeled a limited range of IO behaviors to chain various payloads, including static values, files, shared preferences, databases, and Intent IPC. There are a lot of other input and output behaviours of a gadget beyond our endeavors, e.g., a finer-grained local variable load-store operation. The model can be adequately improved in the future.

Secondly, a dynamic tool is necessary for generating reproducible chains both inside and between applications. Fuzzing and symbolic execution techniques are proven to be effective in generating inputs dynamically. Currently, the AppChainer prototype system proposed by us is a pure static tool for finding potentially chainable payloads. The confirmation of chained payloads requires experienced manual work. Automatic confirmation of the identified chains will help a lot in the acceleration of exploit chain building for security researchers and the identification of hidden attack surfaces beneath applications for developers.

Thirdly, AppChainer does not model the permission and privilege mechanisms in Android. Android has multiple permission levels and privilege domains. A sophisticated attack may go across multiple domains and gather the privileges step by step via various payloads. Therefore, an exploit chain model that considers permission and privilege will be complete.

Conclusion

With stricter mitigation measures and isolated privilege models applied to modern Android operating system, a single bug exerts a limited influence on the whole device. Despite the amount of works on bug patterns, few of the researchers manage to generalize a model for exploit chains involving multiple bugs or features.

This paper proposes a model for building exploit chains on the Android platform. A prototype system named AppChainer is designed and implemented to detect chainable payloads. Experimental results on a large number of popular Android applications show that AppChainer is able to chain payloads either inside

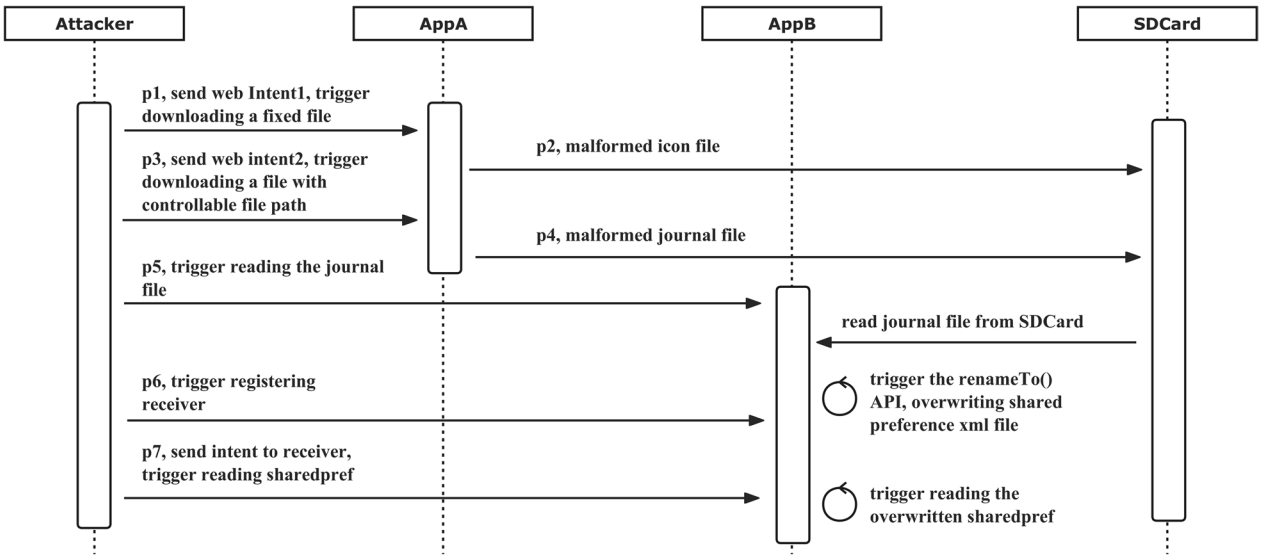

Fig. 8 The exploit chain in two real world applicatons found by AppChainer

one application or among multiple applications to trigger more functionalities than an attacker can originally achieve.

This work is one step towards breaking the prejudice that logic bug exploit chains are built in a case-by-case way. However, future research are ought to be conducted to continuously gain a finer-grained model.

Appendix

In this section, we explain one identified chain with 7 payloads across two applications. The workflow of the exploit is shown in Fig. 8.

In the chain, an attacker sends two web Intents to AppA to trigger two file-download functionalities. The first web intent triggers downloading a fixed-name icon file, while the second one can be leveraged to download a file with an arbitrary file name. The application uses unsafe HTTP connections to download the two files. Thus an attacker can perform MITM and tamper with the file content. Besides, the file name in the second web intent is not sanitized, allowing the attacker to perform a path traversal attack and write arbitrary file under the context of AppA. As shown in the figure, p1 triggers downloading a fixed-name file, p3 triggers downloading a file with a controllable file path, p2 and p4 are the corresponding malformed files.

There is another arbitrary file-overwrite bug leveraging the *File.renameTo()* API in AppB. AppB implements a Latest Recently Used (LRU) cache mechanism for loading and storing its Bitmap images. The LRU cache reads a Journal file line by line from the external storage, and parses each line as a command to perform corresponding behaviors. When the application parses the "CLEAN cleankey" and "DIRTY dirtykey" commands, the application constructs two file names with the two keys, and moves the clean file to the dirty file. As the journal file is stored in the SDCard, attackers can put a malformed journal file with "CLEAN" and "DIRTY" commands to move a file to a destination. In addition, when the application contructs the file path, no path sanitization is performed on the cleankey and dirtykey. If an attacker adds "../" string at the beginning of the two keys, path traversal occurs, which eventually allows an attacker to overwrite arbitrary files under the context of AppB. In this chain, p4 is the malformed journal file that points the cleankey to the malformed icon file.

The *File.renameTo()* API won't be executed directly until the attacker sends the p5 intent to AppB. The p5 starts the exported MainActivity and triggers the file overwriting operation.

AppChainer detects multiple candidate file-reading operations in AppB. We take a shared preference file as an example due to its longer attack paths. The loading of the shared preference file is conducted in a broadcast receiver of AppB, but the receiver is not launched by default. The attacker should first trigger the registration of the receiver with p6, and then trigger the file-loading operation with p7. Unfortunately, we are not able to perform a further powerful attack with the chain after modifying the files in AppB after manually analyzing it.

Acknowledgements
I would like to express my gratitude to all mates who support ideas and advice during the writing of this paper, and thank the anonymous reviewers for their valuable comments.

Author Contributions
XX proposed the model, implemented the prototype system, and wrote the manuscript. YJ helped to develop the prototype system and conducted careful experiments. QG, XZ, XG and BL reviewed the manuscript and gave suggestions on the revision of the details of the article. All authors read and approved the final manuscript.

Funding
This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (No. XDC02040100).

Availability of data and materials
Not applicable.

Declarations

Competing interest
The authors declare that they have no competing interests.

Received: 14 December 2022 Accepted: 9 March 2023
Published online: 02 August 2023

References

Aldoseri A, Oswald D (2022) insecure://vulnerability analysis of URI scheme handling in android mobile browsers. In: Proceedings of the workshop on measurements, attacks, and defenses for the web (MADWeb)
Au KWY, Zhou YF, Huang Z, Lie D (2012) Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217–228
Bagheri H, Kang E, Malek S, Jackson D (2018) A formal approach for detection of security flaws in the android permission system. Form Asp Comput 30:525–544
Bagheri H, Kang E, Malek S, Jackson D (2015) Detection of design flaws in the android permission protocol through bounded verification. In: International symposium on formal methods. Springer (Veranst.), pp. 73–89
Buchanan E, Roemer R, Savage S, Shacham H (2008) Return-oriented programming: exploitation without code injection. Black Hat 8
Buildfile (2022) Mobile app download statistics & usage statistics (2022) https://buildfire.com/app-statistic. – Zugriffsdatum. Accessed 16 June
Chen L, Liu X, Ma T, Shi CC, Li NG (2016) Research on static analysis technology of android application security defects. In: Proceedings of the international conference on electrical engineering and automation, pp. 113–119
Choi K, Ko M, Chang B-M (2018) A practical intent fuzzing tool for robustness of inter-component communication in android apps. KSII Trans Internet Inf Syst TIIS 12(9):4248–4270
Dawn Security Lab (2022) Mystique in the house: the droid vulnerability chain that owns all your applications. https://dawnslab.jd.com/mystique-paper/mystique-paper.pdf. – Zugriffsdatum: Accessed 16 June 2022
Demissie BF, Mariano C, Shar LK (2020) Security analysis of permission re-delegation vulnerabilities in Android apps. Empir Softw Eng 25(6):5084–5136
Demissie Biniam F, Ceccato M (2020) Security testing of second order permission re-delegation vulnerabilities in android apps. In: Proceedings of the IEEE/ACM 7th international conference on mobile software engineering and systems. Association for Computing Machinery (MOBILESoft '20), New York, pp. 1–11. https://doi.org/10.1145/3387905.3388592. ISBN 9781450379595
El-Zawawy MA, Eleonora L, Mauro C (2021) Do not let Next-Intent Vulnerability be your next nightmare: type system-based approach to detect it in Android apps. Int J Inf Secur 20(1):39–58
Elgharabawy M, Kojusner B, Mannan M, Butler KB, Williams B, Youssef A (2022) SAUSAGE: security analysis of unix domain socket usage in android. In: 2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P). p 572–586
Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E (2011) Permission re-delegation: attacks and defenses. In: *USENIX security symposium*, vol. 30, pp. 88
f-secure Lab (2022) *Xiaomi Mi9 (Pwn2Own 2019)*. 2019. https://labs.f-secure.com/advisories/xiaomi-mi9/. Zugriffsdatum: Accessed 16 June 2022
Gao J, Li L, Kong P, Bissyandé TF, Klein J (2018) Poster: on vulnerability evolution in android apps. In: 2018 IEEE/ACM 40th international conference on software engineering: companion (ICSE-Companion) IEEE (Veranst.), pp. 276–277
Gao X, Tan SH, Dong Z, Roychoudhury A (2018) Android testing via synthetic symbolic execution. In: 2018 33rd IEEE/ACM international conference on automated software engineering (ASE) IEEE (Veranst.), pp. 419–429
Garg S, Baliyan N (2020) Machine learning based android vulnerability detection: a roadmap. In: International conference on information systems security. Springer (Veranst.), pp. 87–93
Geshev G, Miller R (2018) Chainspotting: building exploit chains with logic bugs. https://labs.f-secure.com/archive/chainspotting-building-exploit-chains-with-logic-bugs/. Zugriffsdatum: Accessed 16 June 2022
Google (2022) File observer - android developers. https://developer.android.com/reference/android/os/FileObserver. Zugriffsdatum: Accessed 16 June 2022
Google (2022) PendingIntent | Android Developers. https://developer.android.com/reference/android/app/PendingIntent. Zugriffsdatum: Accessed 16 June 2022
Gorski III Sigmund A, Enck W (2019) Arf: identifying re-delegation vulnerabilities in android system services. In: Proceedings of the 12th conference on security and privacy in wireless and mobile networks, pp. 151–161
Groß S, Tiwari A, Hammer C (2018) Pianalyzer: a precise approach for pendingintent vulnerability analysis. In: European symposium on research in computer security. Springer (Veranst.), pp. 41–59
Hyunwoo C, Yongdae K (2018) Large-scale analysis of remote code injection attacks in android apps. Secur Commun Netw. https://doi.org/10.1155/2018/2489214
III Sigmund Albert G., Thorn S, Enck W, Chen H (2022) FReD: identifying file re-delegation in android system services. In: 31st USENIX security symposium (USENIX Security 22).USENIX Association, Boston, pp. 1525–1542. https://www.usenix.org/conference/usenixsecurity22/presentation/gorski. ISBN 978-1-939133-31-1
Initiative Zero D (2022) Pwn2Own Miami 2022 Rules. https://www.zerodayinitiative.com/Pwn2OwnMiami2022Rules.html. Zugriffsdatum: Accessed 16 June 2022
JoMing Y (2022) *Google Play Scraper*. https://github.com/JoMingyu/google-play-scraper. Zugriffsdatum: Accessed 16 June 2022
Lab f-secure (2019) Automating Pwn2Own with Jandroid. https://labs.f-secure.com/blog/automating-pwn2own-with-jandroid. Zugriffsdatum: Accessed 16 June 2022
Lerch J, Hermann B, Bodden E, Mezini M (2014) FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, pp. 98–108
Linares-Vásquez M, Bavota G, Escobar-Velásquez C (2017) An empirical study on android-related vulnerabilities. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), pp. 2–13
Liu F, Wang C, Pico A, Yao D, Wang G (2017) Measuring the insecurity of mobile deep links of android. In: 26th USENIX security symposium (USENIX Security 17), pp. 953–969
Lu L, Li Z, Wu Z, Lee W, Jiang G (2012) Chex: statically vetting android apps for component hijacking vulnerabilities. In: Proceedings of the 2012 ACM conference on Computer and communications security, pp. 229–240
Luo L, Zeng Q, Cao C, Chen K, Liu J, Liu L, Gao N, Yang M, Xing X, Liu P (2019) Tainting-assisted and context-migrated symbolic execution of android framework for vulnerability discovery and exploit generation. IEEE Trans Mob Comput 19(12):2946–2964
Maqsood A, Valerio C, Bruno C, Francesco B, Yury Z (2020) StaDART: addressing the problem of dynamic code updates in the security analysis of android applications. J Syst Softw 159:110386
Min Z, Haimin Y, Ping C, Zhengxing Y (2019) Android software vulnerability mining framework based on dynamic taint analysis technology. In: 2019 IEEE 3rd information technology, networking, electronic and automation control conference (ITNEC) IEEE (Veranst.), pp. 2112–2115
Owasp (2022) OWASP mobile top 10. https://owasp.org/www-project-mobile-top-10/. Zugriffsdatum: Accessed 16 June 2022
Plaskett A, Loureiro J (2018) The mate escape. https://labs.f-secure.com/archive/the-mate-escape-huawei-pwn2owning/. Zugriffsdatum: Accessed 16 June 2022

Qu Z, Alam S, Chen Y, Zhou X, Hong W, Riley R (2017) Dydroid: measuring dynamic code loading and its security implications in android applications. In: 2017 47th annual IEEE/IFIP international conference on dependable systems and networks (DSN). IEEE (Veranst.), pp. 415–426
Sherman M (2014) Attack surfaces for mobile devices. In: Proceedings of the 2nd international workshop on software development lifecycle for mobile, pp. 5–8
Steven A, Siegfried R, Christian F, Eric B, Alexandre B, Jacques K, Yves LT, Damien O, Patrick MD (2014) Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Not 49:259–269
Wang R, Xing L, Wang XF, Chen S (2013) Unauthorized origin crossing on mobile platforms: threats and mitigation. Association for Computing Machinery, (CCS '13), New York, pp. 635–646. https://doi.org/10.1145/2508859.2516727. ISBN 9781450324779
Yang K, Zhuge J, Wang Y, Zhou L, Duan H (2014) IntentFuzzer: detecting capability leaks of android applications. In: Proceedings of the 9th ACM symposium on Information, computer and communications security,pp. 531–536
Ye H, Cheng S, Zhang L, Jiang F (2013) Droidfuzzer: fuzzing the android apps with intent-filter tag. In: Proceedings of international conference on advances in mobile computing & multimedia, pp. 68–74
Zhang C, Li S, Diao W, Guo S (2022) PITracker: detecting android pendingintent vulnerabilities through intent flow analysis. In: Proceedings of the 15th ACM conference on security and privacy in wireless and mobile networks, pp. 20–25
Zhang H, Li Z, Shahriar H, Lo D, Wu F, Qian Y (2019) Protecting data in android external data storage. In: 2019 IEEE 43rd annual computer software and applications conference (COMPSAC), vol. 1, pp. 924–925
Zhauniarovich Y, Ahmad M, Gadyatskaya O, Crispo B, Massacci F (2015) Stadyna: addressing the problem of dynamic code updates in the security analysis of android applications. In: Proceedings of the 5th ACM conference on data and application security and privacy, pp. 37–48

Publisher's Note