Efficient fuzzing payload generation for mobile application security testing

Anton Helin

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.
Espoo 27.4.2024

Supervisor

Dr. Lachlan J. Gunn

Advisor

MSc Topi Toosi

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Anton Helin

Title Efficient fuzzing payload generation for mobile application security testing

Degree programme Master's Programme in Computer, Communication and Information Sciences

Major Computer Science **Code of major** SCI3042

Supervisor Dr. Lachlan J. Gunn

Advisor MSc Topi Toosi

Date 27.4.2024 **Number of pages** 58 **Language** English

Abstract

The topic of this Master's thesis focuses on developing efficient fuzzing payloads for security testing of Android and iOS mobile applications. Fuzzing is a technique which has mostly been used to detect crashes, errors and performance issues within an application by using random and unpredictable inputs. Modern security issues are on a higher abstraction level and require more specific inputs to be discovered and exploited. Many extremely efficient algorithms for fuzzing have been developed in the past, but the entropy is too large for them to be efficient, if purely random inputs are generated. This has led to web application fuzzer tools which have specialized inputs for finding specific target problems. The outcome of the research includes a method, which produces and selects payloads to use for fuzzing a mobile application. The variables that affect these are the input formats of the target app and the vulnerabilities that want to be searched for. Depending on the available static analysis and the user, their accuracy can be adjusted. Examples and evaluation of it's use in practice have been included for several important vulnerability types. The main problem that was solved consisted of constructing input languages of mobile applications in context-free grammar format and optimizing the generation of payloads from the grammars. As Android applications are Java-based, fuzzing payloads had to adhere to Java syntax in many cases, and serialized Java-classes are used to transmit data. One of the main improvements found in this research is generating payloads for Java programs which are specifically running as an Android application, leading to much more optimal use. This also does not require perfect knowledge of the source code of the application, so applications can be tested based on entirely public information on a regular Android device. This reduces the resources required for a testing environment.

Keywords Android, iOS, mobile application, security, fuzzing, programmatic testing

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Anton Helin

Työn nimi Tehokas fuzz-syötegeneraatio mobiilisovellusten tietoturvatestaukseen

Koulutusohjelma Master's Programme in Computer, Communication and Information Sciences

Pääaine Computer Science **Pääaineen koodi** SCI3042

Työn valvoja Dr. Lachlan J. Gunn

Työn ohjaaja MSc Topi Toosi

Päivämäärä 27.4.2024 **Sivumäärä** 58 **Kieli** Englanti

Tiivistelmä

Tämän diplomityön aiheena on tehokkaiden fuzz-testaukseen soveltuvien syötteiden kehittäminen Android- ja iOS-sovellusten tietoturvatestaukseen. Fuzz-testaus on menetelmä, jota on yleisesti käytetty sovelluksen kaatumisten, virheiden ja suorituskykyongelmien löytämiseen käyttämällä sattumanvaraisia ja ennalta-arvaamattomia syötteitä. Nykyiset tietoturvaongelmat ovat korkeammalla abstraktiotasolla ja niiden löytäminen ja hyväksikäyttäminen vaatii yksityiskohtaisempia syötteitä. Monia hyvin tehokkaita algoritmeja fuzz-testaukseen on kehitetty, mutta niiden entropia on liian suuri estääkseen tehokkaan käytön, jos generoidut syötteet ovat täysin sattumanvaraisia. Tämä on johtanut web-sovellusten fuzz-työkaluiden kehittämiseen, jotka käyttävät erityisiä syötteitä tiettyjen kohteena olevien ongelmien löytämiseksi. Tämän tutkimuksen tulos sisältää menetelmän, joka tuottaa ja valitsee syötteitä käytettäväksi mobiilisovelluksen fuzz-testaamiseen. Niihin vaikuttavat tekijät sisältävät kohdesovelluksen syötemuodot ja kohteena olevat tietoturvaongelmat, joita halutaan etsiä. Riippuen saatavilla olevasta staattisen analyysin mahdollisuudesta ja käyttäjästä, syötteiden tarkkuutta voidaan säätää. Esimerkkejä ja arviointia menetelmän käytöstä käytännössä on sisällytetty muutamalle tärkeälle haavoittuvuustyypille. Ratkaistu pääongelma muodostui syötekielien rakentamisesta mobiilisovelluksille kontekstivapaiden kielioppien muodossa ja niistä generoitujen syötteiden optimoinnista. Koska Android-sovellukset ovat Java-pohjaisia, testaussyötteiden on monissa tapauksissa noudatettava Javan syntaksia ja serialisoituja Java-luokkia käytetään sovellusten välisessä tiedonsiirrossa. Yksi tutkimuksen löytämistä parannuksista on syötteiden generointi Java-ohjelmille, jotka ovat nimenomaan Android-sovelluksia, mahdollistaen lähes optimaalisen testauksen. Tämä ei myöskään vaadi täydellistä tietoa sovelluksen lähdekoodista, joten sovelluksia voidaan testata perustuen täysiin julkiseen tietoon tavallisella Android-laitteella. Tämä vähentää testiympäristöön vaadittavia resursseja.

Avainsanat Android, iOS, mobiilisovellus, tietoturva, fuzz-testaus, ohjelmallinen testaus

Preface

I would like to thank my supervisor Lachlan J. Gunn for providing detailed and profound advice during the thesis process. The discussions about the structure of the thesis were very important for the quality of the thesis and have given me valuable experience. I would also like to thank my parents and my girlfriend Venla for motivation and support.

Otaniemi, 27.4.2024

Anton Helin

Contents

Abbreviations

HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
HTML	Hypertext Markup Language
TLS	Transport Layer Security
SSL	Secure Sockets Layer
OS	Operating System
JS	JavaScript
JSON	JavaScript Object Notation
API	Application Programming Interface
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
UID	Unique Identifier
UI	User Interface
BNF	Backus-Naur form
EBNF	Extended Backus-Naur form
SQL	Structured Query Language
XSS	Cross-Site Scripting
UTF	Unicode Transformation Format

1 Introduction

Information security has become a more and more important aspect of software development in recent years [1]. As applications are moved into the cloud and transformed into a more scalable and reliable form, secure development and testing is required to ensure data of users is not leaked and the applications are not misused or broken. Security is often in a tradeoff with user satisfaction and functionality, so enforcing it strictly is expensive and difficult.

Most web applications on the internet are based on a client-server model. In web applications, practically all functionality is either in a browser or a server. Browsers are developed with strict security concerns in mind and the vast majority of users use one of only a few popular browsers [2]. The application on the server is the main security concern for developers, as even client-side JavaScript security issues often come from content generated on the server. Nearly all web functionality uses the HTTP protocol for interacting with the server and content is shown using HTML and JavaScript. This has led to incredible progress in web security testing, as testing tools and fuzzers can be built on top of HTTP and JavaScript [3].

In mobile applications, the model is similiar, but the client is not only a browser. More functionality than in a web application can be developed on the client-side. While Android and iOS enforce many security controls, the advantages of having a mobile application in the first place is lost if security was equally strict as in browsers, because a user can also access a web application with a mobile browser.

Intuitively, this implies imply that mobile applications inherently carry more risk of a developer making a mistake leading to a security problem. Web applications are still tested much more [4] and have far more advanced tools, but mobile applications also use HTTP to connect to an API, so HTTP-based tools are a very important part of mobile security as well.

The differences in security testing come from the additional input points that mobile applications have combined with extended client-side functionality. Additional input points include communication with other applications on the same device, links opened by a user and processed by a mobile application and HTTP responses. Responses are considered additional inputs as they are used to deliver data into the application, and good security design principles include handling all external data as potentially malicious. HTTP response handling is important in web security as well, but since the client is a browser instead of a custom application, it again has clear security controls already in place.

These input points require specialized tooling to test, as they use varying protocols and environments. Automated web testing is done by fuzzing the HTTP requests. Efficient fuzzing requires very accurate manipulation of input, and the goal of this research is to construct a fuzzing method for inputs in mobile application context. Language-based fuzzing is a promising technique [5], which is used in this research to design fuzzing models for the known input formats of mobile applications.

This research focuses on well-defined, relatively easily detectable vulnerabilities with clear risks. Efficient payload generation, delivery and detection for such vulnerabilities will be useful in the case of any vulnerability testing. As the main goal is to

find a generalized, efficient system of payload generation, it is possible to extend it to essentially any vulnerability with similiar payload delivery as the vulnerabilities originally considered.

2 Background

2.1 Fuzzing

At its core, fuzzing is a dynamic software testing technique that involves supplying malformed or random data, known as ”fuzz” as inputs to a target application. The objective is to provoke unexpected behaviors or crashes in the software, which may indicate the presence of security vulnerabilities such as buffer overflows, memory leaks, or logic flaws.

Fuzzing operates on the principle of exploring the vast input space of an application, pushing it beyond its intended usage scenarios to reveal weaknesses that might otherwise go unnoticed. By generating a diverse range of inputs, including valid, invalid, and unexpected data, fuzzing effectively stress-tests software and exposes its susceptibility to exploitation.

Many different approaches to fuzzing exist. Reviewed in [6], fuzzing input generation can be either mutation-based or generation-based. The difference is that mutation-based fuzzing has one seed, which is then mutated to different forms, and generation-based fuzzing has some systematic set of rules which inputs are generated from. Grammars are one such systematic set of rules. Constraint-based fuzzing [7] extends this by enabling generation conditions that cannot be achieved by grammars. More advanced approaches include concolic fuzzing [9], collecting program constraints during execution and describing them as relations, and symbolic fuzzing [8], which tries to determine possible execution paths mathematically without executing the program.

The fuzzing process itself can focus on different goals. Coverage-based fuzzing tries to find many possible inputs to a program, so that many program execution paths can be found. Search-based fuzzing focuses on searching for particular conditions by forming inputs that can pass through multiple conditions in the execution path.

In the context of web applications, mobile applications and any other high level functionality, the inputs being fuzzed must be in some known structure or format. The components of that structure are modified when fuzzing. For example, to deliver a malicious link that is processed by the application might require bypassing host validation, which then may require mutating different parts of a valid URL or a serialized Java URI object in the payload. If the input data is just random bytes, it will nearly always get rejected because it is not a URL. Fuzzing algorithms must not have high entropy and must follow a certain structure. Additionally, applications have known vulnerabilities and security issues, which can be triggered by malicious inputs. These are targets for search-based fuzzing. When fuzzing purely with random data, there is no payload for a vulnerability and it is not clear what the target of the fuzzing is, besides causing a crash. Currently, testers create these structured malicious inputs for mobile applications mostly manually, which is not efficient.

2.2 Mobile Application Structure

Android apps typically consist of a manifest file defining essential app details, Java or Kotlin source code for logic, XML layout files for UI design, and resources like images and strings. Activities, fragments, services, broadcast receivers, and content providers compose the app's components.

In contrast, iOS apps are structured around a project file containing Swift or Objective-C source code for logic, Interface Builder files for UI design, and an Assets Catalog for managing resources. App Delegate manages lifecycle events, while view controllers handle UI interactions, and models encapsulate data and business logic.

2.2.1 Java

Java [15] is a programming language used to develop Android applications. The Java virtual machine executes bytecode, which Java is compiled into. Languages such as Scala and Kotlin also compile into JVM bytecode, so they are compatible with Java. Android systems also convert JVM bytecode into Dalvik bytecode, formatted in .dex-files. It is a format designed for systems with large constraints in speed and memory use, such as mobile devices. Applications are programmed with Java and by using the Android API, they can be run in a Dalvik virtual machine. The Dalvik VM has been replaced by Android Runtime in newer versions, but .dex-files are still used for backwards compatibility of applications [14]. Kotlin has also started to gain popularity in Android application development [12].

Due to the inherent security of the Java virtual machine, it protects the host system from crashing or memory corruption. Isolation of code execution from the operating system has been a core security principle in designing the JVM and Android and because of it, applications usually do not impose similiar security risks as C programs do. Isolation, however, comes with the cost of usability. Mobile applications must be flexible and have many different functionality involving inter-process communication. Security research of the Android operating system is a very large and complex topic in itself, as applications must interact with the underlying Linux-based system, but still remain as isolated as possible.

Application-level security concerns are related to user data and application integrity, which can be compromised without breaking the security of the operating system. In principle, no inherent security structure of Java or Android can protect sensitive data if the application developer has not taken the proper measures to handle it. Many mechanisms exist in the Android API for transferring data and using sensitive functionality, but the developer can misuse them by accident or implement dangerous functionality themselves.

Java is an object-oriented programming language, and it uses classes to implement objects. An object is always an instance of a class, which has well defined methods and variables. Classes are often defined as subclasses of a parent class, which inherit the methods of the parent class and can have their own implementation of a specific method. Java is statically typed, so the type of each variable is checked to match an intended definition when the code is compiled. This can reduce errors

by automatically finding them already in development and makes the design of applications clearly structured.

2.2.2 Data transfer in Android applications

The API involved with data transfer between applications uses the Intent Java class. It wraps data in a format which the Android system and applications all expect, so that communication can be optimized and done in a secure manner. Intent structure is a very important aspect of application security, because Intents contain the entry point for malicious data coming from other applications and the opening of links in Android is implemented using them as well [16].

Many applications receive data in the form of a Java object. Since Java is statically typed, it's type must be exactly same in both the sending and the receiving application, which must define what type of object they assign to a variable before inserting it into an Intent and reading it from a received Intent. Intents usually contain data in the form of extras and a data URI. Extras can be primitive data types or a Bundle object, which implements the Parcelable class. They can also include Parcelable classes by themselves or classes implementing the Serializable class. These data formats are the application level input to an Android application and fuzzing them requires the format to remain valid. Generally, the fields of these Java classes are the only changing part of the fuzzing payloads, as changing the class itself results in incompability. There are some exceptions to this, which can make fuzzing easier and more efficient.

Implementing the Parcelable class allows objects to be stored as a Parcel object. A receiving application can then reconstruct the object from the Parcel, thus requiring only the Parcel type object to be the implemented in the same way in both applications. The Parcel must still include very specific data to properly reconstruct an object. Every class implementing Parcelable must define the *writeToParcel* function, which determines how the object is written into a Parcel. Only the following primitive data types can be written into a Parcel:

1. byte
2. double
3. float
4. int
5. long
6. string

Intents can also include short, char and boolean, but they are also very simple to fuzz. The complete name of the class will also be included in the Parcel. Packing data into primitive data types is a considerable advantage in testing application inputs, as it is possible to send an input to an application without fully implementing the same

classes as that application does. Only the name of the class and the collection of the primitive type variables written into the Parcel according to the *writeToParcel* function must match that of the target application's class.

Serialization is a technique where a Java object is transformed into raw byte data. This has been used in web applications to transmit data in a simple format, but has caused many security issues to rise. If the data is not properly validated, it can be used to construct Java classes in the application that were not originally meant to be constructed, allowing an attacker to access functionality or data from outside the intended context [28]. In the most severe cases, this has lead to arbitrary code execution vulnerabilities on web applications. In Android, similiar risks exist although the inherent structure of the Android API does mitigate them. The serializable format is used to transmit data between Android applications, and this can enable an attacker to force an app to initialize sensitive classes and functionality. This is made easier by the fact that Android application code can be decompiled into clear enough Java so that serialized classes can be very easily identified.

The Serializable interface allows Java to transform an object into serialized byte data. The transformation is handled automatically for any class implementing Serializable. The class to be serialized can implement the methods *writeObject* and *readObject* to write data directly into the byte stream. Methods *writeReplace* and *readResolve* can be used to write an alternative object to the stream if needed. A field containing a unique identifier *serialVersionUID* can optionally be included to confirm compatibility of serialized objects, and if not included it is generated from the class. Only the fields of the class and their types are serialized, and if the parent class is not serializable, it is restored using a default constructor with no arguments. The classes of the fields to be serialized must also implement the Serializable interface or use the transient keyword to indicate that they should not be serialized along with the class. During deserialization, they are given a null value. To fuzz a serializable class in Android application level, more work may be required than in the case of Parcelable classes, as serialization can have many other classes as serializable fields in addition to the primitive data types.

2.3 Mobile Application Testing

Security analysis of a mobile application consists of static analysis, dynamic analysis and the translation of results into practical exploits. Fuzzing is a part of dynamic analysis, and it's purpose is to complement findings of static analysis and help in constructing a practical attack for a security flaw. The approach of this topic is in grey-box testing, where an app is tested as a black-box but with possible knowledge of the source code, usually only what is possible to decompile from the application file. It allows for more generalized methods and efficiency, and allows the testing of an application without the developer having to leak the source code to the tester. It also simulates an attacker trying to find security issues in a published app. Static analysis is however not very easy in grey-box testing, because iOS apps are binaries which are difficult to reverse engineer [10]. Android apps can be decompiled into Java, but it takes a large amount of work to analyze input flow properly [11]. A good

fuzzing method is still be applicable to white-box testing with even better efficiency.

2.3.1 Vulnerabilities

Mobile application vulnerabilities are often closely related to web application vulnerabilities. They evolve contiously and new attacks are found. The OWASP Foundation (Open Worldwide Application Security Project) maintains a list for the top 10 biggest risks in mobile applications [19]. Many of these risks are related to configuration or aspects of development where fuzzing is not helpful. However, in 2024, insufficient input/output validation is listed as the 4th biggest risk. This will be one of the main problems which fuzzing can help with, as malicious inputs are sent to the application. The vulnerability usually occurs as an injection, where user input is misinterpreted as a command instead of it's intended data format. The attack and it's severity depend on which context this misinterpretation occurs, and some of the most common examples are included in our fuzzing payload construction. In mobile application context, the payloads used are similiar to common web vulnerabilities, but the method of exploitation and severity differ. Many of these vulnerabilities have been researched and documented by a company called Oversecured [22].
These include the following vulnerabilities:

1. SQL injections [29]: a database command is injected. This typically occurs when the application considers the character ' in SQL syntax, which closes an inserted data string and interprets the content after the character as SQL commands.

2. Cross-Site Scripting and Cross-Application Scripting [30]: injected JavaScript is executed. In web applications, this occurs when user input includes JavaScript that is embedded into HTML context. It can also occur in a link using the *javascript* protocol [31] instead of *https*. In mobile applications, HTML content and JavaScript are commonly used in WebView components, so similiar risks exist.

3. Command and code injections [34, 26]: an injected operating system command or application code is executed. This is specific to the operating system and techniques used to develop the application. Can occur if operating system commands are used directly instead of secure application interfaces, if binary files are inserted or code is dynamically loaded into the application.

4. Path traversal and file access [32, 27]: file paths used by the application are not restricted properly. This allows the manipulation of the file system by inserting a string such as *../* into paths, indicating the parent directory. The full path or a *file* protocol link to sensitive application files can also be included and all cases should be verified as safe by the application.

5. Open redirection [33]: application opens a malicious link. All links handled by the application should be properly verified to contain a trusted host and valid parameters. Web content opened by an application is likely to be trusted by a

user, increasing the risk of sensitive data leaks. HTML content loaded from unsafe resources can enable Cross-Site Scripting attacks.

In addition to these vulnerabilities, there are several Android-specific vulnerabilities which are included in this research. They are more likely to bypass input validation, because developers may not know about them. Fuzzing can be used to prove the existence of such vulnerabilities and bypass simple mitigation techniques. Static analysis of the application is required to find potentially exploitable application components, and access to the component is configured into the payloads so that fuzzing is efficient and likely to find a vulnerability.

1. Information leakage in returned Intents: manipulation of data returned by the application. Intents are the main form of communication between Android applications. Applications can respond to Intents by sending an Intent back. This can lead to information disclosure or unintended permission grants, if the returning Intent is manipulated [23, 24].

2. Access to protected components: injected Intents are used to launch application components. If the application redirects Intents or generates Intents from external input, it may be possible to access sensitive application functionality [25].

3. Java deserialization: Java classes with sensitive functionality are unserialized. Serialized Java classes are unserialized when an Intent is processed, and if the application includes a class with functionality exploitable only by the creation of the class, it can lead to high severity vulnerabilities such as memory corruption [28, 23].

In general, vulnerabilities can come in different forms and in combinations of less severe vulnerabilities, which can increase in severity under certain conditions. Searching for vulnerabilities generally known to exist in mobile applications is a good starting point and it can also help testers find unintended behaviour and weaknesses in code logic.

Applications may also have vulnerabilities that are specific to their functionality and business logic. Banking applications have different security requirements than a healthcare application, and mistakes in application code may introduce vastly different risks. These type of vulnerabilities are harder to find using fuzzing and won't be directly considered in this research. Fuzzing for such vulnerabilities requires much more application specific configuration, so it is less efficient. There is also no easy way to detect such a vulnerability programmatically, because the risks and impact are not as clearly defined.

There are also technology-specific vulnerabilities such as vulnerabilities related to cryptography implementations in mobile applications. Fuzzing has proved to be a very efficient tool in finding cryptographic flaws, but require fuzzing techniques that have been developed specifically for cryptographic flaw detection [35]. In this research, cryptography is not considered as it is a complex and large topic of its own. Cryptography in mobile applications is also usually implemented using

existing libraries, in which case the flaws on the application level are configuration mistakes, where fuzzing may not be helpful. Testing of libraries is not in the scope of this research, as they are not directly mobile application related. Applications requiring specific use cases or potentially higher security may implement cryptographic algorithms themselves, and requires a different testing method depending on the cryptographic algorithms used. The fuzzing methods developed for the more generic vulnerabilities listed in this section can be used for testing technology specific vulnerabilities, but it requires creating a new configuration for each technology. However, delivery of the fuzzing payloads is a problem generally considered by our research, which will be useful in such cases.

2.3.2 Static Analysis

The analysis of any software program can be divided into two different approaches: static and dynamic analysis. Static analysis refers to any action not involving running the application program, such as examining the source code or compiled code of the program. Dynamic analysis refers to examining the program while it is being executed. Static analysis can reveal clear errors in programming and configuration, but dynamic analysis can be used to find unintended behaviour that is not easily detectable from complex source code. A good testing methodology uses both approaches together and they complement each other well. In this section, the mobile application context of both approaches to analysis are explained along with their relevance for this research.

Static analysis is a crucial component of the overall security assessment and quality assurance process for mobile apps. It can be conducted either manually by human analysts, which try to find flawed patterns in application code by going through it line by line, or by automated tools, scanning the code for potential issues. The depth and usefulness of static analysis depends on the available source code. The source code of an application is often very sensitive information that developers do not want to share, but not sharing it to external security analysts makes the analysis less efficient. However, bug bounty programs where completely arbitrary security professionals can conduct analysis on applications and report vulnerabilities, have gained popularity recently [36]. These analysts only have access to public information and applications downloaded from official app stores. It is therefore extremely useful to have tools which can statically analyse an application without being able to access the source code. In the case of mobile applications, the application code must be on the device in some form and can therefore be analysed.

In iOS, applications are stored on the device as encrypted binary files, which only include native processor instructions. Unencrypted binaries may be supplied by developers, but encryption can still be trivially bypassed, as it is tied to an iTunes account and devices can be simulated. The attempt to transform binary code into original source code is referred to as reverse engineering, and there are many tools such as Ghidra, developed by NSA [41] for that purpose. OWASP [20] provides useful information on iOS application reverse engineering. Perfect analysis of binary code is extremely resource intensive and difficult, but data strings, class information and very specific functionality can be analyzed. Finding vulnerabilities automatically

is not likely in the case of iOS binaries.

Static analysis of Android applications is entirely different than in the case of iOS. The main reason for this is that applications run in Java Virtual Machine, which is further explained in section 2.2.1. Applications on the device are generally in Java bytecode instead of binaries, which is far easier to reverse engineer. Java bytecode can be decompiled into readable Java code. Class, method and variable names may be obfuscated, so it may be difficult to determine the purpose of different parts of the code. However, Android API calls in Java are difficult to obfuscate. This follows from the fact that the application is developed using the large Android Java library, so API calls are inside well-known classes and structures within the library. The class methods involved with API calls in the code can thus be trivially identified. Many tools utilizing static analysis can successfully bypass obfuscation [46]. Critically for security analysis, many security vulnerabilities are related to unsafe Android API calls and the section of application code that is doing them. Obfuscation does not stop Java logic from being readable, it simply makes it more difficult to find interesting sections of the code, whereas in iOS and binaries in general, both of these are difficult tasks. Configuration of Android applications must also define exported components of the application in a manifest file, allowing easier identification of external data entry points in the application. Another notable thing about static analysis is that data strings cannot be obfuscated either, since they need to be used in the application in a specific format. To perform host validation on a URI, the valid hosts must be included in the application as readable cleartext strings. Considering these facts, static analysis is very useful in the analysis of Android applications and obfuscation is not a valid security measure. Many applications are not obfuscated at all, with around 25% of applications using some form of obfuscation [47].

2.3.3 Dynamic Analysis

Dynamic analysis is usually conducted using a variety of tools that monitor the execution of the mobile application in real-time. They track various activities such as file system operations, network requests, API calls, and interactions with device sensors (GPS, camera, microphone, etc.). This monitoring helps identify any abnormal or potentially malicious behavior exhibited by the application.

Dynamic analysis often involves capturing and analyzing network traffic generated by the mobile application. This includes inspecting HTTP/HTTPS requests and responses to identify potential security issues such as insecure communication, data leakage, or unauthorized data transmission. One such tool is BurpSuite [43], which can also be used to modify the requests and responses.

Some dynamic analysis tools utilize techniques like dynamic instrumentation to inject code into the running application for real-time monitoring and analysis. This allows auditors to inspect the application's runtime behavior, identify security flaws, and trace the execution flow to uncover many different security vulnerabilities. Frida [42] is a toolkit for many different operating systems that can be used to hook into processes and modify runtime behaviour.

Generally, dynamic analysis involves profiling the application's normal behavior

to establish a baseline. Any deviations from this baseline during runtime can indicate potential security threats or anomalies that warrant further investigation. Behavioral profiling and anomaly detection techniques help identify suspicious activities and application states.

2.4 Languages and Grammars

Languages are used to define the syntax and structure of programming languages and mathematical expressions. They are useful in fuzzing methods, because they can be used to model input languages for programs in a way that their structure can be perfectly preserved when testing different inputs. Basic definitions of languages from [13] are introduced here.

Definition 2.1 *An **alphabet** is a finite nonempty set of symbols. **Symbols** are unique and not further divisible into other symbols.*

Definition 2.2 *A **string** over an alphabet Σ is a finite sequence of symbols of Σ. The length $|x|$ of a string x is the amount of symbols contained in it. The notation for an empty string with length 0 is defined as ϵ for convenience. The **concatenation** of two strings $x = a_1a_2...a_n$ and $y = b_1b_2...b_m$ is denoted as $xy = a_1a_2...a_nb_1b_2...b_m$. The concatenation of x with the empty string ϵ results in the string x itself. Concatenating the same string x by itself n times is denoted by x^n.*

Definition 2.3 *The set of all strings over an alphabet Σ is denoted by Σ^*. The set of all nonempty strings is denoted by Σ^+. The empty set of strings is denoted by $\emptyset$. A **language** over Σ is defined as a set of strings of Σ, and the strings of a language are referred to as **words**. The concatenation of languages L_1 and L_2 is denoted by L_1L_2 and defined as the language $\{xy|x \in L_1, y \in L_2\}$.*

Languages are more useful when described in a way that it is possible to distinguish strings belonging to them. Finite languages can be described by listing it's elements and infinite languages can have rules defining it's elements clearly. Languages described by systematic rules can be made to include certain properties, which are helpful in many cases. *Grammars* are one of the systems used for describing languages and has been considered to be the most useful option.

Definition 2.4 *A **grammar** is defined as the quadruple (Σ, V, S, P), where the following hold: Σ is the **terminal alphabet**, an alphabet containing symbols referred to as **terminals**. V is a finite nonempty set disjoint from Σ. The elements of V are symbols defined as **nonterminals**. $S \in V$ is a nonterminal defined as the **start symbol**. P is a finite set containing **productions**, denoted by $\alpha \longrightarrow \beta$, where α and β are strings over terminals and nonterminals and α must contain at least one nonterminal.*

Languages can be generated by grammars. The language forms from the productions of the start symbol resulting in a subset of Σ^*, the set of all strings of terminal

symbols. These words can be referred to as *sentences*. An example of this is the English language, where terminal symbols are letters and grammatical terms such as verbs and nouns are nonterminals. Final, constructed sentences only contain letters, but they are built using productions of grammatical terms and letters combined. A start symbol *sentence* can be a *noun* consisting of *letters*, and the resulting set of all nouns only includes strings of letters.

In the context of programming languages, the language consists of valid structures such as a the declaration of a Java variable:

```
int a = 5;
```

Here, the terminals correspond to `int`, `a`, `=`, `5` and `;`. The nonterminals, not included in a valid "sentence" of a programming language, consist of *<intAssignment>*, *<variable>* and *<integer>*. The whole expression can be described as the production

<intAssignment> $\longrightarrow$ `int` *<variable>* `=` *<integer>*`;`.

This is a simple example and real Java is much more complex, especially since types can be any class, which are themselves complex expressions.

Programming languages are obviously very different from natural languages. This can be explained with the Chomsky hierarchy [17], which sets different types of restrictions on grammars. The types of grammars that we are the most interested in are *context-free grammars*.

Definition 2.5 *A grammar G is a* ***context-free grammar*** *if for each production $\alpha \longrightarrow \beta$ in P it holds that $|\alpha| = 1$. A* ***context-free language*** *is a language formed from a context-free grammar.*

This definition restricts α to only include one nonterminal. The consequence is that the grammar will have a tree structure, drastically simplifying the analysis of the grammar and generation of sentences. The types of grammars can be classified as how they relate to *Turing machines*, but only definitions of context-free grammars are relevant to this research, so they are not fully included. The reason context-free grammars are used for programming languages is because they are the simplest grammar that is recognized by *pushdown automata* [18]. Informally, this means that an input, the current state and the top of a stack can be used to determine the next state. A symbol can be pushed to the top of the stack or removed from the top of the stack. A Turing machine can have unrestricted access to written symbols, not only the topmost symbol of a stack. The pushdown automation model is thus more restricted than a Turing machine, but it can still be used to model compilers and parsers, thus reducing the requirements of their implementations. Programming languages can be considered inputs for compilers.

Fuzzing payloads are inputs to programs. By modeling the input as context-free grammar, all fuzzing payloads generated from the grammar will satisfy the valid structure expected by a parser. Ideally, when errors are found, the error always results from the parser, as it fails to parse an input it should be able to. Grammars also enable the construction of highly complex but valid payloads while still being

able to cover many different cases. If grammars are not be used, a valid payload is likely to be either be too specific and not cover a variety of cases, or contain too much completely random data to explore a program deeply. Invalid payloads are often used in testing too, and grammars can be used to generate such payloads as well. This can be achieved through small mutations that break the original, valid grammar slightly, but in a controlled manner.

3 Problem Definition

3.1 Adversary Model

To formally define the input points and components targeted by a fuzzer, an adversary model is needed. The goal is to define what issues found by the fuzzer are considered a vulnerability and what input channels can be used, as they need to be exploitable in this model. The adversary A is targeting the application with vulnerability payloads through different channels, as shown in figure 3.1.

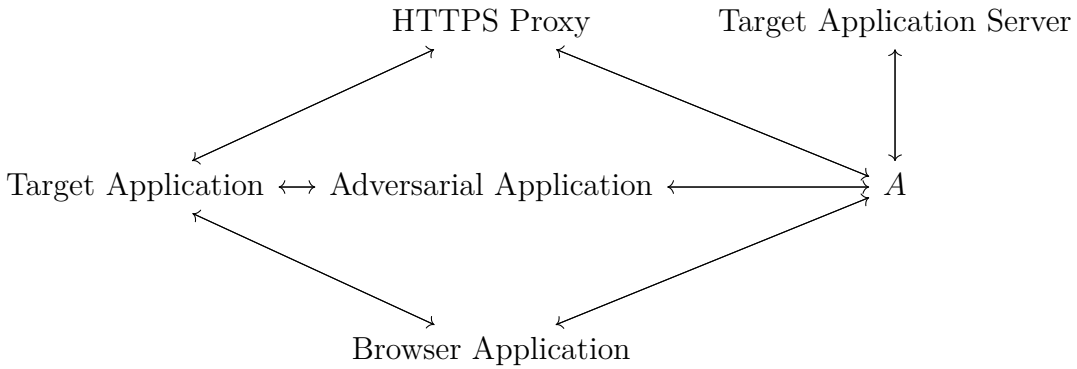

Figure 1: Mobile Application Adversary Model

Three different channels of different strength are used to attack the application. The strength of the adversary allows it to use any channel, but the channel used significantly affects the severity of the vulnerability. The adversary model applies to a mobile application on Android, which sandboxes applications but allows cross-application communication and opening applications with links. The model is also used for iOS, but the adversarial application and browser application are considered equally impactful, because cross-application communication is mostly conducted through a link, at least for the purposes of this research.

The adversary A is assumed to have a man-in-the-middle position on HTTPS traffic, which allows it to inject payloads in HTTPS responses. Similiar risks are present through vulnerabilities on the server itself, and this configuration simulates such cases as well. This is the strongest assumption made for the adversary, so vulnerabilities using this channel have reduced impact. A also controls another application on the same device as the target. Mobile operating systems are extremely secure and applications are well sandboxed, so there are no obvious risks involved, but protecting against a malicious application on the same device should be taken into account in cross-app communication. Vulnerabilities using this channel are considered fairly impactful. Finally, it is assumed that A can manipulate a web page opened inside a browser. Although this can be simulated by the malicious application, it is assumed that the target application or operating system can distinguish a trusted browser application and may only allow it to open links. If a vulnerability involves opening a link and the link parameters being sent to the target application to be

handled, a user may open the link from a web page and no malicious application is required at all. Vulnerabilities that can be exploited through links have high impact. These three input channels used by the components in the model are used to construct delivery methods and input languages for the payloads.

3.2 Research Goals

The expected outcome of the research includes a method, which produces and selects payloads to use for fuzzing a mobile application by using formal languages and grammars. The variables that affect these are the input formats of the target app and the vulnerabilities that want to be searched for. How specific these are, depends on the available static analysis and the user. Examples and evaluation of it's use in practice will be included for common vulnerability types. In the system model, this corresponds to the payload generator and static analyzer. The other components will be implemented but not studied in depth.

The scope of this research includes the two main data input points of applications: cross-app communication and HTTPS responses, which are used to search for conditions of common security issues caused by application logic. Common vulnerabilities of mobile applications are searched for with the payloads. These include HTML injections, JavaScript execution, open redirection, SQL injections, Java deserialization and data or permission leaks through intents. The results should be extendable to any input format and new vulnerability types.

The ideal outcome has similiar efficiency as current web application HTTP fuzzers, but in mobile application context. Web application testing has been easier to develop, as it almost exclusively uses HTTP, and because web vulnerabilities are often a more serious concern. Tools such as BurpSuite and sqlmap search for specific vulnerabilities by modifying HTTP requests in certain ways. The challenge with mobile applications is that data is not inserted in a HTTP request, as the application is the one making the requests, so different input formats must be considered. Mobile application security can be compared to browser security, but browsers are extremely strictly developed and only by a handful of vendors, while mobile applications can be developed and published by almost anyone. Applications also include more functionality, because one of their main purposes is to be an alternative to a browser-based application.

Vulnerability detection is also more difficult, as HTTP always has a clear, predictable response but mobile applications do not. However, mobile applications still have many serious security vulnerabilities, as recently demonstrated by mobile security researchers and companies such as Oversecured [22]. The importance of efficient tools is not any less important than in web security. Currently, mobile application security research is very limited to manual work and code analysis, so testing payloads and building a proof-of-concept attack is done mostly manually. The outcome of this research should reduce manual work in the case of payload generation for suspected vulnerabilities.

In practice, the outcome is a program which requires as input the formats that the app receives, and parameters related to specific vulnerabilities. Default formats are included. The program outputs efficient payloads which are compatible with

these formats. The outcome does not include full, efficient delivery methods of these payloads to the application and detection of vulnerabilities, as these require a large amount of resources to develop properly. A working concept of the system was still constructed, as example use cases will be included and are required for testing of the payloads. For these reasons, the method must also be compatible with different kinds of fuzzing algorithms, and all system components must be defined so that they can be generalized in further work.

In the context of fuzzing, efficiency is difficult to define. At the application level, search-based fuzzing is used to find specific security vulnerabilities, so grammars should generate payloads that focus on the most promising input fields and are able to deliver data past filters and validity checks. To generate inputs efficiently requires the method to have four important properties:

R1 Coverage: The method must be able to consider every possible field of the input points as a vulnerability payload insertion point. Otherwise a vulnerability may be not be found because it wasn't even attempted.

R2 Prioritization: The method must be able to prioritize input fields that are promising according to some heuristic. This can be implemented as a probability distribution over the input fields. Otherwise, many payloads unlikely to be successful will be generated, slowing down testing.

R3 Depth: The method must be able to bypass input validity checks in the same way as an ordinary input can. This is implemented by including payloads which are very similiar to ordinary inputs. Too many mutations and insertions of data may get rejected by the application.

R4 Compatibility: The method must be able to change its configuration and grammars dynamically. An external fuzzing algorithm or other program can evaluate which inputs have promising responses and change the generation of new payloads accordingly. Otherwise, adapting to promising responses cannot be done.

In general, responsibility for using the fuzzer in a way that satisfies these properties well is left to the user configuring the testing. Many other configuration decisions affect the efficiency as well, with the most important aspects being the fuzzing algorithm itself and the detection of responses to the fuzzing payloads. The algorithm controls what payloads are being tested and whether to change the payloads based on the responses. Compatibility with many different options is important as the fuzzing algorithms can vary depending on the input format and how responses are obtained. They can also be optimized for different scenarios or include external tools such as machine learning algorithms. The tools that are used to deliver the fuzzing payloads are not included in the grammar, but the grammar must be constructed in a way that allows adjustments made by the tools. The structure of the grammar must allow the testing framework to implement all the desired properties.

The grammars implemented in the research satisfy the properties by using probability distributions to control what the payload generation should focus on. Different

heuristics can be used to determine what sort of payloads work best, and they depend on the type of vulnerability that is being searched for and the response types. The amount of information available on the target app is also a factor. As the focus of the research is in grammars, these heuristics are not examined in depth, but it is shown how the implementation can be combined with further fuzzing techniques.

4 Efficient Input Generation

Inputs to a mobile application can be modelled as a language generated by a grammar. The sentences of this language are all valid inputs to the mobile application through some input channel. Smaller payloads that are known to trigger specific vulnerabilities are inserted into the grammar as terminals, and the purpose of the full fuzzing payload generated by a grammar is to deliver the smaller payload to a critical part of the application code. This section describes how these inputs were identified and formalized during the research.

4.1 Java classes as Fuzzing Inputs

To be able to construct the correct types of grammars, it was defined how fuzzing payloads can be generated as Java objects, which are an important part of data delivery in Android applications. Parameterization in the context of fuzzing Serializable and Parcelable Java classes involves defining a set of parameters or factors that influence the generation of test inputs. These parameters help in creating diverse and comprehensive test cases to thoroughly evaluate the behavior of the application after the serialization and deserialization processes. The serialization process itself should succeed perfectly on every instance, because our goal is to test the application logic after the Java object has been successfully delivered. An Android application is used to insert the generated fuzzing payload into a modified copy of the target Java class through a custom function and then pack an instance of it into an Intent to be sent to the target. Any other data packed into an Intent is a primitive type as explained in section 2.2.2, and can thus also be included in the parameters. Android Studio can be used to create and deploy a proxy application which implements this. The goal is to simplify the payload itself as much as possible while still having full coverage. It is inefficient to create a new instance of a Java class for each payload if we can simply change the parameters of an existing object.

The parameters must be assigned default values, which are always valid and are partially replaced in fuzzing payloads. Choosing the best possible default value is a complex problem. They can be chosen manually, by recording the values used when the application is running or by a heuristic function.

Parcelables will write only primitive data types and collections of them to a Parcel. This can be used to a considerable advantage in fuzzing, because every Parcelable class can be fuzzed by only generating primitive types. The grammars in this method include four types: string, integer, float and boolean. Bytes and long values can be generated from integers and collections can be dynamically created

if required. Other Parcelables can also be written, but they must eventually only include primitive values as well. By definition, a Parcelable can be modeled simply as an ordered list of values with primitive data types. The model of a Parcelable Android class for fuzzing includes the following components:

Parameter Class Identifies the equivalent primitive types of fields within the Parcelable class. The generated test inputs are first converted into an instance of a Java class which includes a field for each parameter. A JSON string can be trivially converted into such a class, if all the field types match. The field values of the Parcelable can be replaced by this object, as it includes all the necessary data.

Field Values A list of values of primitive types for each field within the Parcelable class. Primitive values can be obtained by observing how the field values of the Parcelable class are written into a Parcel, and such values must always exist by the definition of Parcelable classes. These values are defaults which must be always valid when the Parcelable is delivered.

Parameter Loading Method A method of the Parcelable class which initializes the Parameter class from a JSON string.

Write To Parcel Method The method which each Parcelable class must include. It reads the values from the Parameter class and writes them into a Parcel in the correct order. It is also responsible for correctly constructing and writing exceptions to the primitive values, such as lists and nested Parcelables. Nested Parcelables must have their parameters assigned beforehand, otherwise an error is thrown.

The Serializable class is slightly more complicated in fuzzing context. Each serialized field must be a primitive data type or also Serializable type. However, they must have the exact same field names as the class in the target application, so the Parameter object must be used to assign all of the parameters to the fields of the class. This process is still very similiar to how values are written into Parcels. The added complexity comes from the fact that the serialization process can be customized to write data directly into a byte stream, but it is similiar to writing to a Parcel. The model of a Seriaizable Android class for fuzzing includes the following components:

Parameter Class Identifies the equivalent primitive types of fields within the Serializable class. The generated test inputs are first converted into an instance of a Java class which includes a field for each parameter. A JSON string can be trivially converted into such a class, if all the field types match. The field values of the Serializable are set by reading them from this object, as it includes all the necessary data.

Field Values A list of values of primitive types for each field within the Serializable class. Primitive values can be obtained by observing the constructor of the class. These values are defaults which must be always valid when the Serializable is

delivered. The `transient` keyword can be used to define fields which are not serialized. The Parameter class can be set as transient if it is needed by the Write Object Function.

Serial Version UID The Serial Version UID is a static long variable in every Serializable class. It guarantees that the deserialized object is an instance of the exact same class. It must be set into exactly the same value as in the target application code, and if it not set, it is generated by the JVM based on many different factors. Not setting the value is likely to break functionality and can also make fuzzing impossible, and in most cases it is defined as some unique value.

Parameter Loading Method A method of the Serializable class which initializes the Parameter class from a JSON string and then assigns each parameter to it's corresponding class field.

Write Object Method An optional method, which defines how the class should be written directly into a bytestream. This function may read parameters from the transient Parameter class and write them to a byte stream.

Any data packed into an Intent must be either a primitive type value, a Parcelable object or a Serializable object, as explained in section 2.2.2. Thus, a set of parameters of primitive types covers all inputs of an Intent. These definitions allow for fuzzing payloads to simply generate JSON strings which contain the required parameters. Coverage is not lost, because any fields that can not be parametrized as primitive values, can not be delivered to the target application at all.

4.2 Mobile Application Input Languages as Grammars

In this section, grammars for selected mobile application inputs and programs used to generate them for a specific target application are presented. The languages generated by these grammars are *input languages*, which are valid inputs that a mobile application will process. Input points are defined as code execution starting points that are directly inaccessible by any other input point. The following generic inputs to mobile applications are included:

- HTTP responses
- URLs

HTTP responses bring data to a code execution point where an HTTP request was made, and deeplink data is sent to the application as URLs. Files are also an input to mobile applications, but they are not included for several reasons. Loading arbitrary files with an application is a vulnerability in itself and it is related to URI validation. File type validation can be achieved by testing URI validation with the URI containing a type identifier. Simple test files not requiring their own grammars can be used. Additionally, to cover all file types with grammars is a very large and

inpractical task. If needed, they can be developed with the same principles presented in this research.
The following Android-specific inputs are included:

- Generic Intent data
- Serializable object data
- Parcelable object data

Intents are sent to launch Android components and their data is used when the component code is executed. Objects of Serializable and Parcelable classes create an instance of the class and set it's fields. It is unknown in advance whether any other input point can be used create these objects with arbitrary data, so they are considered their own input points.

It was shown in section 4.1 that all Intent data can be generated as JSON and then processed in Java to construct the Intent with the correct payload. This is important, because a JSON grammar can be used for all three cases. Only few properties related to Java in the grammars must be taken into account. This is much more efficient than having separate grammars for different Java implementations, because grammars only need to generate parameters instead of full Java objects. The Java classes used in different applications also vary, but grammars do not need to be adjusted much.

The formal form used in this section for defining the grammars is referred to as the Backus-Naur form (BNF) [37]. Grammars are implemented in the Python programming language and the *fuzzingbook* [51] library is used to modify the grammar properties and generate payloads. In an implementation, a grammar can be represented as JSON or a dictionary. In BNF, each entry includes a symbol as a key and an expression separated by "::=". Symbols are written using the special characters "<" and ">" around the name of the symbol: <*symbol*>. The expression contains all the possible choices of sequences corresponding to the symbol in a production, separated by "|". Escapes are needed to define the special symbols as part of a terminal string, so they are not interpreted as part of the BNF. Escapes concerning the special symbols in this notation are defined as follows:

- <*left-arrow*> corresponds to the symbol "<"
- <*right-arrow*> corresponds to the symbol ">"
- <*pipe*> corresponds to the symbol "|"

Grammars are also combined with each other, which allows the modification of grammars without breaking functionality in another.

Shortcuts can be used in grammar definitions to simplify them. Three operators are used:

- (terminal<*symbol*>)? is used to indicate that the sequence inside the parenthesis is optional but singular, and can be included 0 or 1 times.

- (terminal<*symbol*>)+ is used to indicate that the sequence inside the parenthesis can be included 1 or more times.
- (terminal<*symbol*>)* is used to indicate that the sequence inside the parenthesis is optional but not limited, and can be included 0 or more times.

Grammars using these operators are referred to be in Extended Backus-Naur form (EBNF), and are converted programmatically into BNF. Since these three operators and the parenthesis around the sequence are required in the grammar in terminal strings, they are also escaped by defining special terminal symbols which produce only the wanted character.

First, an incomplete grammar for basic, primitive type values is defined. This grammar can be joined with other grammars, so these can be included without repetition. The definition is as follows:

BASIC_TYPES :=
<*string*> ::= "<*characters*>" | "<*payload*>" | "<*url*>"
<*characters*> ::= <*character*>*
<*character*> ::= a | b | c | ...
<*int*> ::= <*digit*> | <*onenine*><*digits*> | -<*digit*> | -<*onenine*><*digits*>
<*digits*> ::= <*digit*>+
<*digit*> ::= 0 | <*onenine*>
<*onenine*> ::= 1 | 2 ... 9
<*float*> ::= <*digits*>.<*digits*>
<*boolean*> ::= true | false
<*ws*> ::= ( )

The symbols <*payload*> and <*url*> are defined in separate grammars later, and are also joined with other grammars. Next, the base grammar for JSON input is defined as follows:

JSON_INPUT_EBNF_GRAMMAR :=
<*start*> ::= <*json*>
<*json*> ::= <*element*> | <*elements*>
<*element*> ::= <*value*>
<*elements*> ::= <*element*>
<*value*> ::= <*object*> | <*array*> | <*string*> | <*int*> | <*boolean*> |
<*float*> | null
<*object*> ::= {<*ws*>} | {<*members*>}
<*members*> ::= <*member*>(,<*members*>)*
<*member*> ::= <*ws*><*string*><*ws*>:<*element*>
<*array*> ::= [<*ws*>] | [<*elements*>]

This is not yet a useful grammar, because the symbol <*elements*> only contains one element. The elements are constructed dynamically and the grammar is modified based on the structure of an input HTTP response. An application will, in most cases, expect certain structure, names and types to correctly parse the response.

To avoid obvious parsing errors, the grammar must be able to generate complete payloads that maintain this structure.

JSONToGrammar is defined as a function which takes a grammar and a string in JSON form as input and extends the grammar with symbols for the JSON elements. The assumption is made that only strings, integers, real numbers and booleans are included as terminal values in the JSON input. The JSON can include any amount of objects and lists. When a string representing a JSON object is given, the resulting grammar will generate strings that are similiarly structured and have the same key names.

The algorithm goes through the JSON structure recursively and adds a new symbol into the grammar for each JSON element. The symbol of the element is it's name, and if the same name is defined multiple times in the JSON, it is considered equivalent. This may cause problems in specific cases, but can be solved by implementing a more complex grammar. In objects and lists, elements are optional, whichguarantees that generated payloads have as much variety as possible. Missing elements in specific cases can cause unpredictable behaviour, and is important for large coverage in fuzzing. For elements inside lists, a name is generated by the program. Each element can also be replaced by a payload or an empty instance of the element, or in the case of terminal values, the default value or generated value of the proper type. For example, when given the JSON string

{"key_1" : "value_1", "key_2" : ["value_2", "value_3"]},

the resulting EBNF grammar becomes:

<*start*> ::= <*json*>
<*json*> ::= <*element*> | <*elements*>
<*element*> ::= <*value*>
<*elements*> ::= <*element*>
<*value*> ::= <*object*> | <*array*> | <*string*> | <*int*> | <*boolean*> |
<*float*> | null
<*object*> ::= {<*ws*>} | {<*members*>}
<*members*> ::= <*member*>(,<*members*>)*
<*member*> ::= <*ws*><*string*><*ws*>:<*element*>
<*array*> ::= [<*ws*>] | [<*elements*>]
<*elements*> ::= {"key_1":<*json-key_1*>(, "key_2":<*json-key_2*>)?} |
<*string*> | { }
<*json-key_1*> ::= <*string*> | "value_1"
<*json-key_2*> ::= <*string*> | [<*arrayObject-1*>(, <*arrayObject-2*>)?] | [ ]
<*arrayObject-1*> ::= <*string*> | "value_2"
<*arrayObject-2*> ::= <*string*> | "value_3"

The algorithm performing the conversion is shown in pseudocode in listing 1.

JSON is also used to define parameters for Android Intents. Intents and other Java classes included with them do not require their own grammars because they are

```
JSONToGrammar(grammar, jsonString):

   parsedJson = convert jsonString to a map
   extendGrammar = a function which adds a definition into a
      grammar

   convertRecursively(grammar, object, name):
      i = 1
      if object is a dictionary:
         languageString = "{"
         first = true
         for key, value in items of object:
            jsonKey = "json-" + key
            addString = '"' + key + '"' + ':' + '<' + jsonKey
               + '>'
            if first is false:
               addString = "(, " + addString + ")?"
            else:
               first = false
            languageString += addString
            grammar = convertRecursively(grammar, value,
               jsonKey)
         languageString += "}"
         return extendGrammar(grammar, { '<' + name + '>' ::=
            languageString | <payload> | {} })

      else if object is a list:
         languageString = "["
         first = true
         for listObject in object:
             objectName = "arrayObject-" + str(i)
             addString = '<' + objectName + '>'
             i += 1
             if first is false:
                addString = "(, " + addString + ")?"
             else:
                first = false
             languageString += addString
             grammar = convertRecursively(grammar, listObject,
                 objectName)
         languageString += "]"
         return extendGrammar(grammar, { '<' + name + '>' ::=
            languageString | <payload> | [] })

      else if object is a string:
         return extendGrammar(grammar, { '<' + name + '>' ::=
            <string> | '"' + object + '"'})
      else if object is a boolean:
         return extendGrammar(grammar, { '<' + name + '>' ::=
            <boolean> | object })
      else if object is an integer:
         return extendGrammar(grammar, { '<' + name + '>' ::=
            <int> | object })
      else if object is a floating point number:
         return extendGrammar(grammar, { '<' + name + '>' ::=
            <float> | object })
   return convertRecursively(grammar, parsedJson, "elements")
```

Listing 1: Pseudocode for converting JSON to grammar symbols

a fundamental part of the Android operating system instead of the target application code. They are a delivery method, and ensure that the generated payloads reach their intended destination, so testing them directly is not our goal. To generate and pack Parcelable and Serializable Java classes to an Intent, first the parameters are generated by the grammar and then initialized into the desired Java class. The JSON parameter grammar is defined as follows:

JSON_PARAMETER_EBNF_GRAMMAR :=
<*start*> ::= <*element*>
<*element*> ::= {<*parameters*>}
<*parameters*> ::= "parameter-1":<*string*>

Clearly, this grammar is a simplified version of the earlier JSON HTTP grammar and simply a foundation for extensions. It includes an example expansion for the <*parameters*> symbol, but it is updated dynamically based on the Java code of the target application for each Parcelable and Serializable class in it. *JavaToGrammar* is defined as a function which takes a list of parameters as values and a grammar, extends the grammar with ordered parameters which can be used to generate payloads equivalent to a Parcelable or Serializable class.

The code adds a new symbol for each parameter and preserves it's type. In this case, the parameters cannot be optional as the Java classes require the exact correct types and amounts to be initialized without errors. This grammar and the method which creates it can be used to model Parcelable and Serializable classes in mobile applications. An Android application deployed with Android Studio can be used to deliver these inputs to the target application, as described in the previous section. For example, when given a list containing the default parameters for the fields of a Java object, such as

$$[\text{"string\_1"}, \text{"string\_2"}, 123, 99.99, true],$$

the resulting EBNF grammar becomes:

<*start*> ::= <*element*>
<*element*> ::= {<*parameters*>}
<*parameters*> ::= <*parameter-1*>,<*ws*><*parameter-2*>,
<*ws*><*parameter-3*>,<*ws*><*parameter-4*>,<*ws*><*parameter-5*>
<*parameter-1*> ::= "parameter-1":<*string*> | "parameter-1":"string_1"
<*parameter-2*> ::= "parameter-2":<*string*> | "parameter-2":"string_2"
<*parameter-3*> ::= "parameter-3":<*int*> | "parameter-3":123
<*parameter-4*> ::= "parameter-4":<*float*> | "parameter-4":99.99
<*parameter-5*> ::= "parameter-5":<*boolean*> | "parameter-5":true

The algorithm performing this conversion is shown in pseudocode in listing 2.

URLs are used to open applications from websites in a mobile browser or from another application. They are also generally used in data transfer to imply locations, and can include parameters. Fuzzing them properly is extremely important, as links

```
JavaToGrammar(grammar, values):
   extendGrammar = a function which adds a definition into a
      grammar
   paramString = empty string
   i = 1
   for value in values:
      paramName = 'parameter-' + i
      paramJSON = '"' + paramName + '"'
      if i > 1:
         paramString += ',<ws>' + '<' + paramName + '>'
      else:
         paramString = '<' + paramName + '>'
      if value is a string:
         grammar = extendGrammar(grammar, { '<' + paramName +
            '>' ::= paramJSON + ":<string>" | paramJSON + ':"'
             + value + '"'})
      else if value is an integer:
         grammar = extendGrammar(grammar, { '<' + paramName +
            '>' ::= paramJSON + ":<int>" | paramJSON + ':' +
            value})
      else if value is a floating point number:
         grammar = extendGrammar(grammar, { '<' + paramName +
            '>' ::= paramJSON + ":<float>" | paramJSON + ':' +
             value})
      else if value is a boolean:
         grammar = extendGrammar(grammar, { '<' + paramName +
            '>' ::= paramJSON + ":<boolean>" | paramJSON + ':'
             + value})
      i += 1
   return extendGrammar(grammar, <parameters> ::= paramString)
```

Listing 2: Pseudocode for converting Java parameters to grammar symbols

are a very easy method of getting victims to trigger an exploit on their device. The URL grammar includes multiple schemes. The file schemes describe file locations and the JavaScript scheme can be used to execute code. Applications must be able to properly validate schemes and hosts in URLs to avoid severe vulnerabilities. The URL is an input language but can also be used as a payload embedded in other inputs. The URL grammar is defined as follows:

URL_EBNF_GRAMMAR :=
<*start*> ::= <*url*>
<*url*> ::= <*scheme*>://<*body*> | <*js-url*> | <*intent-url*> |
url:<*url*> | <*path*>
<*scheme*> ::= <*http-scheme*> | <*file-scheme*> | <*intent-scheme*> |
<*js-scheme*> | <*misc-scheme*> | <*app-scheme*>
<*http-scheme*> ::= http | https
<*file-scheme*> ::= file | content | data
<*intent-scheme*> ::= intent
<*js-scheme*> ::= javascript
<*misc-scheme*> ::= url | tel
<*app-scheme*> ::= <*appname*>
<*appname*> ::= com.test.app
<*app-component*> ::= MainActivity
<*body*> ::= <*authority*><*path*><*query*> | <*path*> | <*payload*>
<*js-url*> ::= <*js-scheme*>://<*js*> | <*js-scheme*>:<*js*>
<*intent-url*> ::= <*intent-scheme*>://<*body*>#Intent;package=<*appname*>;
scheme=<*app-scheme*>;component=<*app-component*>;S.<*extra*>;end
<*extra*> ::= <*url-string*>=<*url-string*>
<*js*> ::= alert<*left-par*>1<*right-par*>
<*authority*> ::= <*host*> | <*host*>:<*port*> | <*userinfo*>@<*host*> |
<*userinfo*>@<*host*>:<*port*>
<*host*> ::= () | 0.0.0.0 | localhost
<*port*> ::= 80 | 8080 | <*nat*>
<*nat*> ::= <*digit*> | <*digit*><*digit*>
<*userinfo*> ::= user:password
<*path*> ::= <*slash*>? | <*slash*>?<*url-string*><*slash*>? | <*internal-path*>
<*internal-path*> ::= <*slash*>?internal<*slash*>path<*slash*>file
<*query*> ::= () | <*question-mark*><*params*>
<*params*> ::= <*param*> | <*param*>&<*params*>,<*payload*>
<*param*> ::= <*url-string*>=<*url-string*> | <*url-string*>
<*url-string*> ::= <*characters*> | <*url-payload*>
<*url-payload*> ::= <*enc-payload*> | <*payload*>
<*slash*> ::= /

Note that empty parenthesis correspond to a completely empty string. JSON and URLs can be defined as either input languages or vulnerability payloads, as they are used in both contexts: a URL can be a payload inside a JSON input and it can also be an input in the form of a deeplink. JSON data can be included a string nested in

other JSON or a URL. It is common for URLs to include other URLs inside them, for example as a redirection target. This can be accomplished by copying and renaming the grammar to avoid confusion and the *<start>* symbol must not be included in the grammar. It can then be added without breaking another grammar. Recursively added URLs and JSON do require an additional layer of encoding, which has to be taken into account in the grammar. Doing this will also increase the complexity of fuzzing, so in this implementation we simply include some test JSON strings and URLs alongside other payload strings. Another possibility is to first generate such strings with the grammars and then add them as terminal symbols in other grammars, similiarly to the other payload strings.

In general, any grammar without the start symbol is meant to be added to another grammar, but can also be used on it's own if that symbol is added and it is otherwise valid. If the start symbol was included in every grammar, adding a grammar to another overwrites the start symbol or create other unnecessary confusion.

4.3 Vulnerability Detection Payloads

Payload grammars are important to keep separate from the input grammars, because they serve a completely different purpose. The input grammars preserve valid formats, while the payloads are meant to break the formats and find and trigger specific security vulnerabilities. Payloads can also be altered and changed according to different search targets, but such changes should not affect the input formats in any way.

The payload types defined here are able to trigger the vulnerabilities described in 2.3.1. The examples given only work in specific cases, but adding more payload strings is simple and helps covering as many cases as possible. The memory corruption payload given here can initialize a Java class included in the Android Java library, which includes a pointer, if it is dynamically created from a JSON string. However, a class name as a payload is also enough to detect dynamic class loading, which is a vulnerability in itself. The ”misc” payloads include different data formats, which can be useful in detecting filters and accessing deeper functionality. Polyglots can be used to detect unsafe parsing of data. The collaborator URL is a special URL using a host in the user's control, which can be used to detect whether a call was made to it by the application. Different forms of it can be generated with the URL grammar.

Payloads are included as simple grammars. Different types of vulnerability payloads have a different symbol, so a probability distribution can be used to focus on specific vulnerabilities. Six vulnerability types are defined here, but more can be added to the grammar. The payloads themselves are strings, which are loaded into an array and they must be terminal symbols. The payload grammar is defined as follows:

PAYLOAD :=
<*payload*> ::= <*sql*> | <*xss*> | <*path-traversal*> | <*command-injection*> |
<*memory-corruption*> | <*misc*>
<*sql*> ::= ' | ' OR 1 -- -
<*xss*> ::= "'<*left-arrow*>script<*right-arrow*>alert(1)<*left-arrow*>/script<*right-arrow*>
<*path-traversal*> ::= ../
<*command-injection*> ::= ; sleep 10
<*memory-corruption*> ::= com.android.internal.util.VirtualRefBasePtr |
{'mNativePtr':3735928551}
<*misc*> ::= "'><h1>poly${{5*5}}glot |
{"test_1":"test_2", "test_3":"test_4"} | https://collaborator-url

More payload strings must be added to the grammar before fuzzing, and they can be loaded from a file. An efficient and thorough XSS or SQL injection payload list includes more than 100 strings. The URL grammar also included the <*enc-payload*> symbol. This grammar is a copy of the payload grammar, but each symbol name has the prefix *enc-*. Every payload in that grammar is URL encoded, instead of being a plaintext string.

4.4 Probability Distribution and Efficiency

To make fuzzing efficient, we must prioritize the creation of certain payloads based on increased potential of triggering a vulnerability. This potential can be dynamically recognized from how the application responds to payloads, or by simply planning this based on information known about the application before fuzzing. Some prioritization can also be used on default, such as using the HTTP/HTTPS protocols more often than others, as it is the most used protocol in URLs and most functionality can be assumed to accept it with a high probability.

Grammars themselves do not have a probability distribution defined. In the implementation, by default no expansion for a symbol is prioritized and they are all equally likely. However, there can be a very massive difference in the effectiveness between two expansions. It is possible that if a symbol has two possible expansions, the target program will reject nearly every single string including the first expansion but none of the strings including the second expansion. This results in up to half of the payloads being completely useless, reducing efficiency. When the string is generated using the grammar, it should not select the expansion with 50% probability, but the accepted expansion with a much higher probability. Some randomness should still exist in the choice, because it is also possible that some important edge case exists for the less potential option, so the expansion choice should rarely be set to absolute certainty.

In the context of generating fuzzing payloads, a probability distribution refers to a method of determining the likelihood or frequency with which certain elements or patterns appear in the generated payloads. This distribution can be used to guide the fuzzing process towards generating payloads that are more likely to trigger unexpected behavior or vulnerabilities in a target system. In grammars, this is

implemented as a separate distribution for each production of expansions from a symbol. Let P be the set of productions in a context-free grammar. Define the following set:

$$P_\alpha = \{\beta \text{ so that } (\alpha \longrightarrow \beta) \in P\}$$

Define $\Pr(\beta)$ as the probability that β is selected as the expansion of α when a string is generated using the grammar. The probability distribution has the following property for every P_α in the grammar:

$$\sum_{\beta \in P_\alpha} \Pr(\beta) = 1$$

By default, in a grammar each expansion is equally likely. In this case, no probabilities have been set when generating a string and the probabilities are allocated during the generation process:

$$\Pr(\beta) = \frac{1}{|P_\alpha|}, \text{ where } \beta \in P_\alpha$$

If the probability is set as a fixed value for a specific expansion β, the other expansions will have probabilities distributed equally during generation so that the sum of all probabilities will equal 1. If multiple expansions have a fixed probability value, those values are used on generation and the remaining expansions will again be assigned a probability according to the sum. This also means that the fixed probabilities cannot total more than 1, and if they are equal to 1, the expansions with no assigned probability will never be selected during generation.

Setting these probabilities higher or lower than the default value will enable the prioritization of potential expansions during generation of fuzzing payloads. Equal distributions will remain between expansions that are not specified, so prioritization can be done without having to consider every expansion separately.

Parsers can be thought of as the reverse operation of generating a string from a grammar. They try to recognize which expansions of a grammar were used to generate a given string. This can be used to assign probabilities which will generate payloads similiar to a given sample set by comparing the amount of specific expansions in the samples and all possible expansions. If a set of potential previously generated payloads is known, future payloads can include very similiar payloads. This will, of course, drastically reduce the generation of payloads different from the samples, which reduces the coverage. Machine learning techniques can be used to balance the probabilities this way, but the testing in this research does not include such techniques and probabilities will only be set manually. One of the requirements defined for an efficient fuzzing system was compatibility, and the probability distributions along with parsers satisfy this requirement by allowing any algorithm to decide what kind of payloads to prioritize based on samples or other heuristics. Payload generation can be reduced to other statistical problems by modeling it with probability distributions.

5 Method Implementation

A Linux system was used for development of the implementation. The payload generation functionality was implemented using Python and the payload delivery Android application was developed using Android Studio 2022.2.1.

5.1 Fuzzing System

The model of a fuzzing system used in this research contains several different components, which communicate with each other and transfer objects. In the graph 5.1, the arrows indicate the direction of data transfer and commands sent between components.

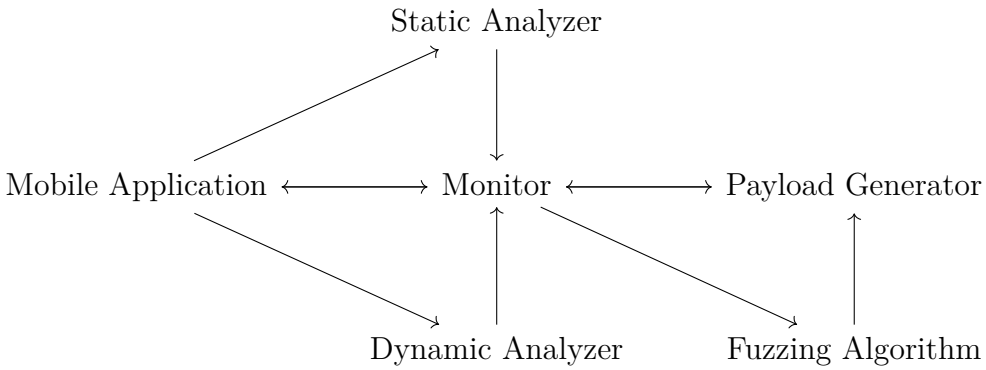

Figure 2: Fuzzing System Model

Mobile Application This is the target application of the fuzzer. It includes either an .apk or .ipa file and is installed on a mobile operating system. The objects related to the application are its source code, byte code or binary code, used by the static analyzer, the application state, monitored by the dynamic analyzer, and the responses delivered directly to the monitor. Responses may include Intent responses in the case of Android or log entries written by the application. The monitor controls the launching and runtime of the application and is able to read operating system logs.

Monitor The monitor is the core component of the system and is responsible for most of the interaction with the mobile application. It includes a tool for delivering payloads to the application and reading the immediate responses given by the application, and can intercept HTTPS requests to insert payloads in responses. In Android, it includes a separate Android application from the target application, which inserts the payloads into Java objects delivered to the target. The mobile operating system is controlled by the monitor to execute fuzzing related tasks such as launching iOS deeplinks, reading system logs or simulating user actions on the target application. The dynamic analyzer relies on the monitor to control dynamic instrumentation tools in the application or the operating system.

Dynamic Analyzer The purpose of dynamic analysis in a fuzzing system is to inspect the state of the application during the fuzzing process. This usually includes inspecting what classes the application has loaded, what methods are run, and what changes happen in the current activity or UI element. A tool such as Frida [42] can be used to implement this and it requires either root access to the operating system or a gadget patched into the application. The dynamic analyzer is separate from the monitor, because it does not control any fuzzing related processes or environments, it simply reads and processes information which is passed to the monitor.

Fuzzing Algorithm The algorithm is the part of the fuzzing system which decides how to change fuzzing inputs based on received outputs. The monitor continously gives it data, such as the latest fuzzing input and the result of that input. The algorithm then sends feedback to the payload generator. The performance of methods for doing that depend on a variety of factors, and fuzzing algorithms are a widely studied topic which is explained briefly in section 2.1. The monitor and the payload generator should be compatible with essentially any fuzzing algorithm, as different algorithms can produce vastly different results. However, since the fuzzer is targeting application level security vulnerabilities, depth is generally preferred to coverage, and this can be taken into account in the design of the system. Also, with the increased popularity of large language models and machine learning, better compatibility can help testing and integration of complex analysis techniques.

Payload Generator This component is the main focus of the research and is responsible for creating random payloads which are structured in the same way that inputs to the mobile application are, but also include specific vulnerability payloads. The requirements for efficient payload generation are defined in section 3.2. The payload generator is configured with known vulnerability payloads that are used to trigger malicious behaviour in the application. It receives grammars of input languages from the static analyzer, which help it structure the payloads targeted for a specific application. During the fuzzing process, it receives data from the fuzzing algorithm which can be used to alter the generation parameters based on previously successful or unsuccessful payloads. Generated payloads are sent to the monitor.

Static Analyzer Static analysis is used at the beginning of the fuzzing process to create grammars of input languages of the application based on it's code. Depending on how much code is available, the static analyzer adapts and generates the best possible set of languages. The supplied data can range from simple configuration files to Java bytecode to full source code. This component is the second main focus point of the research.

5.2 Enumerating Application Inputs

Enumeration forms the foundation of the fuzzing process. The Android application is decompiled into readable Java code using *jadx* [39] and all possible input points are located. The *AndroidManifest.xml* file includes information about exported components, which can be launched by another application using Intents. Serializable and Parcelable classes are searched for in the code but classes included in libraries are not considered. The points of interest are the classes unique to the application. Each exported component and each Serializable and Parcelable class form one input point in an Android application, because the input to each exported component is an Intent, which can be parameterized as explained in section 4.1. A URL is considered an input point if the application defines an intent filter for one of the components, which has the category *BROWSABLE*. It also defines the schemes and possibly the hosts and paths of the expected URLs. They are added to the URL grammars as terminals. This input point can be opened as links from a browser, so it has higher relevance and impact. URIs and objects of a Serializable or Parcelable class can be placed into any Intents already defined. In this case, the URI is a payload instead of an input point, because the URI is simply a string carried by the Intent. The Java objects are considered additional input points, because the code of the classes may be inaccessible without them, as explained in section 4.2.

The Java code is also analyzed to detect Intent data use. If the code of the component they are sent to in an Intent calls the *getIntent()* method, the Intent and all it's content become a Java object and they are automatically deserialized, so the Java object input points can be accessed. Components calling the *getData()* method will use a URI, because the method returns a URI associated with the Intent. This information is not necessary, but helps in prioritizing fuzzing targets.

The iOS application contains the *Info.plist* file, which defines deeplinks that are accepted by the application. The schemes and possible hosts and paths included in it are added to the URL grammar as terminals, so that they are included in payloads.

In both iOS and Android, HTTP responses are collected by a proxy, which records each request made by the application and the corresponding response. Proxies are a common tool and bypassing encryption for HTTPS requests can be accomplished by inserting an SSL certificate and possibly disabling the application's SSL pinning [21]. The details of how it can be done are not included, as it depends highly on the OS, it's version and the application.

To improve the quality of the generated payloads and to avoid the need for randomly guessing parameters, the method includes fetching strings from the application code. They are then set as terminals in the grammars, because comparisons to these strings are likely to exist in the code. This can be done directly on the application files with the *strings* command on Linux, which pulls all readable strings from inside a binary file. On Android, Java code decompiled by *jadx* can be searched for strings. The strings can be identified as paths or hostnames and added to the payload generation.

5.3 Processing Java Classes

The relevant Java classes found in the application are replicated in the payload delivery application and changes are made to them. The process of modifying the Java classes is explained in section 4.1. The source code of the application produced by *jadx* is copied and included in the application with the exact same package and name, so that deserialization works properly. Android Studio can be used to run an application on the device when the source code is added to a default application.

Modifying the Serializable class includes defining a Parameters Java class, which contains the field values that are set to the object when fuzzing. It also contains data written directly to the bytestream. The default values for this data are ordered and defined as a list, which is then used to create a grammar for the class parameters. An object of it is defined as a *transient* field in the Serializable class, so it is not serialized along with the object. Implementing the *writeObject* method is done if the original class contains a *readObject* method. The method performs the actions of the read method in reverse, writing data from the Parameters class into the bytestream. To implement the parameter loading method, the *Gson* [40] class is used to convert a JSON parameter string into a Parameters object. If the serial version UID is not set in the original class, it must be obtained by observing a deserialization error from the Android logs after sending a payload with the default UID. Reading logs is described in section 5.4.

Modifying the Parcelable class also includes defining a Parameters Java class and the parameter loading method in the same way as the Serializable class. The parameters only include a value of the types written into the parcel in the *writeToParcel* method of the original class, in the same order that they are written in. Then, the *writeToParcel* method is modified to write to the Parcel each parameter from the Parameters class in the same order.

Default values for the parameters which the class parameter grammar is based on, can be obtained directly from the class definitions, guessed or dynamically observed by running the application, but obtaining them is not included in the scope of this research. The value *string* is used as default value for strings, *c* for char, 0 for bytes, shorts, ints, longs, floats, and *true* for boolean.

5.4 Delivery of Payloads

The payloads are generated by a Python program utilizing the *fuzzingbook* library [51] and put into output files. These are text files with one payload on each row and each payload type has it's own file. The Android device is connected to the machine with the payload files using the Android Debug Bridge (*adb*) over a local network. The virtual machine has root access to the Android system, allowing full manipulation of the file system, so the files were copied into the payload delivery application's *files* directory on the Android device, which is normally only accessible by the application itself. The application then loads the file into a *BufferedReader*, which only loads one row from the file at a time. This way the entire content of the file does not have to be loaded into the dynamic memory of the Java program, which is very limited in

applications.

The Android application is constructed to include functions for the different input points, which either create the necessary Java object and correctly load the JSON parameter payloads into it or in the case of the URL, create a Java *Uri* object and load the received URL payload into it. These objects are then packed into an Intent which targets a specific component. For maximum coverage, it is possible to include every input object in every component's Intent. For every possible input point, a function is created dynamically, as the name and type of the component is required to create the Intent and the Java class definitions are required from the application. Inputs can also be changed and specified later, if new, nested entry points are discovered during fuzzing.

On iOS, a jailbroken device or emulator is needed in order to gain root access and an SSH connection to it. Frida [42] was used to execute a JavaScript program on the device that can run in the context of the SpringBoard process [44]. SpringBoard is a process which controls the home screen of the device and the launching of other applications. By manipulating it, arbitrary deeplinks can be opened. The Frida program also includes a remote procedure call, receiving a message and opening it as a deeplink. A Python program can be used to read payloads and to send each payload one by one to Frida from the machine with the payload files. Other inputs besides deeplinks and HTTP responses were not considered in iOS, because they are far less common.

In the case of HTTP responses, an HTTP proxy is used to change the HTTP responses. This requires the application to trust certificates of the proxy to manipulate encrypted traffic, which may require manual work, but is a well documented process out of the scope of this research. Many HTTP proxy tools exist, BurpSuite most likely being the best one. Delivery of HTTP responses was not included in the tests. The requirements for an HTTP proxy configuration include being able to identify which responses were used to construct a grammar and a list of payloads, and insert the correct payloads into their corresponding responses. The URL from which the response was received from should be used as an identifier.

5.5 Vulnerability Detection

There are many different ways to detect vulnerabilities during fuzzing, but they all rely on detecting differences to normal behaviour of the application. During testing, the method included using six detection methods. They were the following:

1. Manual observation. This includes simply observing changes in the behaviour of the application through the UI when a vulnerability is triggered. For example, executing JavaScript may create an alert on the screen.

2. Classes and method calls. A tool such as Frida can be used to read what classes have been loaded by the application and which methods of a specific class are being called by the application. Changes in the behaviour of the application can be observed by changes in method calls.

3. Logs. The operating system logs crashes and different actions of applications. Crashes can indicate severe bugs and developers can choose to write to the log in application code, so information leakage can be possible. On Android, the *adb logcat* works very well and on iOS, Xcode and a MacBook is required.

4. Out-of-band requests. When injecting malicious URLs, the application might load a page or make an API call to the injected URL. By logging all requests to the URL, a vulnerability initiating such a request can be detected. This can be done either with an HTTP proxy or by controlling the malicious server.

5. Scanning legitimate HTTP requests. A proxy can be used to detect request content sent to legitimate APIs. If malicious content can be injected to legitimate requests, requests can be forged as user credentials may be included in them.

6. State changes. The state of the application can be monitored programmatically by using Frida other dynamic instrumentation. Changes to the UI, file system, running components or other similiar elements can indicate a vulnerability. Test files can be placed in the file system before fuzzing.

To correctly detect a vulnerability, the detection monitor ideally needs to be able to connect the previously sent payload to the observed changes. If it is not clear which payload triggered the change, replicating payloads may be required to narrow down the source of the change. This may be very difficult if payloads are sent in quick succession, as transmitting data between detection components has innate delays. Slowing down the fuzzing will of course make it slower and damage the efficiency. During testing, payloads were sent with a 1-3 seconds pause in between them. There are also other factors which affect the optimal pause time, such as the actual processing of the payload code, garbage collection, responses to Intent, threading and operating system limitations. In the worst case, some payloads are skipped entirely because the application is processing something else and ignores them. The best solution might include a probabilistic method connecting a set of payloads into a set of findings, but it is outside the scope of this research.

6 Evaluation

The Oversecured application was used as a case study to test the fuzzing method, and the enumeration required by the application-specific implementation was conducted on it. It contains multiple known vulnerabilities for testing purposes and different input points. Testing was not conducted on other applications, because the enumeration and detection processes are very time consuming to conduct separately on many applications. They can be automated, but the scope of the research is only to show their effectiveness, not to provide a complete, generally efficient implementation. The test included four different inputs for the Android application and one input in the iOS application. HTTP responses were tested using an example of a JSON HTTP response, but only payload generation properties were considered. Delivering HTTP

responses can be done with an HTTP proxy, for which countless implementations already exist. Vulnerability detection is not done fundamentally differently when fuzzing HTTP responses, so the dynamic payload generation is the focus of the HTTP response testing.

A Linux system was used to conduct the testing. The target Android application was run on an Android x86 virtual machine with Android version 9. The target iOS application was run on a jailbroken iPhone 8 with iOS version 14.0.1.

The results of testing show how the fuzzing method is successfully applied to selected application components and it can both trigger and detect vulnerabilities. It is then shown how the coverage requirement R1, the prioritization requirement R2 and the depth requirement R3 are satisfied for the considered inputs and their grammars. Finally, the system is evaluated as a whole and the compability requirement R4 is shown to be satisfied. The system is also compared to web security equivalent and other fuzzing systems.

6.1 Enumeration Results

The following input points were found by analyzing the *AndroidManifest.xml* file:

1. Deeplink. The following activity is exported: *.activities.DeeplinkActivity*. It has an Intent filter which defines the *oversecured* as scheme for links that open in this activity.

2. Exported Activities. The following activies are exported: *.activities.LoginActivity*, *.activities.EntranceActivity*, *.activities.MainActivity*.

3. Exported Service. The following service is exported: *.services.InsecureLoggerService*.

4. Content Provider. The following content provider is exported: *.providers.TheftOverwriteProvider*. It has the authority *oversecured.ovaa.theftoverwrite*.

5. Serializable Java classes. The following Serializable classes are defined in the *oversecured.ovaa.objects* package: *DeleteFilesSerializable*, *MemoryCorruptionSerializable*, *LoginData*.

6. Parcelable Java classes. The following Parcelable class is defined in the *oversecured.ovaa.objects* package: *MemoryCorruptionParcelable*.

Manual inspection of the decompiled application code showed that the exported *DeeplinkActivity* calls the *getIntent()* method. This loads the Intent and deserializes any Parcelable or Serializable objects inside. It also calls the *getData()* method, which always contains an Intent URL, being then also an entry point for URLs, but this was also indicated by the definition of a link scheme. This activity was selected as an entry point, as it tests the Intent and the URL input point. The detected scheme was set as a terminal in the URL grammar for the symbol *<scheme>*.

The other input points were selected to be *DeleteFilesSerializable*, *MemoryCorruptionSerializable*, *MemoryCorruptionParcelable*. The Java classes were replicated in the payload delivery application and changes were made to them.

```
{
  "data": [{
    "type": "example",
    "id": "1",
    "attributes": {
      "title": "Example title",
      "body": "Example body",
      "created": "2015-05-22T14:56:29.000Z",
      "updated": "2015-05-22T14:56:28.000Z"
    },
    "relationships": {
      "author": {
        "data": {"id": "42", "type": "people"}
      }
    }
  }],
  "included": [
    {
      "type": "people",
      "id": "42",
      "attributes": {
        "name": "John",
        "age": 80,
        "gender": "male"
      }
    }
  ]
}
```

Listing 3: JSON HTTP response used in testing

The enumeration of the iOS application revealed the following deeplink scheme: *ovia*. The deeplink is selected as an entry point in testing, so the scheme is set as a terminal in the URL grammar for the symbol *<scheme>*.

Strings were enumerated from both the Android and iOS application, and their effect is analyzed in section 6.3.

HTTP responses were not enumerated. as testing a HTTP proxy was not included in the scope. HTTP responses are usually not specific to the operating system, so a single example response is chosen for testing payload generation. The JSON example selected to simulate an HTTP response is shown in listing 3. A JSON grammar was generated from a file including the example response.

6.2 Java Processing Results

DeleteFilesSerializable contains a *readObject* method, which reads a Unicode Transformation Format (UTF) [38] string directly from the serialized bytestream. To deliver a payload, the delivery application implements *writeObject*, writing a UTF string to the bytestream. It also implements the *loadParams* method, which loads a JSON parameter string including one previously generated string payload into a Java object containing this parameter. The parameter is then used by *writeObject*, so that it is written whenever the object is serialized. The Serial Version UID of this class is not set, so it will be generated dynamically. The code of the class is

```
package oversecured.ovaa.objects;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class DeleteFilesSerializable implements Serializable {
    private static final long serialVersionUID = 0;
    transient Parameters1 parameters;
    public void loadParams(String jsonPayload) {
        Gson gson = new Gson();
        this.parameters = gson.fromJson(jsonPayload, new
           TypeToken<Parameters1>(){}.getType());
    };
    private void writeObject(ObjectOutputStream stream) throws
        IOException {
        stream.writeUTF(this.parameters.parameter_1);
    }
}
```

Listing 4: First serializable class definition

```
package oversecured.ovaa.objects;

public class Parameters1 {
    public String parameter_1;
}
```

Listing 5: First parameter class definition

presented in listing 4 and in listing 5.
MemoryCorruptionSerializable contains only one field to be serialized, the *private long ptr*. The Serial Version UID has already been set to 0 in the target application. The implemented *loadParams* method loads a JSON parameter string including one integer value, which is then placed into the *ptr* field. The code of the class is presented in listing 6 and in listing 7.
MemoryCorruptionParcelable contains the *writeToParcel* function, which calls *writeString* two times. The implemented *loadParams* method loads a JSON parameter string including two string values into a Parameters Java class and the implementation of *writeToParcel* is changed to write the two given parameters. The code of the class is presented in listing 8 and listing 9.

JSON parameter grammars were created based on the parameters in these classes. No default values were found, so the value *string* was used for strings and 0 for long values.

A function setting an Intent to match the Intent filter in the selected component in the payload delivery application was created dynamically. Three functions were created, which create an Intent and an instance of one of the Java objects required for the selected input points based on a given payload, then place it in the Intent, and call the function setting the Intent data for the target component. Then the functions launch the Intent.

```
package oversecured.ovaa.objects;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import java.io.Serializable;

public class MemoryCorruptionSerializable implements
   Serializable {
    private static final long serialVersionUID = 0;
    private long ptr;

    public void loadParams(String jsonPayload) {
        Gson gson = new Gson();
        Parameters2 parameters = gson.fromJson(jsonPayload,
           new TypeToken<Parameters2>(){}.getType());
        this.ptr = parameters.parameter_1;
    };
}
```

Listing 6: Second serializable class definition

```
package oversecured.ovaa.objects;

public class Parameters2 {
    public long parameter_1;
}
```

Listing 7: Second parameter class definition

```
package oversecured.ovaa.objects;
import android.os.Parcel;
import android.os.Parcelable;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;


public class MemoryCorruptionParcelable implements Parcelable
   {
    public static final Parcelable.Creator<
       MemoryCorruptionParcelable> CREATOR = new Parcelable.
       Creator<MemoryCorruptionParcelable>() {
        public MemoryCorruptionParcelable[] newArray(int i) {
            return new MemoryCorruptionParcelable[i];
        }

        public MemoryCorruptionParcelable createFromParcel(
           Parcel parcel) {
            return new MemoryCorruptionParcelable();
        }
    };
    public MemoryCorruptionParcelable() {

    }
    public Parameters3 parameters;

    public int describeContents() {
        return 0;
    }

    public void loadParams(String jsonPayload) {
        Gson gson = new Gson();
        this.parameters = gson.fromJson(jsonPayload, new
           TypeToken<Parameters3>(){}.getType());
    };

    public void writeToParcel(Parcel parcel, int i) {
        parcel.writeString(this.parameters.parameter_1);
        parcel.writeString(this.parameters.parameter_2);
    }
}
```

Listing 8: Parcelable class definition

```
public class Parameters3 {
    public String parameter_1;
    public String parameter_2;
}
```

Listing 9: Third parameter class definition

6.3 Testing Results

For each of the selected input types, a file of 300 payloads was loaded and the payloads were successfully delivered.

6.3.1 Deeplink

In the case of URLs on Android, this was detected by observing the successful launch of *DeeplinkActivity* in the logs. R1 is satisfied, as the deeplink target only includes one input point, which is the URL itself.

R2 is satisfied by the URL grammars, as probabilities were set to include the correct URL scheme with 0.95 probability. It is extremely important to prioritize the correct scheme, as deeplinks do not work otherwise.

Successful exploitation of the deeplink required the URL to use the deeplink scheme, include the path */webview* and the URL parameter *url*. The parameter value must be a URL which ends in the text *example.com*. A malicious URL such as *maliciousexample.com* is accepted and opened, which can be detected with a proxy.

Guessing these keywords and generating a payload with all of them together is extremely unlikely. Let n be the amount of terminals in the URL grammar for the symbols corresponding to the path, URL parameter and host. Even if every payload included a path, a parameter and a host, and each keyword had a probability of $\frac{1}{n}$ of being included, a correct payload still only occurs with a probability of $(\frac{1}{n})^3$, equal to $\frac{1}{n^3}$. In this case each keyword has a wordlist of size $n < 10$, but a larger number can be used to detect keywords better and increase the possible depth, with the tradeoff of correct keywords being even more rare. The problem was evident during testing, where the vulnerability could not be triggered without prioritization.

This problem prevents satisfying R3, since the correct keywords are required for reaching full depth in the code. The method was improved to get around this by enumerating all strings hardcoded in the application code, as explained in section 5.2, and by increasing their probability in the grammar. The depth requirement is satisfied by including strings found from the application code in payload generation.

In iOS, deeplinks were successfully opened but the same problem existed on a larger scale, because deeplinks are the only entry point for most functionality and therefore also vulnerabilities. However, R1 is then easily satisfied on iOS, as deeplinks give full coverage. URL parameters, included in the grammar, are used to sent data to the application.

6.3.2 Serializable Class

For Serializable classes, payloads successfully triggered vulnerabilities. Memory corruption was triggered with a random integer payload, showing that Serializable payloads satisfy R1 by setting field values within the application. R3 was easy to satisfy, as there was no deep processing of the serialized data in their case. Memory corruption was detected as a crash in the logs, as shown in listing 10.

A file was successfully deleted from the application files by supplying *DeleteFilesSerializable* with a path to the test file */data/data/oversecured.ovaa/files/test.txt*,

```
Fatal signal 11 (SIGSEGV), code 1 (SEGV\_MAPERR), fault addr 0
   xfff8022c in tid 4294 (FinalizerDaemon), pid 4281 (
   versecured.ovaa)
```

Listing 10: Memory error triggered by a payload

```
(agent) Called java.io.File.equals(java.lang.Object)
(agent) [836808] Called java.io.File.compareTo(java.io.File)
(agent) [836808] Called java.io.File.getPath()
(agent) [836808] Called java.io.File.getPath()
(agent) [293284] Called oversecured.ovaa.objects.
   DeleteFilesSerializable.readObject(java.io.
   ObjectInputStream)
(agent) [836808] Called java.io.File.exists()
(agent) [836808] Called java.io.File.isDirectory()
(agent) [836808] Called java.io.File.getPath()
(agent) [836808] Called java.io.File.getName()
(agent) [836808] Called java.io.File.delete()
```

Listing 11: Methods called by the application when a payload was delivered

```
FATAL EXCEPTION: main
Process: oversecured.ovaa, PID: 26474
java.lang.RuntimeException: Unable to start activity
   ComponentInfo{oversecured.ovaa/oversecured.ovaa.activities.
   DeeplinkActivity}: java.lang.RuntimeException: java.lang.
   ClassNotFoundException: string_1
```

Listing 12: Java crash due to invalid class name

which was placed into the file system before fuzzing. It was prioritized by the payload generation, because the file path was chosen specifically for file system vulnerability detection and did not need to be guessed. This satisfies R2. R1 was satisfied by correctly implementing the write to bytestream, which is considered in the enumeration process of the method. Method call based detection was also successful with the Frida-based Objection tool [45], which was used to detect the use of methods in the Java file representation class, as shown in listing 11.

6.3.3 Parcelable Class

For the Parcelable class, R1 was satisfied as both of the written parameters were used in the application. However, the first parameter is used before the second one. The first parameter caused a crash as it tried to initialize a class. It is detectable in the logs as shown in listing 12.

After this, it was observed that a class is loaded from user input, so the first parameter should be a valid class. By then prioritizing memory corruption payloads, which contain the valid class name *com.android.internal.util.VirtualRefBasePtr*, with probability 0.95 in the generation of the first parameter, the use of the second parameter and further depth were reached. This shows that R2 and R3 were satisfied. Another crash then reveals that JSON is used to initialize an arbitrary class.

6.3.4 HTTP Response

The HTTP response has many fields, which must satisfy R1, covering each field while retaining the structure. The recursive definition in listing 1 ensures that each

parameter is considered and can be replaced with a payload. This was confirmed during testing.

Prioritization and the requirement R2 were crucial for generating payloads with good coverage. Since each parameter normally has an equal probability for each expansion, and the payload and an empty object or array are possible expansions, each recursive layer only has a $\frac{1}{3}$ probability of occurring. In the test case, there is a JSON object with the *data* attribute as an array, including another JSON object with the *type* attribute as a string value. The probability of *type* occurring at all in a payload is only $(\frac{1}{3})^3$, equal to $\frac{1}{27}$. It is therefore quite rare for all parameters to occur at the same time in a payload, which causes R3 to not be satisfied. The payload, empty object and empty array were set to only occur with probability 0.05 each. Only in the case of a string, the probability of a payload is $\frac{1}{2}$. This ensured that payloads are inserted mostly in string parameters, but also occasionally in place of objects and arrays.

To satisfy R3, the JSON must remain very similiar to the original form, with only one parameter altered in a payload. This can be achieved by setting the probability of a payload occurring in the place of a string value as $\frac{1}{S}$, where S is the total amount of strings defined in the JSON structure.

6.3.5 Compatibility

The compability requirement R4 must be satisfied by the system properties on a higher level. It can be achieved by breaking down the system into different components. Each component is able to receive and send data in a uniform manner, in this implementation JSON is used as the main data format. Any program which takes JSON data as input or outputs JSON data is compatible with the fuzzing payload delivery system, and can be used to or extend replace the components that have been used in this implementation. The monitor also controls scheduling the payloads, which is required to use the current detection methods efficiently, so any additional detection methods can be synchronized with payload delivery in the same way.

To show the compatibility of payload generation components, it should be clear that any algorithm, which can identify potentially good payloads, can be used to generate future payloads that are similiar to the selected payloads. With the recent increase of applications in the field of machine learning and artificial intelligence, compatibility of payload generation is equivalent to how well new statistical models and technologies can be applied to it. Grammars make it possible to adapt to any technique by changing the probability distributions dynamically, and the grammars themselves can be changed. Parsers allow automatically setting the probability distributions to favor similiar outputs as in a given sample set, or to avoid similiar outputs. This reduces the problem to being able to recognize and rank good outputs and bad outputs based on responses found, which any other fuzzing system needs to also be able to do. In this case, detection method outputs as the effect of a payload must be converted into a heuristic value, which is then used to rank the payloads similiarly to any fuzzing algorithm. Machine learning is a potential solution to estimating such values well. Currently, the detection of a crash or a meaningful

change is simply evaluated by the user themselves.

6.4 System Analysis

The high level design of the system using language-based payload generation was a large part of the research. To evaluate the whole system, comparisons to different fuzzing systems can be made. Two general goals for the outcome were set in section 3.2, first being bringing mobile application fuzzing methods to the same level as web application fuzzing methods. The second was to reduce the manual work involved in mobile application security analysis.

The system shows progress in bringing mobile application testing closer to web application testing specifically in the application layer. The main input channel to the application in web testing are HTTP requests and in some occasions the WebSocket protocol, and messages are sent over TCP. Web application logic in the browser can be tested by fuzzing URL parameters and user input forms, or by directly fuzzing JavaScript functions. The progress made here is that similiar, unique input channels were identified for mobile applications, and a system for fuzzing them was designed. Unifying all input channels into one fuzzing system is an important step, because it allows for payloads and detection methods to be used together efficiently. Language-based fuzzing improves this even further, as different input languages are easier to combine and unlimited coverage is possible.

In addition to fuzzing for vulnerabilities, language-based payload generation has other advantages. Vulnerabilities suspected or found through code analysis should ideally contain a proof-of-concept payload to confirm and show the impact. Manual work required to do this is reduced when using the system, because the desired payloads can easily be set to occur at high probability, and delivery of payloads to the application has already been automated. The generalized version of the system also automates the steps of constructing the code required for the application inputs, but even if the setup is done manually, the clear definitions and steps compatible with the fuzzing system reduce the work required. It is clearly now easier to construct proof-of-concept attacks for vulnerabilities, so the system also improves the results of static analysis.

6.5 Generalized Method

A generalized method works without any additional setup on any Android or iOS application given the application file, requiring automation of the currently manual steps. It requires generalizing all components of the system model defined in section 5.1. This research contains both generalized components and specifically implemented components. The generalized components include the payload generator, the fuzzing algorithm and the dynamic analyzer, because their operation can easily adapt to any application. The static analyzer was shown to be able to generate grammars from any given JSON data and parameters in section 4.2, but this data is based on the modified Java classes and analyzed knowledge of the application. It is then clear that the only specifically implemented components are the static

analyzer and the monitor. To achieve generalization, the static analyzer should be fully automated and the monitor should be automatically configured independent of the target application.

In this research, the general idea of modifying Java classes to be compatible with the fuzzing parameters is presented in section 4.1. The transformation of Java classes into these modified classes needs to be implemented in the static analyzer as a program, which identifies the Serializable and Parcelable classes in decompiled application code and constructs the code of modified Java classes. This is likely to require different templates, from which the most suitable is chosen for each class. The biggest problem is with custom implementations in Serializable classes, where objects are written directly into the bytestream. The default values for parameters are also difficult to choose, if custom objects are present in class fields. In addition to constructing the Java classes, the static analyzer should automatically find entry points to components of the application by reading the configuration files as was done manually in this resarch, and construct the code required to send data to them. All useful data, such as URIs and other strings found by the static analyzer needs to be taken into account in the fuzzing system as a whole.

To generalize the monitor, a program is required which puts together all the information provided by the static analyzer. The payload delivery code based on application entry points and the Java classes must form a working Android application. It is important for the monitor to automatically understand how to route each payload correctly to the Java classes and Intents. The monitor needs to be able to relay useful data from the static analyzer to both the payload generation and the dynamic analyzer, so that they can be efficient in detecting vulnerabilities specifically from this application. The monitor controls what data is given to the other components to work with, and this approach ensures that the other components do not need to be designed to analyze any data themselves. All input from the user should happen through the monitor, so potential manual work is minimized and made as simple as possible.

Overall, the fuzzing system usually works best with some manual configuration, and is useful as a tool extending other analysis techniques. It is important to automate those components that can be automated without sacrificing the efficiency or coverage of the fuzzing process.

7 Previous Work

Previous works for finding application-level vulnerabilities include research on HTTPS response fuzzing [48], which this research expands on. My contribution also includes cross-application communication as a data input point, while previously only HTTPS responses were considered. This is not a trivial task, mainly because Android applications can process incoming data as Java classes, complicating the delivery of payloads considerably. The solution in this research enables full coverage of every possible input despite complex Java packing methods.

The previous work in [48] only considered four problems to be searched: no response, crashing, HTML content replacement and URL redirection. I consider many different types of vulnerabilities, which brings much larger coverage for security issues. New vulnerabilities can also easily be added to the configuration, as extensibility is an important concern for the developed method. The increased coverage in both input locations and vulnerability specifications brings improvements, but payload generation has also been optimized for application input languages. Defining mobile application input languages and vulnerability payloads together as formal grammars for fuzzing has not been done in any previous work.

Fuzzing Android cross-application communication inputs through Intents on the application level has been researched in [16], but the random data used was not very well structured and the case of Parcelable and Serializable classes was not implemented. Only crashes were detected, and specific vulnerabilities were not tested. The research in the thesis also expands on this work, by having clear targets and highly structured input data.

Besides language-based fuzzing, other approaches such as independent view fuzzing [49] have been researched, but they mostly focus on detecting potentially faulty states and general logic bugs whereas language-based fuzzing can detect the existence of known, exploitable vulnerabilities. Languages can also be integrated with many different fuzzing algorithms and techniques.

Drozer [50] is an open source mobile security tool developed by WithSecure. It acts as a malicious application and can send exploit payloads to other applications. On Android, it can be used to launch components with malicious cross-app functionality, and find vulnerabilities such as SQL injections. There is also a list of known vulnerabilities which can be exploited, but these are mostly related to the Android OS itself. Drozer has also not been maintained for 7 years, so it's exploits may be very outdated. It can be used as a tool for interacting with applications and testing components for SQL injections and content provider issues, but they are also found with the method developed in this research.

8 Conclusion

In this thesis, I have developed a method for security testing mobile applications using language-based fuzzing techniques. The method brings a more efficient approach to finding mobile application security vulnerabilities, as most tools focus on static

analysis of source code instead of testing an application runtime. Most previous dynamic methods focus on finding operating system level bugs or require manually constructing inputs. This solution is also generalizable, as input languages can be adjusted to match those of any application and source code is not a necessity for testing.

In conclusion, the method combines efficient payload generation with known high-risk security vulnerabilities, while maintaining compatibility with other tools and technologies.

This was achieved by detailed analysis of possible data input points into mobile applications, classifying vulnerabilities, their payload types and how they can be embedded in input data. Formalizing input points and payloads as grammars enabled the properties that are required for efficient fuzzing.

References

[1] Poller, A., et al. "Can Security Become a Routine? A Study of Organizational Change in an Agile Software Development Group." Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17). Association for Computing Machinery, New York, NY, USA, 2489–2503. https://doi.org/10.1145/2998181.2998191. 2017.

[2] Rogowski, R. et al. "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses," 2017 IEEE European Symposium on Security and Privacy (EuroS&P), Paris, France, pp. 366-381, doi: 10.1109/EuroSP.2017.39. 2017.

[3] Akhawe, D. et al. "Towards a Formal Foundation of Web Security," 2010 23rd IEEE Computer Security Foundations Symposium, Edinburgh, UK, pp. 290-304, doi: 10.1109/CSF.2010.27. 2010.

[4] Ahmed, M., Ibrahim, R. "A Comparative Study of Web Application Testing and Mobile Application Testing." Lecture Notes in Electrical Engineering. 315. 10.1007/978-3-319-07674-4_48. 2014.

[5] Hodovan, R., Kiss, Á., Gyimóthy, T. "Grammarinator: a grammar-based open source fuzzer." 45-48. 10.1145/3278186.3278193. 2018.

[6] Chen C., et al. "A systematic review of fuzzing techniques." Computers & Security, Volume 75, 118-137, ISSN 0167-4048, 2018.

[7] Steinhöfel, D. and Zeller, A. "Input invariants." In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022). Association for Computing Machinery, New York, NY, USA, 583–594. 2022.

[8] King, J. "Symbolic execution and program testing." Commun. ACM 19, 385–394, 1976.

[9] Choi, J. ”Grey-Box Concolic Testing on Binary Code” 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 736-747, 2019.

[10] Szydlowski, M., et al. ”Challenges for Dynamic Analysis of iOS Applications”. Camenisch, J., Kesdogan, D. (eds) Open Problems in Network Security. iNetSec 2011. Lecture Notes in Computer Science, vol 7039. Springer, Berlin, Heidelberg, 2012.

[11] Payet, E., Spoto, F. ”Static Analysis of Android Programs”. Information and Software Technology. 54. 439-445. 10.1007/978-3-642-22438-6_33. 2011.

[12] Ardito, L., et al. ”Effectiveness of Kotlin vs. Java in Android App Development Tasks”. Information and Software Technology. 127. 106374. 10.1016/j.infsof.2020.106374. 2020.

[13] Jiang, T., et al. ”Formal grammars and languages”. Algorithms and Theory of Computation Handbook, Volume 1. Chapman and Hall/CRC, 2009. 549-574.

[14] Ekanayake, N. ”Android Operating System”. 2018.

[15] ”The Java Programming Language, Fourth Edition”, Addison Wesley Professional, 0-321-34980-6, 2005.

[16] Sasnauskas, R., Regehr, J. ”Intent fuzzer: crafting intents of death.” Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA) (WODA+PERTEA 2014). Association for Computing Machinery, New York, NY, USA, 1–5. 2014.

[17] Chomsky, N. ”Formal properties of grammars”. Handbook of Mathematical Psychology Vol. 2. John Wiley and Sons, New York, 1963. 323-418.

[18] Aho, A. and Ullman, J. ”The theory of parsing, translation, and compiling”. Prentice-Hall, Inc., USA, 1972.

[19] OWASP. ”Mobile Top 10”. Available: https://owasp.org/www-project-mobile-top-10/, April 2024.

[20] OWASP. ”iOS Platform Overview”. Available: https://mas.owasp.org/MASTG/iOS/0x06a-Platform-Overview/, April 2024.

[21] OWASP. ”Bypassing Certificate Pinning”. Available: https://mas.owasp.org/MASTG/techniques/android/MASTG-TECH-0012/, April 2024.

[22] Oversecured. ”Android and iOS vulnerabilities.” Available: https://oversecured.com/vulnerabilities/, April 2024.

[23] Oversecured. "Memory corruption vulnerabilities on Android". Available: https://blog.oversecured.com/Exploiting-memory-corruption-vulnerabilities-on-Android/, April 2024.

[24] Oversecured. "Gaining access to arbitrary Content Providers". Available: https://blog.oversecured.com/Gaining-access-to-arbitrary-Content-Providers/, April 2024.

[25] Oversecured. "Android: Access to app protected components". Available: https://blog.oversecured.com/Android-Access-to-app-protected-components/, April 2024.

[26] Oversecured. "Android: Arbitrary code execution via third-party package contexts". Available: https://blog.oversecured.com/Android-arbitrary-code-execution-via-third-party-package-contexts/, April 2024.

[27] Oversecured. "Android security checklist: Theft of arbitrary files" https://blog.oversecured.com/Android-security-checklist-theft-of-arbitrary-files/, April 2024.

[28] Peles, O. and Hay, R. "One Class to Rule Them All: 0-Day Deserialization Vulnerabilities in Android". 9th USENIX Workshop on Offensive Technologies, 2015.

[29] Ramasamy, S. and V, V. "Preventing Structured Query Language (SQL) Injection Attacks in Mobile Applications". International Journal of Control Theory and Applications, 2016.

[30] Krishnaraj, N., et al. "Common vulnerabilities in real world web applications". doors 2023: 3rd Edge Computing Workshop, 2023.

[31] Mozilla. "JavaScript URIs". Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void#javascript_uris, April 2024.

[32] Flanders, M. "A Simple and Intuitive Algorithm for Preventing Directory Traversal Attacks". arXiv, 1908.04502, 2019.

[33] Akiyama, M., et al. "Analyzing the ecosystem of malicious URL redirection through longitudinal observation from honeypots", Computers & Security, Volume 69, Pages 155-173, ISSN 0167-4048, 2017.

[34] Choi, H., Kim, Y. "Large-Scale Analysis of Remote Code Injection Attacks in Android Apps". Security and Communication Networks. 2018. 1-17. 10.1155/2018/2489214.

[35] Zhou, Y., et al. "CLFuzz: Vulnerability Detection of Cryptographic Algorithm Implementation via Semantic-aware Fuzzing". Association for Computing Machinery, New York, 2023.

[36] Walshe, T. and Simpson, A. "An Empirical Study of Bug Bounty Programs". 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF), London, ON, Canada, 2020, pp. 35-44

[37] Backus, J. W. "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". Proceedings of the International Conference on Information Processing. UNESCO. pp. 125–132. 1959.

[38] Modified UTF-8. Available: https://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html#utf-8, April 2024.

[39] Jadx. Available: https://github.com/skylot/jadx, April 2024.

[40] Gson. Available: https://github.com/google/gson, April 2024.

[41] Ghidra. Available: https://ghidra-sre.org/, April 2024.

[42] Frida. Available: https://frida.re/, April 2024.

[43] BurpSuite. Available: https://portswigger.net/burp/, April 2024.

[44] Frida codeshare. Project ios-deeplink-fuzzing. Available: https://codeshare.frida.re/@ivan-sincek/ios-deeplink-fuzzing/, April 2024.

[45] Objection. Available: https://github.com/sensepost/objection/, April 2024.

[46] Ikram, M., Beaume, P., and Kâafar, M. A. "DaDiDroid: An obfuscation resilient tool for detecting android malware via weighted directed call graph modelling". arXiv preprint arXiv:1905.09136, 2019.

[47] Zhang, X., et al. "Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations". Forensic Science International: Digital Investigation, Volume 39. 2021.

[48] Huang, X., et al. "Fuzzing the Android applications with HTTP/HTTPS network data". IEEE Access. PP. 1-1. 10.1109/ACCESS.2019.2915339, 2019.

[49] Su, T., et al. "Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs". Proc. ACM Program. Lang. 5, OOPSLA, Article 156, 2021.

[50] Drozer. Available: https://github.com/WithSecureLabs/drozer, April 2024.

[51] The Fuzzing Book. Available: https://www.fuzzingbook.org/, April 2024.