Quokka

ACADEMIC PAPER

Detecting and Measuring Misconfigured Manifest in Android Apps

Detecting and Measuring Misconfigured Manifests in Android Apps

Yuqing Yang
The Ohio State University
yang.5656@osu.edu

Mohamed Elsabagh
Quokka*
melsabagh@quokka.io

Chaoshun Zuo
The Ohio State University
zuo.118@osu.edu

Ryan Johnson
Quokka*
rjohnson@quokka.io

Angelos Stavrou
Quokka*
astavrou@quokka.io

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

ABSTRACT

The manifest file of an Android app is crucial for app security as it declares sensitive app configurations, such as access permissions required to access app components. Surprisingly, we noticed a number of widely-used apps (some with over 500 million downloads) containing misconfigurations in their manifest files that can result in severe security issues. This paper presents ManiScope, a tool to automatically detect misconfigurations of manifest files when given an Android APK. The key idea is to build a manifest XML Schema by extracting manifest constraints from the manifest documentation with novel domain-aware NLP techniques and rules, and validate manifest files against the schema to detect misconfigurations. We have implemented ManiScope, with which we have identified 609,428 (33.20%) misconfigured Android apps out of 1,853,862 apps from Google Play, and 246,658 (35.64%) misconfigured ones out of 692,106 pre-installed apps from 4,580 Samsung firmwares, respectively. Among them, 84,117 (13.80%) of misconfigured Google Play apps and 56,611 (22.95%) of misconfigured pre-installed apps have various security implications including app defrauding, message spoofing, secret data leakage, and component hijacking.

CCS CONCEPTS

• **Security and privacy** → **Mobile and wireless security**; **Mobile platform security**.

KEYWORDS

Mobile security; security configuration

ACM Reference Format:
Yuqing Yang, Mohamed Elsabagh, Chaoshun Zuo, Ryan Johnson, Angelos Stavrou, and Zhiqiang Lin. 2022. Detecting and Measuring Misconfigured Manifests in Android Apps [-3pt]. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560607

*Quokka was formerly known as Kryptowire.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560607

1 INTRODUCTION

Android follows a declarative app deployment model where each app is required to declare certain configurations in a file named `AndroidManifest.xml` in the root directory of an app package (APK) [3]. This app manifest file describes essential information about the app to both Android marketplaces and the Android OS to guarantee proper publishing, installation, and execution of the app on an end-user's device. Among many others, an app manifest file declares a variety of important information, including the unique app package name, Android versions compatible with the app, app components and their security and access control settings, permissions requested by the app, and configurations necessary for libraries and features needed by the app.

Due to its importance to app security and reliability, an app manifest file must pass multiple checks by Android app development tools during app development [6], by Google Play during app publishing [20], and by the Android runtime during app installation and execution [5]. However, by examining the open-source code and documentation on how Android validates a manifest file, we found that the validation process does not provide systematic coverage of all possible manifest misconfigurations. While we cannot access the source code of Google Play, we notice from this documentation [11] that Google Play just filters the elements and attributes that are related to feature requirements and compatibility, so as to avoid the app being installed on an incompatible device. Consequently, this can lead to apps with misconfigured manifest files in the wild, creating security issues as witnessed by the disclosed CVEs (e.g., CVE-2017-16835 [1] and CVE-2017-17551 [2]). Moreover, our preliminary investigation showed that even some applications associated with world's top vendor, e.g., Amazon as shown in Figure 1, may also involve such mistakes, which may lead to concerning purchase replay attack, inflicting losses to the vendor.

As such, it is imperative to perform a large-scale study to understand the problem of the misconfigured manifest files, including the history of this misconfiguration problem, the prevalence of misconfiguration in current market, and the impact of these miconfigured manifest in the entire ecosystem, so as to raise the awareness from the community and draw insights to help mitigate this problem. To perform a systematic and automatic check of the misconfiguration, we utilize a standard approach to validate the XML files with XSD schema. To generate the XSD schema, we leverage the official documentation of Android manifest file, which is provided by Google because it is the only source and standard for developer's reference when creating manifest files. Putting it all together, we develop

ManiScope, an NLP-based context-aware analysis tool to identify the manifest entities and their constraints from the documentation, and then generate the XSD for the validation.

We have implemented ManiScope and tested it with 1,853,862 Android apps collected from Google Play between January 2020 and May 2020, and 692,106 pre-installed apps from 4,580 Samsung firmwares (which were released between September 2011 and January 2020) collected from SamMobile [18]. Our investigation revealed a worrying situation: for Googe Play apps, ManiScope detected 265,028 misconfigured elements in 230,330 apps and 718,207 misconfigured attributes in 428,440 apps (in total 609,428 unique misconfigured apps). For the pre-installed apps, ManiScope detected 1,731,451 misconfigured elements in 152,046 apps and 386,346 misconfigured attributes in 114,494 apps (in total 246,658 unique misconfigured apps). These results indicate a concerning prevalence of manifest misconfigurations across mobile apps, and these problems can date back to the very early Android version (as early as Android 2.0+). Moreover, we found that 84,117 (13.80%) of misconfigured Google Play apps and 56,611 (22.95%) of misconfigured pre-installed apps could have various security issues, ranging from component hijacking, data leakage, and app crashes, among others.

Contributions. We make the following contributions:

- We present ManiScope, a novel tool to extract manifest constraints from the Android documentation, build a manifest XML Schema, and detect misconfigurations in Android app manifests.
- We propose novel domain- and context-aware NLP techniques to extract manifest constraints from the documentation and handle ambiguities and incomplete sentences in the natural language texts of the documentation.
- We present a large-scale study on over 2.4 million apps and detected that about a third of these apps contain misconfigurations. We provide an analysis of the prevelance, history and the security threats of these misconfigurations and their root causes.

2 PRELIMINARIES

2.1 Android App Manifest File

An Android app is packaged as an archive file (APK) that contains app code, assets, certificates, along with an app manifest file called `AndroidManifest.xml`, which is an XML file that specifies app components (the building blocks of an app, such as `activity` and `receiver`), permissions, and various configurations needed for the proper execution of the app [3]. When installing an app, the Android PackageParser configures the app's metadata and runtime settings based on the configurations defined in the app's `AndroidManifest.xml` [5].

As illustrated in Figure 1, app manifest files are composed of XML elements. Each element has a start and an end tag, can have a number of attributes (e.g., attribute `android:name` at line 5 that sets the name of the `receiver` element), and can contain other nested elements. The elements are organized in a tree structure where a child element can belong to only one parent element (e.g., the `<intent-filter>` element at line 6 is nested in its parent element `<receiver>`).

```
01 <manifest package="com.example.app"...>
02  ...
03  <application ...>
04  ...
05   <receiver android:name="com.amazon.*">
06    <intent-filter>
07     <action
08      android:name="com.amazon.*.NOTIFY"
09      android:permission="com.amazon.*.Permission.NOTIFY">
11     </action>
12    </intent-filter>
13   </receiver>
14  ...
15  </application>
16  ...
17 </manifest>
```

Figure 1: A simplified excerpt from the manifest file of a real-world app containing a misplaced permission attribute. Due to the severity of this misconfiguration, we redacted the name of the app and the vulnerable component until the issue is patched.

2.2 Misconfigurations in Manifest Files

The structure of a manifest XML tree specifies the relative positions of the manifest elements, though it does not enforce any particular occurrence constraints. Elements and attributes in an Android app manifest file can be required or optional, and some elements can also occur multiple times. For example, according to the Android Manifest Documentation [3], an `<action>` element *must occur at least once* inside an `<intent-filter>` parent element.

When developing Android apps, developers have to manually configure app manifest files, though there are some tools to partially automate some of the configurations. Such manual configurations can certainly introduce errors, as evidenced by the example in Figure 1 in which the `android:permission` attribute, which is supposed to declare the permission required to access the `<receiver>` component, is incorrectly placed in the `<action>` element instead of `<receiver>`. As a result, the `receiver` component is left unprotected at runtime, allowing arbitrary apps to access and invoke its functionality.

To avoid misconfigurations, developers must clearly understand the XML Schema of app manifest files, i.e., the correct structure and constraints governing elements and attributes in a manifest file. In general, an XML Schema describes three classes of requirements [19], violating any of which causes misconfiguration: (1) Manifest vocabulary and structure, describing what the valid elements and attributes are and where exactly they can be placed. (2) Occurrence constraints dictating how many times an element or an attribute can appear. (3) Valid attribute values and their data types. Misconfigurations resulting from violating these requirements can be classified based on their root causes into the following categories:

- **Misplaced elements and attributes**, which can be caused by (1) an element exceeding the upper bound of an occurrence (e.g., can only appear once but appeared multiple times), or (2) an element placed in an unexpected parent (e.g., if `<action>` element has an invalid parent `<receiver>`), or (3) an attribute declared inside a wrong element (e.g., `android:permission` in Figure 1).
- **Absent elements and attributes**, which occurs when a required element or attribute is missing, i.e., violating its lower bound occurrence constraint.

```
01  <xs:element name="intent-filter">
02    <xs:complexType mixed="true">
03      <xs:sequence>
04        <xs:element ref="action" minOccurs="1" />
05        <xs:element ref="category" />
06        <xs:element ref="data" />
07      </xs:sequence>
08      <xs:attribute name="autoVerify" type="xs:string"/>
09      ...
10    </xs:complexType>
11  </xs:element>
12  <xs:element name="action">
13    <xs:complexType mixed="true">
14      <xs:sequence>
15      </xs:sequence>
16      <xs:attribute name="name" type="xs:string"/>
17      ...
18    </xs:complexType>
19  </xs:element>
```

Figure 2: Example XSD schema snippet for detecting the misconfiguration in Figure 1.

- **Unexpected elements and attributes**, which can be caused by an element or attribute that does not appear in the valid manifest vocabulary, e.g., an undefined `<foo>` element or a misspelled element in the app manifest file.
- **Wrong attribute values**, which can be caused by an attribute value that does not satisfy the required data types or allowed data values for the attribute, e.g., value `true` or `false` is misspelled for a Boolean type.

Note that a misplaced element or attribute could also be categorized as absent. For instance, if `<action>` is misplaced inside `<data>`, which is required by `<intent-filter>`, it is identified as missing under `<intent-filter>` and misplaced under `<data>`.

2.3 Approaches for Validating Manifest Files

App manifest files must be validated to ensure their correctness. In fact, Google provides a number of tools for this purpose. In particular, at development time, Android Studio checks the manifest XML tree for the absence of some critical elements and attributes [6]. During the publishing phase, Google Play checks the app manifest file and applies filters on special compatibility elements (e.g., `<compatible-screen>`) to decide which devices are compatible with an app [20]. At installation time, the Android PackageParser parses the app manifest file in the APK, checks for required elements, and configures the app runtime accordingly [5].

A systematic and well-known approach to validate a manifest XML file is through the use of a corresponding XML Schema file that defines the constraints on the structure and content of the XML file. However, by checking the source code of AOSP [4], we did not find any XML Schema for app manifest files. Instead we found that AOSP uses hand-rolled code to validate manifest files [5]. We found that while AOSP validates all manifest attribute values and their data types [8], it uses ad-hoc constraints to validate the manifest structure itself (e.g., only checking for occurrence of certain elements and attributes). These hardcoded checks result in incomplete coverage since hand-rolling a complete XML validator that can capture all possible cases of misoccurrences is an exhaustive and error-prone task. As a result, many apps end up on the market with critical misconfigurations, as shown in Figure 1.

Therefore, to systematically validate app manifest files, we need to construct the XML Schema for Android app manifests and then perform the validation using the schema file. In particular, we need to know both the structure and occurrence constraints of elements and attributes of an Android manifest file, where the structure refers to the specific child and parent elements at each particular position of the XML tree and their corresponding attributes, and occurrence constraints refer to the upper and lower bounds of the occurrences of a child element in the tree, i.e., whether it is optional or required and how many times it can appear under the same parent.

After obtaining the structures and occurrence constraints of XML elements and attributes, various XML Schema languages, such as Document Type Definition (DTD) [12], Relax-NG [17], Schematron [13], and XML Schema Definition (XSD) [45], can be used to develop a specification using these structures and constraints to validate XML files. XSD is the most popular one among these schema languages since it is also written in XML, offers a strong set of specification facilities, and is widely supported by XML parsing packages for many programming languages [7, 21]. XSD supports various features that can be used to directly describe the correct structure and constraints of Android manifest elements and attributes: it can declare valid child elements and attributes of each element, minimum and maximum occurrence of elements, and whether an attribute is required or optional, and so on. An example of an XSD file is shown in Figure 2. All the tag names in XSD files begin with common prefix `xs:` since they all belong to the XML Schema (XS) namespace. Manifest elements such as `<intent-filter>` are declared one-by-one using `<xs:element>`. It can be noticed that an element is an `<xs:complexType>` if it contains both child elements and attributes. Child elements are then specified inside `<xs:sequence>` as `<xs:element>` references, and their number of occurrences are specified using the `minOccur` and `maxOccur` attributes. Similarly, valid attributes are specified in `<xs:attribute>` schema elements. For a full treatment of XSD, we refer interested readers to XSD definition [45].

3 OVERVIEW

The goal of this study is to understand the prevalence, history and security impact of the misconfiguration of Android manifest files. As such, we need to first generate the validation schema for the manifest files. To do so, an intuitive approach is to extract manifest constraints by analyzing the documentation as it is the official guide used by app developers to develop manifest files. Unfortunately, this is still non-trivial, requiring overcoming multiple key challenges as discussed in the following.

3.1 Challenges

C1: Identifying Manifest-Related Documentation Pages. To automatically extract constraints from the documentation, the first challenge is to identify the documentation pages relevant to app manifest files. Currently, there are over 1,000 HTML pages in the latest version of the Android documentation, and they specify constraints not only related to developing Android apps and configuring manifest files, but also to other XML files such as the Android resource XML file, which share the similar structure as the manifest documentation. As such, we need to avoid capturing descriptions

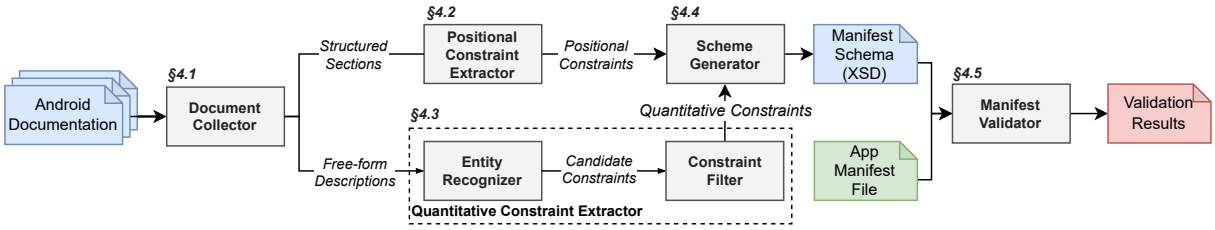

Figure 3: The Workflow of ManiScope.

in every documentation page; otherwise, a large amount of constraints irrelevant to manifest file configuration may be erroneously extracted.

C2: Handling Ambiguity and Incompleteness in Manifest Descriptions. It is challenging for an automated system to deal with ambiguities in the descriptions of manifest elements and attributes since they are written in free-form text. Compared with prior NLP-based document mining techniques (e.g., [22, 32, 43]), we need to perform more complex tasks because we need to not only identify manifest entities referred to in a sentence and determine their relationships (parent or child), applicable positional and quantitative constraints (see §2), but also translate them to valid XSD.

Fortunately, we find that sentences discussing manifest elements and attributes have common structures. First, sentences describing how to configure manifest entities are mostly imperative sentences that use modal verbs, whereas descriptions specifying maximal occurrences use numerical words to emphasize that an element is unique (e.g., '*there is only one...*'). Second, we find that the subjects and objects of complete sentences refer to the parent and child elements respectively. For example, in sentence *'An* `<intent-filter>` *element must contain one or more* `<action>` *elements'*, `<action>` is the current element and `<intent-filter>` is its parent. Although in some contexts, the parent element may be omitted for brevity (e.g., '*The name must be specified*'), the overall structure of subject–verb–object (SVO) remains.

C3: Performing Context-Aware and Domain-Guided Parsing. Since sentences in the documentation may omit parent elements for brevity, there may be relevant manifest constraints not captured by the above two sentence structures. In addition, specifications irrelevant to manifest constraints may be mistakenly identified as manifest constraints when they use sentences with SVO structure. Therefore, we need to carefully filter out irrelevant sentences by reasoning about the context in which a sentence occurs and also using the domain-knowledge extracted to guide the filtering.

Interestingly, we noticed that the structural information in the manifest documentation can help build domain-knowledge and identify sentence contexts. Specifically, (1) the omitted entities are often in a structure context. For example, there is a sentence in the documentation for `<activity>` *'The name must be specified'*, the omitted `<activity>` is exactly the name of the documentation. (2) The element and attribute names in the titles of each section (such as `contained in`) yields a dictionary as well as the structure of manifest-related entities, which can be used to build the domain knowledge and filter out the irrelevant and misplaced ones.

<action>
syntax:
<action android:name="*string*" />
contained in:
<intent-filter>
description:
Adds an action to an intent filter. An <intent-filter> element must contain one or more <action> elements. If there are no <action> elements in an intent filter, the filter doesn't accept any Intent objects. See Intents and Intent Filters for details on intent filters and the role of action specifications within a filter.
attributes:
android:name
The name of the action. Some standard actions are defined in the Intent class as ACTION_*string* constants...

Figure 4: An excerpt from Google's official documentation for the `<action>` element.

3.2 Problem Scope and Assumptions

In this work, we focus on identifying and measuring the unexpected, misplaced, and missing manifest configurations by using constraints extracted from the Android manifest documentation. We assume that the manifest documentation is the ground truth that specifies all constraints needed for validating a manifest file. We also assume that input manifest files have valid value types since this is a requirement for compiling an app and is already handled by Android development tools. Finally, extracting constraints from undocumented manifest elements or attributes that may be internally supported by Android is out of scope.

4 DETAILED DESIGN

The workflow of ManiScope is shown in Figure 3. At a high level, it contains five key components: *Document Collector* (§4.1), *Positional Constraint Extractor* (§4.2), *Quantitative Constraint Extractor* (§4.3), *Scheme Generator* (§4.4), and finally *Manifest Validator* (§4.5). In this section, we present the detailed design of these components.

4.1 Document Collector

As described in **C1**, we need to automatically collect the documentations related to the manifest file, and extract the structured sections and descriptions for positional and quantitative constraint extraction, respectively. To avoid overly capturing the irrelevant elements and attributes when only using the structure of a document to determine whether it is related to an app manifest, we use a recursive top-down traversal algorithm to identify the attributes and elements related to manifest descriptions. In particular, the *Document Collector* traverses the documentation pages by starting from the documentation page of the root element `<manifest>`, then recursively visiting the documentation pages of each child element.

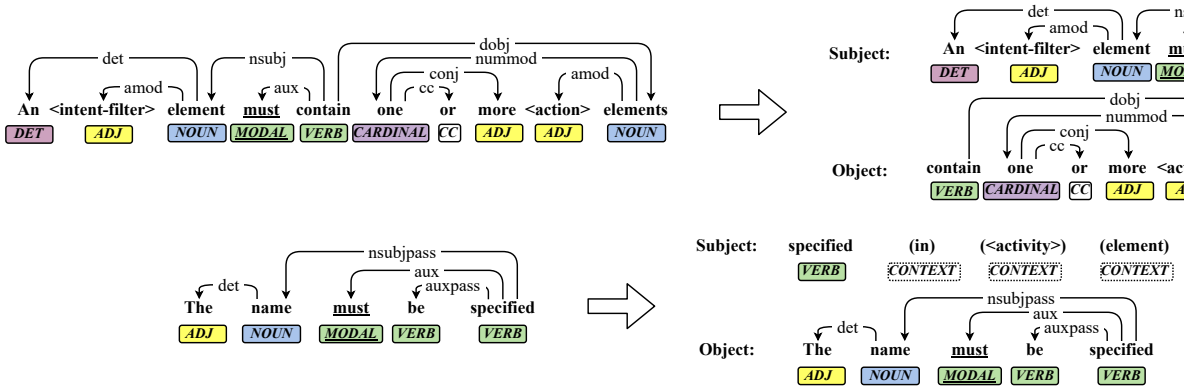

Figure 5: An example illustrating the dependencies between the words in a sentence, and how the documentation sentences are divided into subject and object clauses. ManiScope partitions the sentences into subject and object phrases, and leverages the dependencies to recognize the elements and attributes with a state machine based parsing approach.

As a result, it fetches manifest documentation pages for all elements that appear in the manifest file.

4.2 Positional Constraint Extractor

Given the collected manifest documentation pages, our *Positional Constraint Extractor* parses each documentation page and extracts child elements under '`can contain`' and '`must contain`' sections and attributes under `attributes` sections, respectively. These child elements and attributes are used to construct positional constraints, i.e., valid child elements and attributes for each parent element. For example, when parsing the document example in Figure 4, there is an `android:name` in the `attributes` section; therefore, we infer that `<action>` can have a child attribute of `android:name`. We also infer that the `<action>` element has no child elements because there is neither '`can contain`' nor '`must contain`' sections in its documentation page. When all the positional constraints are extracted, the parser generates a dictionary of all of the names of valid elements and attributes, which is used for filtering out non-manifest related constraints that may be mistakenly identified by the NLP parser. In addition, the parser also extracts descriptions of elements under the `descriptions` section, and attributes under each attribute name, which are required to extract quantitative constraints as described next.

4.3 Quantitative Constraint Extractor

Since the quantitative constraints are located in descriptions, our *Quantitative Constraint Extractor* uses NLP techniques to extract these constraints from free-form sentences in the descriptions. However, these natural language sentences are usually ambiguous and incomplete. To deal with the challenges of complex sentences (**C2**) and improve extraction precision (**C3**), we design two sub-components: (1) *Entity Recognizer* (§4.3.1) to identify manifest entities (i.e., elements and attributes) and handle ambiguities, and (2) *Constraint Filter* (§4.3.2) to filter out non-manifest related constraints.

4.3.1 Entity Recognizer. As discussed in **C2**, to extract constraints from free-form sentences, we need to extract manifest entities, their relationships, and handle missing references. To illustrate these challenges, we present two sentences in Figure 5 with a normal

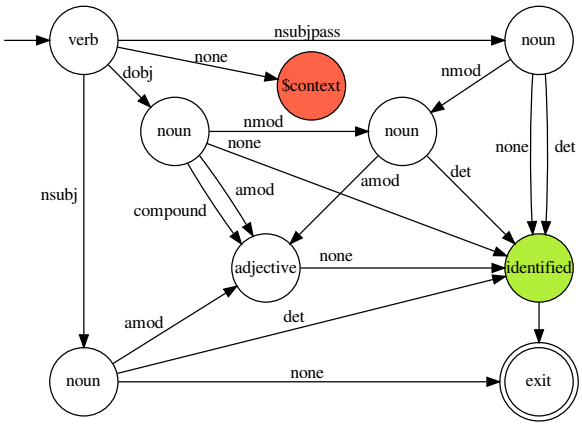

Figure 6: The finite state machine (FSM) for identifying elements, attributes, and $`context`.

voice (containing `nsubj` dependency) and a passive voice (containing `nsubjpass` dependency). The first sentence is written in normal voice, and it suggests that there is a minimum constraint for the child element `<action>` in the parent element `<intent-filter>`. We observe that these sentences often appear in Subject-Verb-Object structure. For instance, the subject phrase could be '*An <intent-filter> element*', and the object phrase could be '*one or more <action> elements*'. Therefore, we can extract the parent and child entity from the subject phrase and child phrase, respectively. However, since these phrases still contain complex structures such as modifiers and conjectures, we still need to locate the exact word such as *<intent-filter>* from these phrases, and to handle sentences where object phrases are omitted (e.g., '*The name must be specified*'). To this end, we extract this information by first (*i*) recognizing sentence dependencies using a Finite State Machine (FSM), then (*ii*) handling missing entities using contextual information.

(I) Recognizing Sentence Dependencies. We observe that in sentences specifying manifest constraints, the parent and child manifest entity appear in subject and object phrases, respectively (e.g., the child element `<action>` is in the object phrase *'one <action> element'*). Therefore, our *Entity Recognizer* extracts the parent and child entity by identifying the dependencies of subject and

Table 1: Context filtering rules of ManiScope.

Rules to filter out non-manifest constraints			
ID	**Type**	**Rule to filter out candidate constraint**	**Insights**
R1	context	$\neg parent.inSentence() \wedge \neg child.inSentence()$	If both manifest parent and child are not mentioned in sentence, it is not likely to relate to manifest constraint.
R2	context	$\neg dictionary.contains(parent) \wedge \neg dictionary.contains(child)$	If neither parent nor child is in manifest dictionary of element and attribute names, it is not a manifest constraint
R3	context	$child_{ex} \notin parent_{ex}.children$	If extracted child entity does not exist in the parent's valid manifest children, it is not a valid manifest constraint.
R4	sentence	'$advcl$' $\in sentence \vee$ 'acl' $\in sentence$	The adverbial clause of C in 'A must specify B when C' voids the constraint that A 'must contain' B.
R5	word	'$nummod$' $\notin sentence \wedge (w \notin sentence\ for\ w \in \{modal\ verbs\})$	If sentence contains neither nummod ('*only* **one**') nor model verbs ('**must** *contain*'), there is no occurrence constraint.
Rules to identify misspelled elements and attributes			
ID	**Type**	**Rule to identify misspelled entities**	**Insights**
R6	attribute	$\exists a \in attrs : a.replace(\text{'android:'}) = name.replace(\text{'android:'})$	Attributes may fail to include android namespace prefix: if *android:* is replaced, the postfix is the same.
R7	element	$\exists e \in elems : e.lower() = name.lower()$	Characters may be mistakenly capitalized: when all being converted to all lower case, the name is in dictionary.
R8	attribute	$\exists a \in attrs : a.lower() = name.lower()$	Characters may be mistakenly capitalized: when all being converted to all lower case, the name is in dictionary.
R9	element	$\exists e \in elems : 0 < distance\,(e.lower(), name.lower()) \leq \alpha$	If edit distance between encountered name and an entity in dictionary is less than α, it is a typo.
R10	attribute	$\exists a \in attrs : 0 < distance\,(a.lower(), name.lower()) \leq \alpha$	If edit distance between encountered name and an entity in dictionary is less than α, it is a typo.

object phrases until it finds a manifest entity or aborts the parsing based on the FSM, according to the dependency encountered at each word. We first identify dependencies used in the extraction procedure, including subjects (nsubj, nsubjpass), direct objects (dobj), adjective and noun modifiers (amod, nmod), compound statements (compound), and determiners (det) such as the name of an element. We then process these dependencies using the FSM to trace dependencies and identify entities as shown in Figure 6.

In the following, we discuss how our FSM-based approach works using an example of extracting the parent element *<intent-filter>* from the first sentence in Figure 5. Specifically, as shown in Figure 6, starting at the state verb (points to the word *'contain'* in the input as shown in Figure 5), our *Entity Recognizer* first moves to the noun state (points to word *'element'*) based on the state transition of nsubj. Next it moves to adjective state through amod and points to the word *'<intent-filter>'*. Since there are no more dependencies according to the parsed dependencies illustrated in Figure 5, the *Entity Recognizer* moves to a special identified state through the none edge. A none edge is a special edge where the state cannot be transferred. As such, when reaching the identified state, we successfully identify the word *'<intent-filter>'* as a manifest entity. If there is no object phrase in the sentence (e.g., no object phrase after *'specified'* in *'The name must be specified'*), the *Entity Recognizer* regards the corresponding entity as missing and holds its processing until more context information is collected, which will be handled in the next step. If the *Entity Recognizer* moves to the exit state without reaching the identified state, the tracing process aborts and no constraint is extracted from the sentence.

(II) Identifying Context Information. Due to the complexity and ambiguity of sentences, there is a chance where manifest-related constraint is not uncovered by our *Entity Recognizer*. In general, there are two scenarios where a sentence containing manifest constraints may be missed: 1) when the sentence has a missing entity that needs context information to be resolved (e.g., *'The **name** must be specified'*), and 2) when the identified word does not point to a specific manifest element or attribute (e.g., *'**this element** must be placed inside the <manifest> element'*). Therefore, we need to handle these incomplete and ambiguous manifest entities to avoid missing manifest constraints. To accomplish this, we notice that contextual information in documentation sections and paragraphs provide enough hints for inferring these missing entities.

- **Section-level Context.** Section-level context refers to information about element and attribute names associated with section titles in the documentation . For example, if the sentence '*The name must be specified*' appears in the description of the android:name attribute in the documentation section for <activity>, we can associate it with the <activity> element as its attribute. When a parent entity is missing, we associate the parent entity with the element name in the title of documentation (because only elements can be parent entities that contain child elements or attributes). When a child entity is missing, we associate the entity with the nearest section context: if the sentence is in the description of an element, we associate the entity with the element name; if it is in an attribute description, we associate it with the attribute name.
- **Paragraph-level Context.** At the beginning of paragraphs, we observe that a key sentence is often used to summarize the meaning or functionality of an element or attribute. As such, we utilize this context to improve the constraint extraction by identifying the subject and object from the first sentence of the paragraph (taking sentence dependencies into account). Of course, not all paragraphs provide contextual information in the first sentence, and non-manifest related information may be mistakenly generated. For example, in the documentation of the android:backupAgent attribute under the <application> element, the first sentence says *'The name of the class that implements the application's backup agent'*. Although the sentence merely indicates that the attribute is associated with a backup agent class in the source code, the context information may be mistakenly extracted as backupAgent. As a result, when we later encounter the sentence *'The name must be specified'* in the context of android:backupAgent we may identify the child to be 'name' but mistakenly identify parent as backupAgent, which is not a valid manifest entity. Hence, it is vital for utilizing the knowledge we extracted about manifest file to filter out these non-manifest constraints to avoid mistakes in the schema.

4.3.2 Constraint Filter. As discussed in **C3**, when extracting manifest constraints by parsing sentence structures, non-manifest constraints can appear in sentences with similar structures. Meanwhile, constraints not related to manifest may occur when we infer missing entities from contextual information. For example, in the description of android:label in <activity> documentation, *'The label is displayed on-screen when the activity must be represented to the user.'* has a similar 'subject-verb-object' structure: written in passive voice with a modal verb *'must'*, our *Entity Recognizer* identifies the child entity <activity> from the subject phrase, then extracts the missing parent from context information in the section title (i.e., <activity>). Subsequently, our *Entity Recognizer* would

extract a constraint that says '<activity> must be in <activity>', which is of course incorrect. As such, we need to filter out these erroneously-extracted constraints, and have designed five rules as shown in Table 1 to filter out the non-manifest constraints at three levels: context, sentence, and word.

- **Context Filter**. The context filter uses the contextual relationship between the parent and child entity to filter the non-related constraints. There are three rules used by this filter: **(R1)** When extracting constraints from broken phrases and sentences that do not contain any manifest entities, our recognizer may treat both the parent and child as missing and extract them from the context. However, there may be sentences completely irrelevant to manifest constraints where both parent and child entities are mistakenly inferred from the context. Hence, we need to focus on sentences containing at least one entity explicitly related to manifest (not inferred from contexts). As such, the constraints where both the parent and the child are extracted from contextual information need to be filtered out. **(R2)** As we focus on manifest-related constraints, it is natural that we force all identified parents and children to be contained in the manifest dictionary. **(R3)** In addition, we need to further ensure that extracted child is within the valid children list of the extracted parents. For instance, if a parent is `action` and a child is `<intent-filter>`, this is not a valid manifest constraint because we know from manifest dictionary that `<intent-filter>` cannot be a child of `<action>`.
- **Sentence Filter.** On top of contextual information, the sentence structures also provide hint for improving the accuracy. Particularly, in rule **R4**, we use sentence structure to filter sentences with noun (`acl`) or adverbial clauses (`advcl`) that voids occurrence constraints in main clauses such as 'must have'. For example, in *'You **should always** declare this attribute **if** you want to configure [...]'*, although it seems to be a minimal constraint because this is an imperative sentence with a phrase *should always*, the adverbial clause '*if you want to configure*' have voided such minimal requirement because it indicates that the attribute is mandatory only when the developer wants a certain configuration to be effective, whereas it is optional if developers do not want the configuration. Thus, the attribute mentioned in such a sentence is still optional in the manifest file.
- **Word Filter**. We also utilize words in sentences to reduce errors in occurrence constraint extraction, both for minimal and maximal constraints. On one hand, model verbs that carry strong tone like 'must' have to appear to clearly convey the minimal constraints ('must have' constraints). Therefore, we systematically checked all the modal verbs, and found only *must* and *should* conveys such strong tone, whereas other modal verbs can merely convey suggestions or predictions, such as *will* and *may*. On the other hand, numerical modifiers, when accompanied by model verb, help identifying maximum constraints. For example, in *'Only one instance of the <compatible-screens> element is allowed in the manifest'*, the manifest entity `<compatible-screens>` has a numerical modifier *one*. Therefore, it specifies that the maximum of the element is 1. As such, the word filter filters out non-manifest constraints with a set of modal verb keywords and the numerical modifier dependency `nummod` **(R5)**

4.4 Schema Generator

With positional constraints and quantitative constraints extracted and reformed into structured data, we then generate the XSD file for validation. In particular, the positional constraints are transformed by declaring each element with `xs:element` and then listing its child elements in `<xs:element>` and attributes in `<xs:attribute>`, respectively, e.g., in the declaration of `<intent-filter>` at line 1 in Figure 2, it contains references to child elements such as `<action>` at line 4, and attributes such as `android:autoVerify` at line 8 (which is declared at line 11). With the structure of elements and attributes being constructed in XSD, quantitative constraints are generated by setting `minOccurs` and `maxOccurs` for elements, and `required` for attribute (no `maxOccurs` for attributes as they are unique by nature). For example, the minimum occurrence of `<action>` is 1, and therefore the `minOccurs` of `<action>` is set to 1.

4.5 Manifest Validator

With the generated XSD schema, our *Manifest Validator* validates an app manifest file by detecting three types of misconfigurations: missing, misplaced, and unexpected. Missing entities are identified when the validator finds an element or attribute missing. Misplaced and misspelled entities, however, are both reported as unexpected keywords, so we need to compare the related element or attribute name with the manifest dictionary. If the entity is a valid manifest name, it is considered misplaced; otherwise, the entity name is misspelled. However, although our validator can detect all the unexpected attributes and elements, they are not always misspelled by developers. For example, compilers may add attributes to provide information of the compiler, and there may be system-only elements and attributes that do not appear in the documentations. As such, to avoid false-positives of identifying these manifest entities as "misspelled", we only focus on the following three types of misspelling errors:

- **Prefix Errors.** This error occurs when developers forget to add or mistakenly add the `android:` prefix for an attribute (e.g., `android:package` v.s. `package`, and `android:name` v.s. `name`). To identify this type of error, we remove the `android:` prefix of the encountered attribute name and compare the attribute name to attributes names in the manifest dictionary (**R6**).
- **Capitalization Errors.** A capitalization error occurs when the name of an element or attribute is mistakenly capitalized (e.g., `meta-data` v.s. `Meta-Data`). To identify such errors, we match the lowercase prefix-free strings of unexpected names to names in the manifest dictionary (**R7** and **R8**).
- **Typos.** To identify misspelled element or attribute names (e.g. `meta-data` v.s. `mata-data`) we compute the Levenshtein edit distance between an unexpected name and names in the manifest dictionary and check if it is below a certain threshold α, indicating the two words are highly similar (**R9** and **R10**). This threshold must be larger than 0, because no typos will be identified otherwise. However, if this threshold is set too high, it may introduce a large amount of false-positives (e.g., the distance between unexpected name `tag` and a valid manifest element name `data` is 3, and hence if the distance is set too high, our tool will regard the `tag` as a misspelled). To minimize possible false-positives, we set $\alpha = 1$ as default value for our tool, though it can be configured by users.

5 EVALUATION

We have implemented ManiScope in Python. For documentation parsing, we used the lxml [14] and BeautifulSoup4 [9] libraries. To extract grammatical structures from sentences, we used the NLTK CoreNLP Parser 3.9.2 [26]. We evaluated ManiScope on 1.8 million Android apps downloaded from Google Play between January 2020 and May 2020, and 0.6 million pre-installed apps collected from 4,580 Samsung firmware (released between September 2011 and January 2020) from SamMobile [18]. We used axmlparserpy [16] to decode the binary manifest file of each APK into plain-text XML. Our experiments were carried out on a laptop running Ubuntu 18.04.1 with 8 GB RAM and an Intel Core i7-8500U CPU. In this section, we first present our evaluation results of schema extraction in §5.1. Then, we present our findings with regard to misconfigurations in §5.2. Lastly, we provide statistics on security-related misconfigurations in §5.3.

5.1 Manifest Constraint Extraction

(I) Extraction Result. We first present how ManiScope performs when provided with the Android documentation. Since it is a fully automated system, it can parse all Android documentation including the historical ones. As such, we tested ManiScope with 20 different Android documentation from Android developers website from the most recent one (after 7.1.2) to the oldest available one, namely Android 1.6, and this result is reported in Table 2. Note that the source code of the historical documentation after 7.1.2 is no longer published on public Google repositories, and we obtain the most recent one by directly fetching the online HTML files.

In particular, as illustrated in the first row for the most recent documentation, ManiScope collected 26 documentation files related to manifest declaration, and identified in total 348 sections containing 849 paragraphs. When printing them in a format preserved manner (they are organized in a structure), we obtained 190 pages. Among the paragraphs parsed, ManiScope found that 1,326 sentences are written in normal voice and 256 are written in passive voice. Additionally, there are 404 phrases that do not have nominal subjects, either in normal voice or passive voice, which are identified as simple phrases rather than complete sentences. Our *Constraint Filter* filtered over 90% of non-manifest related constraints through context-filtering rules, and the word filter rules filtered out additional 1.3% of non-manifest constraints, and eventually it obtained 254 manifest constraints for 28 elements and 125 attributes.

(II) The Evolution of Manifest Documentations. Being able to analyze the historical manifest documentation, we can draw insights such as how they evolved. As such, we quantified such evolution by presenting the difference between two adjacent version of manifest documentations, as shown in Figure 7. First, we observe that the total sentences of manifest constraints, although added or removed, are constantly growing, where the growth rate can range from 0% to over 50%. Second, we notice that during updates, sentences may often be removed with new sentences added, be those removal of deprecated elements or attributes or changes made to descriptions. Interestingly, we also observe that fixing for some typos that eventually caused confusion among developers

Table 2: Constraint Extraction Statistics of ManiScope. (vers. = version, sect. = section, para. = paragraph, constr. = constraints, extra. = extracted)

	Documentations Parsed					Sentences Recognized			Constraints Filtered			Constr.
Vers.	files	pages	sect.	para.	words	phrase	normal	passive	context	clause	word	Extra.
7.1.2+	26	190	348	849	28,765	404	1,326	256	2,379	139	34	254
7.1.2	26	158	308	687	25,585	361	1,135	235	2,104	126	21	219
7.1.1	26	158	308	687	25,585	361	1,135	235	2,104	126	21	219
7.0.0	26	157	305	672	25,292	358	1,115	232	2,078	125	21	216
6.0.1	26	148	302	665	25,294	361	1,119	233	2,094	122	22	217
6.0.0	26	148	302	665	25,294	361	1,119	233	2,094	122	22	217
5.1.1	26	148	301	656	25,025	362	1,108	227	2,076	123	21	216
5.1.0	26	148	301	656	25,025	362	1,108	227	2,076	123	21	216
5.0.0	26	146	298	643	24,592	352	1,094	224	2,058	122	20	213
4.4.4	26	143	292	612	22,846	340	1,006	210	1,900	120	19	200
4.4.3	26	143	292	612	22,846	340	1,006	210	1,900	120	19	200
4.4.2	26	143	292	612	22,846	340	1,006	210	1,900	120	19	200
4.1.2	26	138	286	589	22,009	321	971	208	1,834	115	14	194
4.1.1	26	138	285	585	21,867	317	963	207	1,821	115	14	193
4.0.4	26	128	283	598	23,235	348	1,019	218	1,933	122	15	191
2.3.7	24	109	262	514	19,552	288	881	186	1,651	109	12	178
2.3.6	24	109	262	514	19,552	288	881	186	1,651	109	12	178
2.2.3	24	95	262	507	19,459	284	888	192	1,632	110	12	179
2.1	24	92	257	487	18,331	269	838	180	1,548	105	12	174
1.6	24	89	256	482	17,756	264	804	176	1,501	102	12	172

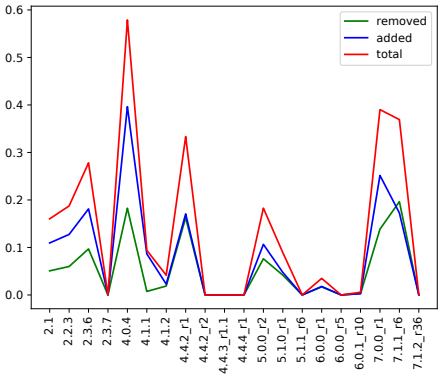

Figure 7: The evolution of manifest documentations

resulted in some of the misconfigurations, which will be introduced in the correctness evaluation of documentation later.

(III) False Positives (FPs) and False Negatives (FNs) Analysis of Extracted Constraints. The accuracy of the extracted constraints directly determines the accuracy of our misconfiguration detection. Therefore, we must first make sure there is no false positive or false negative. If so, we must correct them. To validate the accuracy of our constraint extraction, we chose the most recent documentation and manually constructed the schema by going over all the documentations. In total, there are 190 pages with 849 paragraphs. To generate the ground truth, we have two security researchers each read the documentations, manually extracted the constraints, wrote the manifest schemas; then the two researchers cross-validated their results to converge. It took 20 days for both researchers to read the documentation, pick out manifest-related documentations, understand contexts, construct schema, and validate them.

Then, we compared the manually constructed ground-truth schemas with the automatically generated ones. Among them, we found no false positives but 3 false negatives in *constraint generation* out of 257 (1.17%) total schema constraints generated manually. The reason is that the documentation of `compatible-screen` did

Table 3: Detailed overview of the identified misconfigurations with respect to the number of downloads for Google Play apps, and different system version for pre-installed apps. Note that Cap. represents Capitalization error.

Google Play apps															
Total installs	Misplaced element		Missing element		Misspelled element			Misplaced attribute		Missing attribute		Misspelled attribute			
	# Apps	# Misplaced	# Apps	# Missing	# Apps	# Cap.	# Typo	# Apps	# Misplaced	# Apps	# Missing	# Apps	# Prefix	# Cap.	# Typo
1B+	8	11	0	0	0	0	0	14	166	0	0	15	128	0	0
100M-1B	76	116	1	1	1	1	0	114	297	0	0	131	531	0	0
10M-100M	595	709	8	9	4	1	3	1,057	2,154	8	10	940	2,441	0	0
1M-10M	3,323	4,617	37	97	123	4	121	6,226	10,350	47	53	3,098	6,791	0	0
100k-1M	11,156	13,759	139	311	635	5	632	18,740	28,198	106	115	5,400	11,569	0	0
10k-100k	27,070	32,837	452	740	1,154	7	1,148	41,973	62,823	144	147	7,865	14,754	0	0
1k-10k	50,937	60,110	744	1,102	2,193	12	2,181	76,104	119,184	242	246	11,303	26,234	0	0
100-1k	65,252	72,285	854	1,124	1,553	4	1,558	107,344	154,858	339	343	15,992	55,119	0	0
0-10	69,482	76,645	422	516	251	5	246	127,018	173,644	562	565	16,904	47,486	1	4
total	227,899	261,089	2,657	3,900	5,914	39	5,889	378,590	551,674	1,448	1,479	61,648	165,053	1	4
Pre-installed apps															
Firmware ver.	Misplaced element		Missing element		Misspelled element			Misplaced attribute		Missing attribute		Misspelled attribute			
	# Apps	# Misplaced	# Apps	# Missing	# Apps	# Cap.	# Typo	# Apps	# Misplaced	# Apps	# Missing	# Apps	# Prefix	# Cap.	# Typo
9	0	0	701	769	0	0	0	785	2,311	0	0	1,471	4,169	0	0
8	3	3	7,360	38,153	9	9	0	9,349	43,049	0	0	15,719	45,033	0	0
7	82	82	16,626	98,104	0	0	0	15,914	61,749	0	0	5,500	13,562	0	0
6	634	634	49,056	795,310	0	0	0	14,882	62,440	0	0	0	0	0	0
5	87	87	58,844	771,054	0	0	0	18,510	71,356	0	0	2	2	0	0
4	8	8	18,973	27,013	0	0	0	20,411	58,930	0	0	22,154	1,158	21,549	0
3	0	0	72	72	0	0	0	131	335	0	0	12	30	0	0
2	0	0	153	153	0	0	0	386	670	0	0	3	3	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
total	814	814	151,785	1,730,628	9	9	0	80,368	300,840	0	0	44,861	63,957	21,549	0

not follow the documentation structure. In particular, it did not specify its child elements in the '`can contain`' section as other documentations but in the '`child elements`' section which is a new section that does not exist in other documentations. As a result, ManiScope failed to determine that (1) `screen` is a child of `compatible-screen`, and (2) `android:screenSize` and `android:screenDensity` is valid attribute in `screen`, resulting in 3 false negatives. As this is due to the inconsistent structure of the documentation and easy to fix, we manually added these elements and attributes to the generated schema of all versions, and then the generated schema is used to perform the large scale analysis on Android apps as presented next.

5.2 Manifest Misconfiguration Detection

(I) Detection Result. With the XML schemas reconstructed by ManiScope, we then use them to detect the misconfigurations in most recent apps in Google Play and preinstalled apps in historical firmware, whose overall results are presented in Table 3. Note that the subtotal of apps may not always be equal to the subtotal of misconfigurations as a single app may contain multiple misconfigurations. For Google Play apps, we identified 812,763 misplaced configurations, 5,379 missing configurations, and 165,093 misspelled configurations. For pre-installed apps, we found 301,654 misplaced configurations, 1,730,628 missing configurations, and 85,515 misspelled configurations. We found that manifest misconfigurations are quite prevalent in real-world apps where more than 30% of these apps have at least one misconfiguration.

Misplaced Configuration. Most of the misconfigurations among manifest files are misplaced configurations, and ManiScope identified 261,089 misplaced elements and 551,674 misplaced attributes among the 1.8 million Google Play apps, and 814 misplaced elements and 300,840 misplaced attributes among the 0.6 million pre-installed apps, as shown in Table 3. We also found that most of the misplaced attributes were related to feature requirements (e.g., `android:hardwareAcclerated`, `android:required`), and most of the misplaced elements were frequently used manifest elements (e.g., `<meta-data>`, `<category>`), and elements used to configure access permissions (e.g., `<permission>`, `<uses-permission>`). Additionally, we observed misconfigurations in extremely popular apps related to icons and themes (e.g., the YouTube app contained a misplaced `android:theme` attribute), although they are likely to be of no security concern.

Missing Configuration. For Google Play apps, missing configurations occur in both elements and attribute. For missing elements, all the 3,900 misconfigurations are related to `<action>` element in `<intent-filter>` element. The missing attributes, on the other hand, mainly involved in component name attributes (e.g., `android:name`) and compatibility attributes (e.g., `android:minSdkVersion`). One possible explanation is that the compiler already examines some critical missing problems and aborts compilation if these misconfigurations exist. However, missing configurations are still concerning since they can result in unavailability of app components and create compatibility issues. For example, if the `android:minSdkVersion` attribute in the `<uses-sdk>` element is missing, the system regards the app as compatible with all Android versions, which can cause the app to crash.

For pre-installed apps, although ManiScope did not find any missing attributes, we still identified a large amount of missing `<action>` (1,673,727 of 1,730,628) and `<application>` (56,901 of 1,730,628). This could be explained by the difference between pre-installed apps and Google Play apps. For instance, compared with Google Play apps that rely on Intents to perform functionality, most of the pre-installed apps do not need to specify actions for intent-filter, and therefore many `<action>` elements are not present in `<intent-filter>` element.

Misspelled Configuration. ManiScope detected a large number of misspelled elements and attributes. Among them, we found that there are many more typos than capitalization errors (such as

Table 4: CVSS 3.1 scores of security-related misconfigurations. AV: Attack Vector, AC: Access Complexibility, C: Confidentiality Impact, I: Integrity Impact, A: Availability Impact, G: Google play app, P: pre-installed app. ○: None, ◐: Low, ●: High.

Type	Category	Name	AV	AC	C	I	A	Score*	Severity	# G	# P	Sample Impact
Element	Permission	permission	Local	Low	●	●	○	7.7	High	2,722	0	Component hijacking
		uses-permission	Local	Low	○	○	●	6.2	Medium	1,037	0	App crashing
	Compatibility	minSdkVersion	Local	Low	○	◐	●	6.8	Medium	2,156	408	Data leakage
		required	Local	Low	○	○	●	6.2	Medium	21,855	0	App crashing
Attribute	Functionality	allowBackup	Physical	Low	●	●	●	6.8	Medium	7,432	25,999	Data leakage
		enabled	Local	Low	○	○	◐	4	Medium	1,114	2	Data leakage
		excludeFromRecents	Local	High	◐	○	○	2.9	Low	2,395	12,013	Replay attack
		exported	Local	Low	◐	◐	◐	5.9	Medium	2,120	1,734	Component hijacking
		largeHeap	Local	Low	○	○	◐	4	Medium	7,086	3,950	App crashing
		multiprocess	Local	Low	◐	◐	◐	5.9	Medium	15,511	0	App crashing
		persistent	Local	Low	○	○	◐	4	Medium	16,429	2,391	App crashing
		priority	Local	High	○	○	◐	2.9	Low	2,477	6,907	Component hijacking
		taskAffinity	Local	Low	◐	◐	◐	5.9	Medium	555	5,291	Component hijacking
	Permission	permission	Local	Low	●	●	○	7.7	High	10,348	36	Component hijacking
		protectionLevel	Local	Low	●	●	○	7.7	High	6,839	6,787	Component hijacking

*: For all entries, Privileges Required (PR) is None, User Interaction (UI) is None, and Scope (S) is Unchanged. †: The total downloads of all apps in this category.

`Service` v.s. `service`) in misspelled elements. Also, most of the capitalization errors of elements (30 of 39) have the first character capitalized (e.g., `Activity`). All the 9 capitlization errors in pre-installed apps are the first-character-caplitalization problem of `<service>` (i.e., `Service`). For typos of elements, most are due to spelling errors (e.g., `mata-data` v.s. `meta-data`, which accounts for 5,585 misconfiguration among the 6,446 misconfigurations). Another source of typos comes from a missing hyphen (e.g., `intentfilter` v.s. `intent-filter`), and incorrect usage of plural/singular form (e.g., `support-screen` v.s. `support-screens`). For pre-installed apps, 472 out of 486 misconfigurations are typo from `intent-filter` to `intent-flter`, whereas the rest 14 are plural problems, i.e., `permission` spelled into `permissions`. For the top misspelled attributes, we found that missing prefixes are most prevalent (e.g., `exported` v.s. `android:exported`).

(II) FP and FN Analysis of the Detected Misconfigurations. To confirm whether there are any FPs and FNs in the identified misconfigurations, we manually checked random samples of 500 misconfigurations identified by ManiScope from pre-installed and Google Play apps, respectively. Among these 1,000 misconfigurations, we identified zero FNs but 27 FPs (2.70%). For the false positives, we found that the root cause is due to the typos in the official documentations, which involve two attributes: (1) 5 out of 27 FPs involve `android:allowBackup`, which was misspelled into `android:allowbackup` from 4.4.2 to 4.4.4. As such, ManiScope may only regard `android:allowbackup` as correct name for a certain version. If a manifest file contains `android:allowBackup` in `application` element, which is actually correct, ManiScope identifies it as misspelled instead, resulting in a FP. (2) 22 out of 27 FPs involve `resizeableActivity`, where the `android:` prefix is missing from the documentation. As such, ManiScope will identify the correct attribute with prefix as misspelled, resulting in a FP. Interestingly, although the typo of `allowBackup` is fixed after 4.4.4 (but still causing FPs when ManiScope analyzes apps for these versions), the typo of `resizeableActivity` remained until our responsible disclosure as in July 2021.

The reason why we have zero FN is two-fold. First, identification of positional constraint will not yield FN because we have enforced an allow-list mechanism to detect misplaced manifest entities. As such, the positional constraint will be even stricter than the documentation if we fail to extract any positional constraints. As manifest files containing misplaced element will for sure be inconsistent with the documentation, it will for sure be identified as misplaced by ManiScope. Second, although the quantitative constraint extraction which involves NLP may have FN if we fail to extract some quantitative constraints (thus making the constraint less strict than documentation), we have manually validated with the documentations and found no such a problem.

5.3 Security-Related Misconfigurations

(I) Severity and impacts of the misconfiguration. To determine the security impact of these misconfigurations, we manually checked all of the elements and attributes associated with the misconfigurations to understand their potential security impact. Among them, we identified 2 elements and 13 attributes that could have an impact on security. To rate the security severity of the identified misconfigurations, we categorized them based on their expected severity according to the CVSS (Common Vulnerability Scoring System) 3.1 [10] scoring metric. This metric is widely used in industry and academia to provide an assigned Common Vulnerabilities and Exposures (CVE) with a severity score. A CVSS score includes six metrics that can be scored with values of high, medium, and low security impact: the attack vector (same network, adjacent network, local, or physical access), access complexity (whether an attacker can expect repeatable success or needs to create certain conditions), confidentiality impact (whether all the exported components are divulged to the attacker), integrity impact (whether the attacker can manipulate the file and data freely), and availability impact (whether it causes a denial of service, or heavy performance losses). The CVSS scores for these 15 misconfigurations are presented in the Score-column of Table 4.

According to the CVSS system, among these 15 misconfigurations that could cause security concerns, 3 of them have high severity, 10 have medium severity, and 2 have low severity. These misconfigurations can result in various security issues, including component hijacking, data leakage, and app crashing. For instance, we can see that apps with a misplaced `android:permission` attribute are associated with most installs, which may cause purchasing replay attacks. In addition, some misconfigurations (e.g., the data leakage and component hijacking caused by the `android:allowBackup`

Table 5: Distributions of security-related misconfiguration in Google Play and Preinstalled apps.

# Google Play apps																		
			Element		Attribute													
App Category	# Apps	# Misconf.	Permission	Uses-permission	MinSdkVersion	Required	AllowBackup	Enabled	ExcludeFromRecents	Exported	LargeHeap	Multiprocess	Persistent	Priority	TaskAffinity	Permission	ProtectionLevel
ART & DESIGN	21,303	1,076	6	2	4	52	18	2	19	7	73	146	876	6	0	13	11
AUTO & VEHICLES	11,971	419	9	5	2	129	13	5	15	11	27	82	103	19	0	63	31
BEAUTY	10,259	353	19	0	19	204	4	0	2	1	15	71	70	6	1	5	13
BOOKS & REFERENCE	98,634	2,216	67	28	99	925	29	15	13	32	415	365	377	23	2	268	272
BUSINESS	104,354	5,667	51	33	96	2,585	114	128	67	237	473	181	367	99	15	167	1,431
COMICS	2,758	120	0	2	10	11	48	0	0	1	1	18	18	1	0	19	2
COMMUNICATION	36,602	2,310	39	17	27	446	28	40	101	69	105	1,071	1,057	103	53	68	154
DATING	3,405	294	3	7	4	28	7	1	1	2	4	210	217	10	0	1	6
EDUCATION	161,908	5,626	183	47	333	1,713	243	402	29	476	547	917	964	82	1	516	618
ENTERTAINMENT	139,840	5,254	408	38	66	1,347	288	42	58	104	439	1,564	1,422	129	17	390	312
EVENTS	9,544	751	1	5	6	367	22	17	2	10	25	31	32	5	0	1	273
FINANCE	43,315	2,856	49	16	49	1,222	64	22	22	107	590	141	155	62	3	41	776
FOOD & DRINK	40,630	958	34	5	25	376	78	17	12	28	45	131	150	32	1	102	68
GAME	296,079	16,706	74	602	404	866	5,370	102	91	84	406	1,224	1,204	426	362	6,580	167
HEALTH & FITNESS	57,481	1,671	26	8	53	607	90	23	25	46	143	303	339	117	1	110	149
HOUSE & HOME	9,246	326	4	1	3	146	4	2	2	14	29	37	55	17	0	15	40
LIBRARIES & DEMO	4,866	119	2	1	7	42	5	3	2	6	25	7	20	2	0	3	2
LIFESTYLE	113,470	4,737	74	16	57	2,403	170	27	50	89	668	387	514	111	2	285	273
MAPS & NAVIGATION	19,082	819	5	4	12	385	12	3	20	36	41	43	102	27	3	20	214
MEDICAL	20,998	802	7	45	33	328	17	9	10	22	65	22	32	28	1	85	146
MUSIC & AUDIO	130,235	5,064	1,292	13	21	422	285	28	20	55	83	2,420	2,473	150	6	103	78
NEWS & MAGAZINES	40,664	2,387	65	7	24	319	13	14	351	70	203	593	632	31	1	657	50
PARENTING	2,531	114	3	1	7	47	5	3	2	6	8	10	19	13	0	4	8
PERSONALIZATION	91,609	5,060	7	2	15	170	45	17	585	54	171	2,703	1,665	192	22	243	51
PHOTOGRAPHY	51,093	2,535	5	6	9	330	20	7	18	17	1,425	657	730	41	1	40	71
PRODUCTIVITY	53,150	2,251	43	17	71	1,051	77	43	52	93	224	97	161	101	22	88	317
SHOPPING	38,943	1,904	15	10	504	670	48	14	16	58	111	172	219	43	1	26	201
SOCIAL	33,288	1,886	56	43	24	608	41	21	13	48	198	581	646	69	4	24	100
SPORTS	30,937	1,237	30	9	18	356	46	12	40	74	42	338	374	41	2	128	84
TOOLS	103,933	4,561	108	30	116	1,914	155	73	269	191	197	544	701	397	32	122	390
TRAVEL	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
TRAVEL & LOCAL	53,138	3,124	20	13	26	1,651	58	14	389	42	239	113	355	46	1	105	466
VIDEO PLAYERS	12,873	577	11	2	7	77	7	5	20	24	42	193	248	37	1	46	58
WEATHER	5,718	337	6	2	5	58	8	3	79	6	7	139	132	11	0	10	7
Total	1,853,862	84,117	2,722	1,037	2,156	21,855	7,432	1,114	2,395	2,120	7,086	15,511	16,429	2,477	555	10,348	6,839

# Pre-installed apps																	
				Element		Attribute											
Android Version	# Firm.	# Apps	# Misconf.														
9	41	10,559	390	0	0	6	0	32	0	112	67	0	0	2	103	97	0	37
8	331	69,515	4,129	0	0	92	0	366	0	1,734	296	391	0	33	689	1,023	1	501
7	523	133,643	7,313	0	0	253	0	707	0	2,182	358	1,025	0	58	1,118	711	0	1,428
6	607	132,537	7,071	0	0	57	0	960	0	1,906	140	1,071	0	125	1,072	1,019	1	1,616
5	1052	177,512	8,580	0	0	0	0	546	0	2,497	265	773	0	843	1,137	1,636	1	2,179
4	1974	163,933	31,420	0	0	0	0	23,388	2	3,527	608	643	0	1,330	2,742	802	33	1,026
3	10	1,201	54	0	0	0	0	0	0	0	0	42	0	0	10	2	0	0
2	42	3,206	96	0	0	0	0	0	0	55	0	5	0	0	36	1	0	0
Total	4,580	692,106	59,053	0	0	408	0	25,999	2	12,013	1,734	3,950	0	2,391	6,907	5,291	36	6,787

Total Downloads: 0–1K: 1K–1M: 1M–1B: 1B+:

and `android:exported` attributes) may also affect both thousands of Google Play apps and pre-installed apps. However, compared to Google Play apps, the pre-installed apps make less mistakes, and these pre-installed apps contain significantly less misconfigurations in elements and several attributes (e.g., `permission`). This might be explained by the limited but essential functionalities of pre-installed apps that make developers avoid using some manifest entities.

(II) Affected apps with security-related misconfiguration. To further understand the effects of these misconfigurations, we grouped the Google Play apps based on their categories and the pre-installed apps on the firmware versions, as shown in Table 5, where the cell color denotes the scale of total install numbers of affected apps. We notice that most of the misconfigured apps are in the game category, which may be explained by additional system resources required by games to avoid decreased performance or process termination. For pre-installed apps, the misconfigured apps also grow as total amount of apps grow: most of the pre-installed apps are in version 4 to 8, and the problem still exists in recent devices after version 7.

6 SECURITY CASE STUDIES

6.1 Component Hijacking

There are several attributes in the manifest file to protect a component from unauthorized access (i.e., component hijacking). However, with misconfigurations of those attributes, the component would have been exposed to attackers. Through a victim's component, a malicious app can perform illicit actions such as component hijacking, assume there is a malicious app in the victim's phone and this app will attack the app with misconfigured attributes.

Misplaced `android:permission` attribute. This attribute specifies the permissions required by other apps for component communication, in order to defend against unauthorized access from

Table 6: Top five categories of apps affected by security-related misconfigurations of different types.

Table A. Permission

	App category	# App
Payment	Game	6,406
	News	621
	Education	488
	Books	255
	Personalization	237
Cloud Msg	Lifestyle	104
	Sports	61
	Entertainment	55
	Tools	55
	Books	47
SMS Msg	Tools	11
	Productivity	10
	Communication	6
	Social	3
	Lifestyle	1

Table B. Allow Backup

App category	# App
Game	5,368
Entertainment	288
Music	285
Education	238
Lifestyle	168
Tools	155
Business	99
Health	86
Food	78
Productivity	72
Travel	56
Finance	53
Comics	48
Sports	46
Shopping	45

Table C. Protected Broadcast

	App category	# App
Media	Photography	2,317
	Entertainment	299
	Video	227
	Tools	212
	Music	171
Headset	Finance	12
	Communication	3
	Productivity	2
	Auto	2
	Personalization	1
USB	Personalization	3
	Communication	2
	Photography	2
	Books	1
	Lifestyle	1

Total downloads: 0–1K: 1K–1M: 1M–1B: 1B+:

apps that do not have these permissions. Misplaced permissions will allowing arbitrary apps to interact with them, thereby making the apps vulnerable to component hijacking attacks. As presented in Table 6(A), among the 10,348 apps that contained misplaced permission attributes, 9,627 of them were related to payment as shown in Figure 1, and all of these affected payment components are associated with the Amazon Appstore with the majority of these apps being games (6,561/9,627). We were surprised to find that this flaw primarily stemmed from an incorrect code snippet provided by Amazon official support team [15] for the Amazon in-app purchasing SDK. Technically, this permission is used to protect the app from fraudulent attempts to replay transactions. Ironically, such protection is voided by the erroneous code snippet, leaving thousands of apps vulnerable to fraudulent attacks: with the permission enforcement ineffective, apps can be exploited by purchasing an in-app item and capturing the transaction receipt sent from Amazon Appstore to the app, then replaying that same receipt to the corresponding receiver at will to repurchase more units without paying. This vulnerability impacted very popular apps, some of which with more than 100 million installs. We responsibly disclosed this vulnerability to impacted app developers and Amazon, and it has been confirmed right after our disclosure.

6.2 Data Leakage

On Android, private app data can be copied out of a device using the `adb backup` command if an app has its `android:allowBackup` attribute set to true. In this case study, we present a data leakage caused by a misplaced `android:allowBackup` attribute.

Misplaced `android:allowBackup` attribute. This attribute should be set in the `<application>` element to specify whether or not the app allows its data to be backed up and restored. When developers set this attribute to `false`, their intentions are likely to keep sensitive user data protected by preventing this data from being extracted from the device. However, if developers configure it to be `false` and misplace it, it will lead to data leakage attacks (i.e., perform backup through `adb`) since the default value of this attribute is `true`. In total, we have identified 7,432 Google Play apps that have such misconfiguration, as shown in Table 6(B). A concrete example we have investigated is a game named `superOscar` (with over 10 million downloads) where the `android:allowBackup="false"` is placed inside the `<manifest>` element, allowing attackers with physical access to obtain the login credentials through the backup process.

6.3 Channel Hijacking

Interestingly, on top of the misconfigurations of manifest elements or attributes that are in the documentation, we also detected a wide usage of elements and attributes that are not on the documentation, appearing as unexpected elements/attributes but not identified as typos. This is caused by a set of undocumented manifest entities for applications from Android or OEM producers carrying system signature only, which are designed for testing or privilege-protected configuration. Unfortunately, there are still third-party developers that attempt to use these elements or attributes for configuration, which will eventually be ignored by Android. For instance, among the undocumented elements, we found a particular element called `protected-broadcast` which appeared in 4,098 apps in total. Due to space limits, we only present the top five categories of each types of componentss as presented in Table 6(C), which contains 3,261 apps in total. This element is only usable by pre-installed privileged system apps and the Android framework, allowing them to restrict certain broadcast actions to be sent only by the system. When this element is configured in third-party apps, the Android PackageParser will silently ignore the element and no protection will be granted. This can create a severe vulnerability since any app on the device can send these messages and the receiving app will treat them as though they have been sent by the system.

7 DISCUSSION

7.1 Root Causes and Mitigation

One plausible cause of misconfigurations is developer's carelessness. Ideally, instead of allowing developers to manually configure the manifests, additional tools should be provided to automate these configurations to reduce potential errors. Second, as evidenced in §5, the official documentation pages provided by Google contain mistakes (e.g., typos, or missing attributes) which lead to misconfigurations in the manifest files. Finally, similar to many other bugs, code reuse is another root cause. For instance, the Amazon app defrauding case caused by the manifest misconfiguration of the component exposed 9,474 apps to defrauding due to a single misplaced attribute in the official guide on Amazon website that was copied by developers when integrating amazon in-app purchases.

Explicit warnings during validation. The Android operating system currently only triggers error logs on essential problems in app manifest files, and these error logs cannot be easily viewed by the users. Android system could proactively display the warnings to developers and end-users, to help them identify and fix any issues.

Correct and clear documentations. IDE and SDK providers such as Google and Amazon, should provide clear documentation to facilitate developer comprehension for manifest configuration. They also need to ensure that the code snippets provided in their documentation and online resources are correct. Otherwise, defects in the manifest snippets could be propagated to a large number of apps. In addition, they should provide systematic, rigorous validation tools for developers to proactively detect and fix misconfigurations.

Ensuring manifest file correctness. For app developers, they have to ensure that they understand the configurations correctly, and then leverage automated tools to reduce errors. Meanwhile, they have to be careful copying snippets online as they may contain mistakes that eventually impair the security of their applications.

7.2 Limitations and Future Work

Covering undocumented elements and attributes. Although ManiScope identified all the manifest elements and attributes defined in the official documentation, there may be other elements and attributes defined elsewhere. For instance, developers might define their own attributes and elements. Also, there might be some attributes and elements exclusively for pre-installed apps. Future work could automate the element and attribute extraction from other sources in addition to the official documentation.

Providing more comprehensive case studies. In this paper, we only discussed security-related cases from three categories of misconfgiurations. An immediate future work could be performing more comprehensive case studies to measure and identify the potential attacks to raise the attention from community and fix the problems to prevent from exploitation.

7.3 Responsible Disclosure

We have disclosed our findings to Amazon about the issues in apps that use its in-app purchasing SDK, and the incorrect snippets in its documentation and online forum. We have also disclosed all issues involving `android:allowBackup` attribute and `<protected-broadcast>` element to developers of impacted apps. Our disclosure of the misconfigurations have been confirmed by various developers, and we were informed that they have fixed or will fix the issue in the future. We had also informed Google about typos in documentations, and the issue was then fixed on July 13th, 2021.

8 RELATED WORK

Extracting information of interest using NLP techniques. As a powerful technique, NLP has been widely used to extract information of interest from free-form texts. For example, to extract constraints from technical documents, Kof et al. and Sadoun et al. [36, 42] combined lexical, syntactical, and semantic analysis. Korner et al. [37] integrated part-of-speech tagger, statistic parser, and named entity recognizer to extract the information after splitting the text into chunks, and then validated them with common sense. NLP has also been used to solve various security issues, such as detecting policy declaration and contradictions (e.g., [22]), bug finding (e.g., [27]), and cybercrime (e.g., [29, 38, 39, 41, 46]). All of these efforts also need to solve the ambiguity problem. Various approaches have been proposed, by adopting data mining [44], developing deep learning models [32], or using crowd-sourcing approaches to manually identify ontologies [43, 47]. We enrich the state-of-the-art with NLP techniques to extract XSD from documentations.

Android security analysis. Numerous prior efforts on Android security have mainly focused on investigating and identifying security threats in Android apps including requesting excessive permissions, component hijacking, and insecure driver implementations. For instance, for analyzing permission issues in Android systems, PSCout [23] adopted code analysis to trace the path of API calls and permission checks, produced a specification of API permission requirements, while Backes et al. [24] performed analysis of Android permission model across different Android versions. Approaches to derive precise protection by converting CFG [24] to Access-control flow graph determining necessary protections have also been proposed [24]. Additionally, there have been efforts that looked into insecure components and driver implementations [30, 31, 34, 40, 50–52]. Compared to these efforts, we systematically investigate novel security issues caused by manifest misconfigurations.

Detecting misconfigurations. On detecting misconfigurations of Android manifest files, Jha et al [33] identified configuration errors in about 13, 000 Android apps using manually constructed constraints. The study, however, relies on predefined rules gathered by manually reading the documentations, and therefore cannot be adopted to generate schema for various versions of documentations for pre-installed app validation. Additionally, the manual approach did not provide a comprehensive coverage of manifest configurations, quantitative constraints, nor potential security issues. To identify potential policy misconfigurations in access control systems, Bauer et al. [25] applied association rule mining on previously observed accesses to extract statistical patterns (i.e., rules), and then used the rules to detect misconfigurations. Das et al. [28] proposed to detect inconsistencies of access control updates by correlating access control between group memberships and using statistical techniques to find differences between users. Yuan et al. [49] discovered user-defined policy violations and inconsistencies among firewall rules. There are a number of other blackbox [35, 53, 54] and whitebox [48] approaches to detect misconfigurations. To the best of our knowledge, none of the existing efforts have been used to analyze misconfigurations in Android app manifests.

9 CONCLUSION

We have presented ManiScope, a tool to automatically construct Android app manifest schema from the official documentation and detect misconfigurations in app manifest files. ManiScope employs novel domain-aware NLP parsing and pruning techniques that allow it to accurately capture positional and quantitative constraints on manifest elements and attributes. We have tested ManiScope with 1,853,862 Google Play apps and 692,106 preinstalled apps, with which ManiScope identified 609,428 misconfigured Google Play apps and 246,658 misconfigured preinstalled apps, respectively. We provided an in-depth analysis and measurement of the security threats posed by these misconfigurations, together with case studies to show their potential impacts.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedbacks. The team at The Ohio State University was partially supported by NSF awards 1834215 and 2112471. Any opinions, findings, conclusions, or recommendations are those of the authors and not necessarily those of the NSF.

REFERENCES

[1] 2017. CVE-2017-16835. https://nvd.nist.gov/vuln/detail/CVE-2017-16835. (Accessed on 2021-01-18).

[2] 2017. CVE-2017-17551. https://nvd.nist.gov/vuln/detail/CVE-2017-17551. (Accessed on 2021-01-18).

[3] 2021. Android manifest development documents. https://developer.android.com/guide/topics/manifest/manifest-intro. (Accessed on 2021-01-18).

[4] 2021. Android Open Source Project. https://cs.android.com/android/platform/superproject.

[5] 2021. Android Package Parser. http://androidxref.com/9.0.0_r3/xref/frameworks/base/core/java/android/content/pm/PackageParser.java#parseVerifier. (Accessed on 2021-01-12).

[6] 2021. Android Studio linter. https://developer.android.com/studio/write/lint. (Accessed on 2021-01-18).

[7] 2021. Apache Xerces. https://en.wikipedia.org/wiki/Apache_Xerces. (Accessed on 2021-01-18).

[8] 2021. The attributes used in AndroidManifest.xml. https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/res/res/values/attrs_manifest.xml. (Accessed on 2021-01-18).

[9] 2021. BeautifulSoup Parser. https://lxml.de/elementsoup.html. (Accessed on 2021-01-18).

[10] 2021. CVSS v3 Calculator. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator. (Accessed on 2021-01-18).

[11] 2021. Filters on Google Play. https://developer.android.com/google/play/filters. (Accessed on 2021-01-18).

[12] 2021. Introduction to DTD. https://www.w3schools.com/xml/xml_dtd_intro.asp. (Accessed on 2021-01-18).

[13] 2021. An Introduction to Schematron. https://www.xml.com/pub/a/2003/11/12/schematron.html.

[14] 2021. lxml - XML and HTML with Python. https://lxml.de/. (Accessed on 2021-01-18).

[15] 2021. Purchasing Listener doesn't get called. https://forums.developer.amazon.com/questions/16519/purchasinglistener-doesnt-get-called.html. (Accessed on 2021-01-18).

[16] 2021. Python AxmlParser. https://github.com/antitree/AxmlParserPY. (Accessed on 2021-01-18).

[17] 2021. Relax NG home page. https://relaxng.org/. (Accessed on 2021-01-18).

[18] 2021. SamMobile - Your authority on all things Samsung. https://www.sammobile.com/. (Accessed on 2021-05-30).

[19] 2021. Schema - W3C. https://www.w3.org/standards/xml/schema.

[20] 2021. View & restrict your app's compatible devices | Play Console Help. https://support.google.com/googleplay/android-developer/answer/7353455?hl=en. (Accessed on 2021-01-18).

[21] 2021. XML Schema Languages. https://en.wikipedia.org/wiki/XML_schema#Languages. (Accessed on 2021-01-18).

[22] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. 2019. PolicyLint: Investigating Internal Privacy Policy Contradictions on Google Play. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*.

[23] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *the ACM Conference on Computer and Communications Security, CCS'12, 2012*. ACM.

[24] Michael Backes, Sven Bugiel, Erik Derr, Patrick D. McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*.

[25] Lujo Bauer, Scott Garriss, and Michael K. Reiter. 2011. Detecting and resolving policy misconfigurations in access-control systems. *ACM Trans. Inf. Syst. Secur.* 14, 1 (2011), 2:1–2:28.

[26] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media.

[27] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. 2019. Devils in the Guidance: Predicting Logic Vulnerabilities in Payment Syndication Services through Automated Documentation Analysis. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association.

[28] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. 2010. Baaz: A System for Detecting Access Control Misconfigurations. In *19th USENIX Security Symposium, Proceedings*. 161–176.

[29] Greg Durrett, Jonathan K Kummerfeld, Taylor Berg-Kirkpatrick, Rebecca S Portnoff, Sadia Afroz, Damon McCoy, Kirill Levchenko, and Vern Paxson. 2017. Identifying products in online cybercrime marketplaces: A dataset for fine-grained domain adaptation. *arXiv preprint arXiv:1708.09609* (2017).

[30] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic Uncovering of Privilege-Escalation Vulnerabilities in Pre-Installed Apps in Android Firmware. In *29th USENIX Security Symposium (USENIX Security 20)*. 2379–2396.

[31] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. [n.d.]. An Analysis of Pre-installed Android Software. In *2020 IEEE Symposium on Security and Privacy*.

[32] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G. Shin, and Karl Aberer. 2018. Polisis: Automated Analysis and Presentation of Privacy Policies Using Deep Learning. In *27th USENIX Security Symposium, USENIX Security 2018*.

[33] Ajay Kumar Jha, Sunghee Lee, and Woo Jin Lee. 2017. Developer mistakes in writing Android manifests: an empirical study of configuration errors. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017*. IEEE Computer Society, 25–36.

[34] Ryan Johnson, Mohamed Elsabagh, Angelos Stavrou, and Jeff Offutt. 2018. Dazed Droids: A Longitudinal Study of Android Inter-App Vulnerabilities. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM.

[35] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008, June 24-27, 2008, Anchorage, Alaska, USA, Proceedings*. IEEE Computer Society.

[36] Leonid Kof. 2005. Natural Language Processing: Mature Enough for Requirements Documents Analysis?. In *10th International Conference on Applications of Natural Language to Information Systems, NLDB 2005, Proceedings*. Springer.

[37] Sven J. Körner and Mathias Landhäußer. 2010. Semantic Enriching of Natural Language Texts with Automatic Thematic Role Annotation. In *15th International Conference on Applications of Natural Language to Information Systems, NLDB 2010, Proceedings*. Springer.

[38] Yeonjoon Lee, Xueqiang Wang, Kwangwuk Lee, Xiaojing Liao, XiaoFeng Wang, Tongxin Li, and Xianghang Mi. 2019. Understanding iOS-based Crowdturfing Through Hidden {UI} Analysis. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 765–781.

[39] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, Zhongyu Pei, Hao Yang, Jianjun Chen, Haixin Duan, Kun Du, Eihal Alowaisheq, Sumayah Alrwais, et al. 2016. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 707–723.

[40] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM.

[41] Rebecca S Portnoff, Sadia Afroz, Greg Durrett, Jonathan K Kummerfeld, Taylor Berg-Kirkpatrick, Damon McCoy, Kirill Levchenko, and Vern Paxson. 2017. Tools for automated analysis of cybercriminal markets. In *Proceedings of the 26th International Conference on World Wide Web*. 657–666.

[42] Driss Sadoun, Catherine Dubois, Yacine Ghamri-Doudane, and Brigitte Grau. 2013. From Natural Language Requirements to Formal Specification Using an Ontology. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*. IEEE Computer Society.

[43] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in Android application code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM.

[44] John W. Stamey and Ryan A. Rossi. 2009. Automatically identifying relations in privacy policies. In *Proceedings of the 27th Annual International Conference on Design of Communication, SIGDOC 2009*. ACM.

[45] Henry S Thompson, Noah Mendelsohn, D Beech, and M Maloney. 2009. W3C XML schema definition language (XSD) 1.1 part 1: Structures. *The World Wide Web Consortium (W3C), W3C Working Draft Dec* 3 (2009).

[46] Peng Wang, Xianghang Mi, Xiaojing Liao, XiaoFeng Wang, Kan Yuan, Feng Qian, and Raheem A Beyah. 2018. Game of Missuggestions: Semantic Analysis of Search-Autocomplete Manipulations.. In *NDSS*.

[47] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D. Breaux, and Jianwei Niu. 2018. GUILeak: tracing privacy policy claims on user input data for Android applications. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*. ACM.

[48] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *24th Symposium on Operating Systems Principles, SOSP*. ACM.

[49] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. 2006. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 199–213.

[50] Lei Zhang, Zhemin Yang, Yuyu He, Zhenyu Zhang, Zhiyun Qian, Geng Hong, Yuan Zhang, and Min Yang. 2018. Invetter: Locating Insecure Input Validations in Android Services. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. ACM.

[51] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic Uncovering of Hidden Behaviors From Input

Validation in Mobile Apps. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*. San Francisco, CA.

[52] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. 2014. The peril of fragmentation: Security hazards in Android device driver customizations. In *2014 IEEE Symposium on Security and Privacy*. IEEE.

[53] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *2019 IEEE Symposium on Security and Privacy, SP 2019*. IEEE.

[54] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AuthScope: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*. Dallas, TX.