Automatic Uncovering of Hidden Behaviors From Input Validation in Mobile Apps

Qingchuan Zhao*, Chaoshun Zuo*, Brendan Dolan-Gavitt†, Giancarlo Pellegrino‡, Zhiqiang Lin*

*The Ohio State University, †New York University, ‡CISPA Helmholtz Center for Information Security

***Abstract*—Mobile applications (apps) have exploded in popularity, with billions of smartphone users using millions of apps available through markets such as the Google Play Store or the Apple App Store. While these apps have rich and useful functionality that is publicly exposed to end users, they also contain hidden behaviors that are not disclosed, such as backdoors and blacklists designed to block unwanted content. In this paper, we show that the *input validation behavior*—the way the mobile apps process and respond to data entered by users—can serve as a powerful tool for uncovering such hidden functionality. We therefore have developed a tool, INPUTSCOPE, that automatically detects both the execution context of user input validation and also the content involved in the validation, to automatically expose the secrets of interest. We have tested INPUTSCOPE with over 150,000 mobile apps, including popular apps from major app stores and pre-installed apps shipped with the phone, and found 12,706 mobile apps with backdoor secrets and 4,028 mobile apps containing blacklist secrets.**

I. INTRODUCTION

Mobile applications (apps) now number in the millions and provide useful functionality to billions of users. However, alongside this useful functionality, many apps also include hidden behaviors that are not publicly disclosed to users. These behaviors may range from innocuous Easter eggs, such as custom animations used in Google Hangouts when certain keywords are mentioned, to more pernicious behaviors like backdoors and censorship blacklists.

The harm caused by such behaviors affects both users and developers. Users' security may be compromised if an ostensibly secure app, such as a lock screen app, contains a backdoor that allows anyone who knows the master password to bypass the lock screen. Backdoors may also harm developers when backdoor secrets are exposed, since the hidden functionality can allow users to bypass restrictions built into the app (*e.g.*, a hidden menu protected by a password may enable paid features for free). Finally, censorship blacklists may prevent users from exercising their freedom of expression by banning the discussion of sensitive political topics (although such blacklists may also have benign uses, such as preventing users from choosing offensive usernames).

Nor are such cases hypothetical: by manually examining several mobile apps, we found that a popular remote control app[1] (10 million installs) contains a master password that can unlock access even when locked remotely by the phone owner when device is lost. Meanwhile, we also discovered a popular screen locker app (5 million installs) uses an access key to reset arbitrary users' passwords to unlock the screen and enter the system. In addition, we also found that a live streaming app (5 million installs) contains an access key to enter its administrator interface, through which an attacker can reconfigure the app and unlock additional functionality. Finally, we found a popular translation app (1 million installs) contains a secret key to bypass the payment for advanced services such as removing the advertisements displayed in the app.

Motivated by the above examples, in this paper we tackle the problem of uncovering hidden behaviors in mobile apps. The key insight of our work is the observation that hidden functionality can be uncovered by examining ways user inputs are *validated*. Over the past decades, we have seen several program analysis techniques that can analyze user input validation (*e.g.*, [4], [8], [9], [27], [28], [34], [36]). However, existing approaches are too often specific to the class of input validation vulnerabilities, such as SQL injection (*e.g.*, [16], [24]). Also, these approaches can only determine when a program fails to neutralize dangerous characters and fall short at determining when input validation results in the execution of hidden functions.

In this paper, therefore, we present a new static analysis technique, INPUTSCOPE, to automatically uncover hidden functionality in mobile apps. INPUTSCOPE takes as input an Android mobile app, and then combines static taint analysis with backward slicing to determine when the input app compares data entered by the user against some value stored in the app or retrieved over the network. Then, INPUTSCOPE exposes input-triggered secrets by introducing the novel concept of the *execution context* of user input validation, which combines two orthogonal aspects of the input validation procedure: (i) the types of the data being validated, and (ii) the code dispatch behavior associated with the result of the comparison, such as the number of times the validation is iterated and the number of potential branches following a successful validation. Finally, INPUTSCOPE inspects both the content and execution context with the aid of a set of *security policies* to expose the hidden secrets, *e.g.*, backdoors or blacklist secrets.

We have implemented a prototype of INPUTSCOPE and studied the incidence of user input-triggered hidden secrets in top-installed mobile apps. To that end, we created a dataset of 150,000 apps, including the top 100,000 apps from the Google Play by the number of installations, the top 20,000 apps from an alternative store by the number of installations, and 30,000 pre-installed apps extracted from Samsung smartphones' firmware.

[1] Note that we do not reveal the concrete names of apps whose vulnerabilities remain unpatched at the time of publication.

DOI 10.1109/SP40000.2020.00072

Our evaluation uncovered a concerning situation. We identified 12,706 apps containing a variety of backdoors such as secret access keys, master passwords, and secret commands that can allow users to access admin-only functions or attackers to gain unauthorized access to users' accounts. Also, our analysis discovered 4,028 apps validating user input against blacklisted words of different categories such as insults, racial discrimination, political leader names, and mass incidents.

Contribution. In short, we make the following contributions:

- **Novel Discovery.** We find that input validation in mobile apps can be used to expose input triggered secrets such as backdoors and blacklist secrets, and that input-dependent hidden functionality is widespread in Android apps.
- **Systematic Tool.** We develop a systematic, open source tool[2], INPUTSCOPE, to automatically identify both execution context and validated target content from input validation, which we use to uncover input-triggered secrets in mobile apps.
- **Comprehensive Evaluation.** We have tested our tool with more than 150,000 popular mobile apps and discovered that 8.47% of them contain backdoor secrets such as secret access keys, master passwords, and secret commands, and 2.69% of them contain blacklist secrets such as offensive forbidden words.

II. BACKGROUND AND MOTIVATION

In this section, we present the necessary background to better understand INPUTSCOPE. We begin by describing the types of input received by mobile apps in §II-A. Then, we briefly present how user input is typically validated in a mobile app in §II-B. Finally, we examine three real world apps to motivate the problem we aim to solve in §II-C.

A. Types of Input to Mobile Apps

Similar to the software in non-mobile platforms, the input to a mobile app can be generated from a variety of sources, which can be classified into the following two categories:

Internal Input. An app can directly read the inputs from itself (*e.g.*, for configuration), and we call these inputs internal inputs. There are two types of internal inputs, based on where the input comes from: input coming from the program code (*e.g.*, a hardcoded string) of the app, or input coming from the resource files (*e.g.*, a database) carried within the app.

External Input. In addition to internal input, apps consume input from the external world. Based on where an external input comes from, we can also classify them into two subcategories:

- **External Local Input.** Typically, an app will consume local input such as keystrokes typed by a user, input that originates from system libraries (*e.g.*, a GPS library), or input that is generated by other apps locally and transmitted via an *intent*.
- **External Remote Input.** In addition to local input, an app can also consume input from remote servers or external peripherals (*e.g.*, a bluetooth device). We call these inputs external remote inputs because they are generated by remote parties.

B. How to Validate an Input

Input must be validated prior to being acted upon. Depending on whether the allowed inputs are known by the user, input validation can be performed via either a blacklist or a whitelist:

- **Blacklist.** If an input is compared with a list that contains the blocked content, this list is called a blacklist. In this case, the user typically is not aware of the complete list and the list is often not bounded (it can increase over time); such lists are often kept secret. Anti-virus signatures are an example of a blacklist and viruses should not be aware of the signatures to prevent evasion.
- **Whitelist.** If an input is compared with a list that contains the allowed content, this list is called whitelist. Unlike blacklists, in which the item in the list is a secret, users must know the items in the whitelist (and this list is often bounded with a fixed size), otherwise they will not be able to use the system.

Input validation can be performed at either syntactic level or semantic level (or both), and consequently we can have syntactic validation and semantic validation:

- **Syntactic validation.** Syntactic validation operates on structural properties of data, such as the format or size of the input, with the goal to accept well-formed inputs and disregard malformed ones (*e.g.*, an invalid email address, phone number, or zip code) [1].
- **Semantic validation.** Semantic validation focuses on the *meaning* of the user input, *e.g.*, a social app could check whether an entered date is illegal, such as February 31st [1], and a shopping app could check whether the number of the items in the shopping cart is greater than 0 when checking out.

C. Motivating Examples

Next, we present three real world examples to illustrate how input validation can be used to reveal backdoors and blacklist secrets.

Backdoor Secrets. If an input is used to bypass the access control (*e.g.*, authentication) in an app, this input is a backdoor secret. We have witnessed numerous such backdoor secrets. In the following, we use a popular file encryption app with 500,000+ installs, which is used to hide or lock private files from being accessed by others, to illustrate how its validation process exposes its master password (Figure 1).

In particular, we notice this app assigns a string converted from a user input to variable `v2` (at line 8 in Figure 1), where the user input is identified by searching for its resource ID from line 5 to 7. Then, variable `v2` is used in a validation check at line 11. In this validation, it has two conditions concatenated with logic relation `OR`. In one of the conditions, the app checks whether variable `v2` is equal to a string value, `b***1`,[3] which is hardcoded in plaintext in the app. Because of the `OR` logic, if

[2] The source code is available at github.com/OSUSecLab/InputScope.

[3] We redact the exact content of secret values for apps that have not fixed at the time of this writing and for which disclosure could cause negative impacts for app developers.

```
1  public void onClick(DialogInterface arg7, int arg8) {
2      String v2 = "";
3      View v0 = this.a;
4      int v3 = 0;
5      while(v3 < ((ViewGroup)v0).getChildCount()) {
6          View v1 = ((ViewGroup)v0).getChildAt(v3);
7          if(v1 != null && v1.getId() == 2131624072)
8              v2 = ((EditText)v1).getText().toString();
9          ++v3;
10     }
11     if(v2.equals(this.b) | v2.equals("b***1")) {
12         ... // viewing files
13     } else {
14         Toast.makeText(this,"Incorrect password", 1).show();
15     }
16 }
```

Fig. 1: A backdoor triggered by a master password in a file encryption app.

this sub-condition is satisfied, then the app will allow any user to view all hidden or locked files by the original user who has the correct password (stored in this.b). Otherwise, it displays an error message, "Incorrect password". This hardcoded string is a backdoor secret that can be used to bypass the entire access control mechanism implemented in the app.

Next, we use a dictionary app with 1 million installs as an example to illustrate another type of backdoor, the secret access key. As shown in Figure 2, this app uses variable this.d to store user inputs at line 2. Then, the app converts user input to a string and compares it with a hardcoded string, q***d, to check equivalence. If their values are identical to each other, then the app will remove advertisements displayed in the app. Otherwise, it will continue with the normal actions to translate user input text from English to Arabic. In fact, removing advertisements is an in-app service with fees, which means that this hardcoded string is a backdoor secret to bypass app restrictions.

```
1  public boolean onKey(View arg4, int arg5, KeyEvent arg6) {
2      this.d = this.findViewById(2131296464);
3      int v0 = 66;
4      ...
5      if(arg5 == v0 && arg4 == this.d && arg6.getAction() == 0){
6          if((this.d.getText().toString().equals("q***d"))
                   && this.a != null) {
7              this.a.setVisibility(8); // hide advertisements
8              return 0;
9          }
10         // normal translation actions
11         return 1;
12     ...
13 }
```

Fig. 2: A backdoor triggered by a secret access key in a dictionary app.

Blacklist Secrets. If a list is used to inspect the user input to filter out unwanted items, we call this list a blacklist secret. Many apps use blacklists to validate user input. In the following, we use a popular news app, which has 50,000 total installs in Google Play, and 1.1 billion total installs in all alternative markets together[4], as an example to demonstrate how its validation leaks its blacklist secrets. Because this app has been obfuscated, for better illustration, we use human readable method names (*e.g.*, validate_nickname) instead of the obfuscated names, as shown in Figure 3.

```
1  private void validate_nickname(String arg3, Dialog arg4) {
2      if(!TextUtils.isEmpty(((CharSequence)arg3))) {
3          String v0 = this.a.getText().toString();
4          if(StringUtil.isInterceptedNickName(this.e, v0)) {
5              String v1 = "Nickname contains illegal
6                           characters!";
7              ann.a(this.e).a(v4, v1);
8          } else ...

9  public static boolean isInterceptedNickName
10                               (Context arg5, String arg6) {
11     boolean v0 = false;
12     String v0_0 = "intercepted_word";
13     String v1 = StringUtil.readAssetsTxt(arg5, v0_0);
14     if(!TextUtils.isEmpty(((CharSequence)v1))) {
15         String[] v2 = v1.split("\\|");
16         int v3 = v2.length;
17         int v1_1 = 0;
18         while(v1_1 < v3 && !v0) {
19             if(TextUtils.equals(v2[v1_1], arg6)) {
20                 v0 = true;
21             }
22         }
23     }
24     return v0;
25 }

         File Location: /assets/intercepted_word.txt
```

Fig. 3: Nickname validation revealing a blacklist in a news app.

Specifically, the user input validation for a nickname goes through two methods. First, the variable v0 is assigned with a user input String converted from a UI widget this.a (at line 3). Then, validate_nickname invokes isInterceptedNickName and passes variable v0 as an argument in line 4 for further validation. In the invoked method, variable v0 is denoted as parameter arg6. The app validates parameter arg6 in a while loop against each element stored in an array v2 in line 19. Array v2 contains values loaded from a local file (lines 13–15); the file name is "intercepted_word.txt" located under the "/assets" directory. Within the while loop that starts at line 18, if the app detects a match between parameter arg6 and any element in array v2, then it will inform method validate_nickname of the failure of the validation and validate_nickname will show the error message "Nickname contains illegal characters". In this example, it is clear that the app filters illegal characters in the nickname based on a blacklist, which is stored locally, allowing its content to be extracted.

III. Overview

The goal of InputScope is analyzing the ways mobile apps process user inputs to uncover hidden behaviors. Reaching this goal is by no means trivial. In this section, we present an overview of InputScope. We first describe the challenges we must solve and insights we have in §III-A. Then, we describe how InputScope works in §III-B, and finally we discuss the scope and assumptions in §III-C.

A. Challenges and Insights

We identify three key challenges to build InputScope:

- **C1: How to pinpoint secret-exposing validations.** While an input validation needs to involve comparisons (for

[4] https://www.qimai.cn/

whitelist/blacklist, syntatics/semantics checks), an app can contain many comparison instructions and these comparisons can be implemented in completely different ways across different apps (or even within the same app). Moreover, some checks are not related to hidden functionality (*e.g.*, format checks). Therefore, it is a challenge to pinpoint secret-exposing validations from a large number of comparison instructions, especially without having false positives and compromising scalability.

- **C2: How to resolve the compared content in validations.** After detecting the user input validations of interest, the next step is to resolve the content (*e.g.*, censorship keywords) used in the validation. In some cases, it may be trivial to resolve the content by directly inspecting an instruction that compares with a literal value. However, the content used in the validation could come from a variety of sources, such as hardcoded values, file inputs, or server responses, some of which cannot be resolved via static analysis (*e.g.*, server responses cannot be retrieved without actually connecting to the server). On the other hand, even when compared content can be resolved from the code alone (*e.g.*, hardcoded values), it may be the result of a series of computations, *e.g.*, string concatenations, that cannot be resolved directly.
- **C3: How to identify input-triggered secrets.** Having detected the user input validation and resolved the content used in the corresponding validations, we still need to identify whether a validation exposes input-triggered secrets. However, this is by no means trivial because a validation between the same pair of user input and content could lead to completely different conclusions. For instance, an app may check whether the user provided password is "`123456`". If this occurs in user registration, it could be just checking whether a user-provided password is a blacklisted weak password. However, if this occurs at login, then it could be a backdoor. Therefore, identifying these different cases is another challenge.

After analyzing mobile app code manually, we have obtained the following insights to solve the above challenges.

- **S1: Using taint analysis to pinpoint the input validation of interest.** While an app can contain numerous and different types of comparison, we notice user input validation often starts from input, followed by string conversions if necessary,[5] and then performs the comparison with another object using standard APIs (*e.g.*, `equals` as shown in the two motivating examples in §II-C). Therefore, we can use static taint analysis to taint the user input and monitor whether it propagates to system APIs (*i.e.*, the taint sinks) to detect user input validations in mobile apps at scale.
- **S2: Using backward slicing and string value analysis to resolve the compared content in validation.** With taint analysis, we are able to identify the taint sinks, from which we can identify the compared content. Note that the secrets in this study are often in the form of strings. If the compared string content is directly visible at the type sink, we directly extract its value. Otherwise, we perform backward slicing to identify how the compared string is generated. If it is from external remote input, our analysis will produce no concrete value since we do not perform real execution of the app (but we can output that the type of the content is from remote input). Otherwise, if it originates from internal input, *e.g.*, a file, we then open the file and follow the execution path identified by the backward slicing to retrieve the string. If there are any string operations (*e.g.*, concatenation or substrings), we simulate these operations to obtain the final computed values.
- **S3: Using the comparison contexts of validation to identify input-triggered secrets.** After resolving the compared content used in the validation, we have to identify whether this validation exposes a secret of interest. Our key insight is to use the comparison contexts of the validation extracted from the app code to solve this problem. More specifically, we can construct a comparison context of input validation using two orthogonal pieces of information: (i) the type of either the user input (*e.g.*, a password) or the compared content (*e.g.*, a hardcoded string) used in the validation, and (ii) the code dispatch behavior associated with the result of the validation. For example, as shown in Figure 1, the type of content for validation is a hardcoded value compared with a user input type `password`,[6] and the code dispatch has two actions: the `true` branch, which overlaps with the comparison to `this.b`, and the `false` branch, which rejects invalid passwords. Based on this code execution context, we can conclude it is a master password secret, since a hardcoded secret can cause the same action as a legitimate password. We derive a number of such execution context-based policies to identify other type of input-triggered secrets, detailed in §IV-C.

B. InputScope *Overview*

An overview of InputScope is presented in Figure 4. There are four key components: (i) Input Validation Detection detects the existence of validation behavior with static taint analysis; using the taint sinks, our (ii) Compared Content Resolution performs backward slicing to identify the sources of compared content and then uses the slice to compute the final `String` type value. Next, (iii) Comparison Context Recovery takes the types of user input and compared content, and recovers its code dispatch behavior such as one-to-two, many-to-two, or many-to-many. Finally, using both the comparison context and compared content, (iv) the Secret Uncovering component uses each specific policy to find secrets of interest such as backdoors or censorship keywords.

C. *Scope and Assumptions*

In this paper we focus on input validation that can lead to the identification of backdoors or blacklist secrets; other types of input validation, such as those that may lead to XSS or SQL injection, have been covered extensively in prior work and are out of scope of this paper. InputScope analyzes mobile apps for the Android platform and, in our prototype, we only focus on input validation at the Java bytecode level, and exclude input validation in native libraries.

[5] Note that we have not observed other types of data such as integer or floating point. This is likely because backdoors or censorship blacklist secrets are often stored as strings.

[6] The UI widget of the user input for this particular case is `password` type.

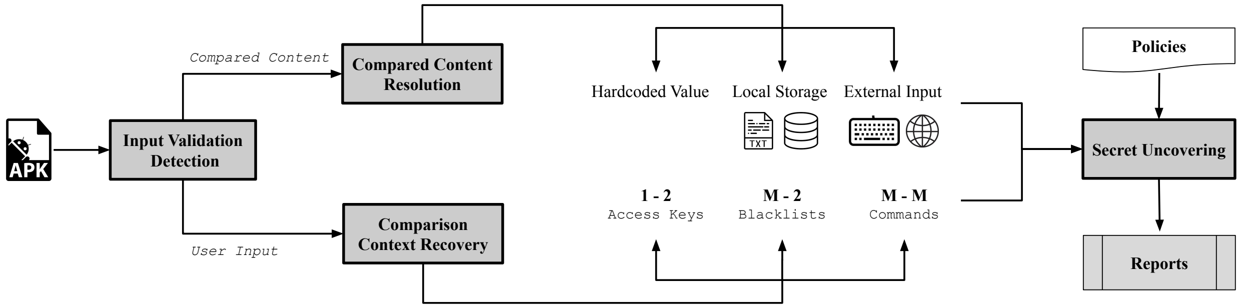

Fig. 4: Overview of INPUTSCOPE.

In addition, we only focus on the user input that is provided by users via keystrokes, namely `EditText`. Moreover, we assume the user input field is implemented using Android UI widgets that allow app to read user input by invoking system APIs (*e.g.*, `EditText.getText`). Other input such as network-input triggered behavior is out of scope as well.

Also, since the secrets of our interest are all concrete values, we particularly focus on the taint sinks that use the `equals` type of comparison. That is, we do not focus on other comparisons (*e.g.*, using regular expressions) for two reasons: (*i*) scalability (we do not want to track too many taint paths), and (*ii*) the nature of our problem (the secrets are concrete string values that are entered from `EditText` from the input perspective, or stored somewhere inside the app from the compared target perspective).

INPUTSCOPE is resilient to many common types of obfuscation (such as variable/class renaming), but can miss some cases where the app's use of system APIs is obfuscated (*e.g.*, through reflection), or where an app uses private APIs to implement its string operations and comparison. Although we hope to cover such cases in future work, such heavily-obfuscated apps are also out of scope for our current work.

IV. DETAILED DESIGN

In this section, we present the detailed design of each component of INPUTSCOPE. Based on the execution order, we first describe how to detect user input validation of interest in §IV-A, then describe our approach to resolve compared content and recover comparison contexts in §IV-B and §IV-C, respectively. Finally, we explain how to uncover backdoors and blacklist secrets based on the information we collected and policies we defined in §IV-D.

A. Input Validation Detection

The key objective of INPUTSCOPE is to uncover hidden behaviors from input validations at scale. Since there are a variety of comparisons in app code, we use static taint analysis to taint user input and monitor its propagation to identify user input validations as discussed in §III-A. Today, there are many open source implementations of static taint analysis, *e.g.*, FlowDroid [7], Amandroid [37], and DroidSafe [20]. We therefore leverage these open source implementations to solve our input validation identification problem instead of developing from scratch.

Type	Class	API
Sources	EditText	getText()
	EditText	getEditableText()
	Editable	toString()
Sinks	Object	equals(Object)
	String	equals(Object)
	String	indexOf(String)
	String	lastIndexOf(String)
	String	equalsIgnoreCase(String)
	String	contentEquals(StringBuffer)
	StringBuffer	indexOf(String)
	StringBuffer	lastIndexOf(String)
	TextUtils	equals(CharSequence, CharSequence)
	HashMap	containsKey(java.lang.Object)
	Map	get(java.lang.Object)

TABLE I: The list of primary taint sources and sinks used in the detection of user-input validation.

Since static taint analysis with mobile apps has been well studied, we omit its technical details for brevity here. In the following, we only describe how we customize its taint sources and taint sinks in our particular problem. These sources and sinks have been derived by systematically examining all Android framework APIs.

- **Taint Sources.** We only focus on user input that comes from local user keystrokes. In Android, this type of user input is obtained by invoking a few specific system APIs. There are three such APIs: `EditText.getText`, `EditText.getEditableText`, and `Editable.toString`, as shown in Table I. Therefore, these system APIs are our taint sources.
- **Taint Sinks.** As discussed in §III-C, we only focus on the system APIs that are used for equivalence checks between strings; this set of APIs is detailed in Table I. Note that, in addition to the APIs that directly check the equivalence (*e.g.*, `equals`), we also include the APIs that can be used for this type of checking indirectly (*e.g.*, `Map.get`).

B. Compared Content Resolution

After the detection of user input validation of interest in a mobile app, next we need to resolve the compared content. Since the secrets in this study are often in the form of concrete strings, the primary objective of our Compared Content Resolution is to resolve the compared string values.

However, these values are not always directly visible at taint sinks. Therefore, we first perform a backward slicing on the bytecode to identify how a the string is generated, and then use string value analysis to obtain the final computed values.

Static Backward Slicing. We use static backward slicing to identify how a compared string is generated. Similar to static taint analysis, static backward slicing is performed on the inter-procedural data-flow graph (IDFG), which is derived from the inter-procedural control-flow graph (ICFG), where the nodes are instructions and the edges are control-flow transfers, but in the opposite direction (since it is backward). At a high level, it starts from where a targeted variable is used and ends at where it is generated. Since a compared string could come from a variety of sources and its value can be generated in different ways (*e.g.*, from a local file, or a remote server response), we have to resolve them accordingly.

In particular, if it comes from external input (either external local input or external remote input), our backward slicing will produce no concrete string value because the value of these external inputs can only be obtained with real executions (*e.g.*, by connecting to remote servers to fetch them). However, if it is from internal input, which is statically carried within a mobile app, either in its program code (*i.e.*, hardcoded values) or its resource files, we use the following policies to identify them.

- **String values from program code.** Since the values from program code are typically hardcoded strings in our focused problem, our backward slicing will stop at APIs such as `getString`. Then we will perform string value analysis (described below) along the data path from where the compared string is generated to where it is used in our taint sinks, to finally resolve the string values.
- **String values from resource file.** There are three types of resource files that contain string values: files (*e.g.*, text files, JSON files), databases (*e.g.*, SQLite databases), and key-value stores (*e.g.*, `sharedPereferences`). Because different types of file store data in different formats, we have to resolve their values accordingly. At a high level, we first resolve its name and file-specific semantics, and then resolve the values of interest. For example, in order to resolve a value from a key-value data `SharedPereference` object, we need to resolve the name of this file and the corresponding "key" to eventually reach the generation of the string.

Meanwhile, to ease the effort for string value analysis in the next step, during the backward slicing, we also maintain an inter-procedural data-dependency graph (IDDG). This IDDG is used to record the computation sequences of relevant string values along the data-flow paths. These sequences are important to reproduce the final string value of the compared content.

String Value Analysis. During the backward slicing, we have obtained a set of targeted string values to resolve. Next, we use a static string value analysis technique we developed earlier in LeakScope [46] (which has been open sourced) to reproduce these string values without actually running the program but simulating the string related computations. In particular, with the IDDG that is maintained during the backward slicing, we forwardly calculate the string value of the target variable by following its original execution order captured in IDDG. During this calculation, we simulate the same operation defined by the system APIs. For example, if the string operation is `substring`, we follow the standard procedure to obtain this substring value. In doing so, we can eventually resolve the string values of the compared content accordingly.

Sources	Class	API

Local Storage	File	getName()
		getAbsolutePath()

	SQLiteDatabase	rawQuery(String, String[])
		openDatabase(String, CursorFactory, int)

	SharedPreferences	getSharedPreferences(String, int)
		getString(String, String)

External Values	Bundle	getString(String)
		getCharSequence(String)

	Intent	getStringExtra(String)
		getCharSequenceExtra(String)

	EditText
	Editable
		getText()
		toString()

	Socket SSLSocket	getInputStream() getInputStream()

TABLE II: The list of the system APIs used for uncovering the types of the compared content.

Pruning Compared Content That is Known by Users. Recall that the primary objective of this study is to uncover hidden behaviors, such as backdoors and blacklist secrets, and these secrets should be unknown to the majority of normal users. However, some of the compared content we resolved could come from visible user interfaces (*e.g.*, `EditText.setHint`). Therefore, we have to prune the resolved compared strings of which normal users are already aware.

To this end, we need to understand the specific text of which users are aware, before typing them into the input field. According to our observation, mobile apps often provide sufficient information in their interface where they ask users to type text and mobile users rarely consult other materials than descriptions displayed in the interface. In other words, the majority of users are only aware of the descriptions from the interface before typing text into input fields. In addition, these descriptions could be either static strings existing in related resource files associated with resource IDs or strings hardcoded in the code that are dynamically loaded by invoking system APIs (*e.g.*, `EditText.setHint`) that can be obtained automatically in the same way as described above. In either case, if we identify the strings that come from these sources, we exclude this comparison in our result.

C. Comparison Context Recovery

We have to use the comparison context, *i.e.*, how a user input is compared and its code dispatch behavior, to determine the hidden behaviors (*e.g.*, backdoors) and their types (*e.g.*, censorship keywords). In general, the code dispatch of a user input could have two attributes: (i) how many times a user input is validated within a judgement block of a method, and (ii) how many potential branches could be taken if the validation is satisfied. These two attributes together can reveal how a user input is validated in terms of deciding the code execution flow.

To simplify the description of the code dispatch context, we present it in the form of a pair of these two attributes. Since we are interested in understanding the overall quantity property of an attribute (one, two, or more than two) rather than its exact number, we mark each attribute as “one”, “two”, or “many”. Meanwhile, since each satisfied condition can only produce two branches (true or false), we consider it two actions. By counting how many times an input is compared, and also how many actions the comparison can generate, we can have (*i*) one comparison and two actions, (*ii*) multiple comparisons and two actions, and (*iii*) multiple comparisons and multiple actions. More specifically, we classify dispatch behaviors as:

- **One-to-Two Dispatch.** This code dispatch indicates that a user input is validated only once in a judgement block within a method. Accordingly, there is only one desired branch to be taken if the condition of user input validation is satisfied. An example of such dispatch is the single `if` block between line 6 and 9 shown in Figure 2.
- **Many-to-Two Dispatch.** This code patch means that a user input is validated multiple times in a judgement block. But there would be only one desired branch that will be taken if any of these validations is satisfied. An example of such a dispatch is presented in Figure 3, where the user input is validated with every element in an array. In this case, each comparison between the user input and an element in the array is one condition. Consequently, it has “many” conditions. However, regardless of which condition is satisfied, there would be one desired dispatch to be taken.
- **Many-to-Many Dispatch.** If there are multiple comparisons and multiple actions, then it means that a user input is validated multiple times with different compared targets, and multiple outcomes can be generated depending on the comparison. A representative example for such dispatch is the `switch-case` block, where each action is assigned to a unique case.

D. Secrets Uncovering

Having recovered code dispatches and the resolved compared content for user input validation of interest, next our Secret Uncovering component will use a set of specific policies to uncover the hidden behaviors and secrets. In total, we have defined four policies to uncover four types of hidden behaviors: secret access keys, master passwords, secret commands, and blacklist secrets, based on the three types of different code dispatch behaviors we have recovered.

(I). Uncovering hidden behaviors from one-to-two code dispatch. With this type of code dispatch, since the user input will only be validated once in a method of the app and the compared content is also not known to the user, and meanwhile there are only two outcome actions resulting from the comparison, we can conclude it is likely that the user input serves as a key to unlock a behavior and such a user input can be considered a secret access key.

However, there are still caveats because in some apps, there could be a normal service instead of a hidden service that requires users to type text not shown in the UI for further functionality. For example, in some puzzle game apps, users could be asked to provide correct answers to go to the next round. In such cases, users are also unaware of what to enter. Fortunately, we can use another dimension of the compared content, namely whether the compared content is from an internal hardcoded string inside the app or not. This is because for these interactive types of apps, especially games, they would have made their compared target more flexible (*e.g.*, coming from network servers) instead of directly hardcoding them in the app (otherwise, it can easily lead to game cheating). Therefore, we use the following policy to decide whether there is a secret access key:

> **Identifying a Secret Access Key.**
>
> *A secret access key is identified if (i) the code dispatch of a user input validation is one-to-two and (ii) the compared content is a hardcoded string inside the app.*

(II). Uncovering secrets from many-to-two dispatch. In this code dispatch, the user input is validated more than once in a method and the satisfaction for different validations all lead to the same program behavior. Meanwhile, for all of the validations with the same user input, its compared content could be from one source or multiple sources. Therefore, we further break down this code dispatch context into two categories:

- **Compared Content from Multiple Sources.** If the compared content comes from multiple sources, then this type of comparison illustrates a scenario where, within a method, if a user input is equal to any value among multiple sources, the program will perform the same action. In other words, each compared value can override others. Therefore, if one of these values is a secret hardcoded string, then such a string can be used to override other sources of values to drive the app into the same state. Note that the compared content from different sources indicates that they are generated in different ways and their values are supposed to be different. An example of such behavior is shown in Figure 1, where a hardcoded string in the comparison and also another source of input together decide the branch outcome. Therefore, this behavior is a hidden feature because normal users are unaware the existence of such a string. Inspired by the actions from Figure 1, we call this type of string a master password. However, we do not have to explicitly use the `password` type of `EditText` to decide this master password type of backdoor, because the code pattern of (*i*) multiple sources of compared content and (*ii*) a hardcoded string that can override other input sources has already sufficiently allowed us to decide it is a master password.

> **Identifying a Master Password.**
>
> *A master password is uncovered if (i) the comparison context of user input validation has the many-to-two code dispatch, (ii) the compared content comes from multiple different sources, and (iii) one of the compared content is a secret hardcoded string.*

- **Compared Content from the Same Source.** If the compared content is all from the same source, then this type of comparison context presents a scenario where, within a method, if a user input is equal to any value of the compared content, the app will always move to the same state. In other words, these compared content items together form a list, and the user input is compared with every item in the list to check equivalence. Each equivalence results in the same program behavior. An example of such comparison is shown in Figure 3, where the app validates the user input with a blacklist to identify the forbidden keywords. Therefore, the compared content actually forms a blacklist, and we can use the following policy to detect it.

Identifying a Blacklist Secret.

A blacklist secret is identified if (i) the comparison context of user input validation has many-to-two code dispatch and (ii) the compared content all come from the same source.

(III). Uncovering secrets from many-to-many dispatch. In a many-to-many dispatch, the same user input is validated with different compared values, either from the same source or different sources. Meanwhile, if some of the compared content is resolved as secret strings, then such a context indicates that, within a method, a user input could take a value from a set of secret strings and each string can trigger a different program action. In other words, the value space of a user input contains a subset of concrete strings whose values are unknown to normal users, and each of them can drive the app into a different state. Such behavior is very similar to a terminal that accepts different commands. Therefore, we call these secret strings secret commands and we use the following policy to identify them.

Identifying a Secret Command.

A secret command is identified if (i) the comparison context of user input validation has many-to-many code dispatch and (ii) the compared content includes more than one hardcoded secret string.

V. Evaluation

We have implemented a prototype of InputScope atop Soot [2] and LeakScope [46], with borrowed code from FlowDroid [7] to statically detect the user-input validation, reveal its contexts, and extract its compared content. In total, InputScope consists of around 5,500 lines of our own code. In this section, we present the evaluation results. We first describe how the evaluation is set up in §V-A, and then present our detailed evaluation results in §V-B.

A. Evaluation Setup

Dataset Collection. We collected the Android apps from three different sources to evaluate InputScope. The first source is Google Play, which is the largest world-wide Android app market. To ensure a reasonable distribution of the apps, we successfully crawled the top 100,000 free apps across all categories based on number of installations at the end of April, 2019. The second source is from an alternative app store, Baidu Market, from which we have crawled the top 20,000 free apps during the same time period as our crawl of Google Play apps. The third source is pre-installed apps, and we obtain 30,000 of them directly from over 1,000 Samsung firmware images, which were downloaded from SamMobile[7]. Altogether, our dataset consists of 150,000 mobile apps.

Item	Value
# Apps tested	150,000
# Apps containing equivalence checking	114,797
# Apps check empty input only	34,958
# Apps check non-empty input	79,839
# Apps contain backdoor secrets	12,706
% Apps in Google Play	6.86%
% Apps in alternative Market	5.32%
% Apps in pre-installed apps	15.96%
# Apps - secret access keys	7,584
# Apps - master passwords	501
# Apps - secret privileged commands	6,013
# Apps contain blacklist secrets	4,028
% Apps in Google Play	1.98%
% Apps in alternative Market	4.46%
% Apps in pre-installed apps	3.87%

TABLE III: Overall statistics of the evaluation results.

Testing Environment. We use two servers to run our experiments. One server runs Ubuntu 16.04 with 256 GB memory and an Intel Xeon E5-2695 v4 CPU that crawls apps from the Google Play and analyzes them with InputScope, and the other one runs Ubuntu 16.04 with an AMD EPYC 7251 CPU and 256G memory that is in charge of extracting the pre-installed apps from Samsung firmware images, downloading apps from the alternative market, and executing InputScope to analyze these apps.

B. Evaluation Results

In total, InputScope took around 24 days to discover mobile apps containing backdoors or blacklist secrets from these 150,000 mobile apps. Specifically, as presented in Table III, we first identified 114,797 mobile apps that contain equivalence checking. Note that an app can detect whether a user input is empty by simply checking whether the input is equivalent to an empty string. There are 34,958 mobile apps that perform these empty-only checks, and we thus exclude them from further analysis. In the remaining 79,839 mobile apps, InputScope identified 4,028 apps containing blacklist secrets and 12,706 apps containing backdoor secrets. There are 7,584 apps with secret access keys, 501 apps that embed master passwords, and 6,013 apps with secret commands. Moreover, these security risks hold generally across all of our data sources. Specifically, the prevalence of backdoor secrets in apps is 6.86%, 5.32%, and 15.96% on the Google Play store, the alternative market, and pre-installed apps, respectively, and the percentage of apps containing blacklist secrets in these three data sources are 1.98%, 4.46%, and 3.87%.

Next, we examine these results in greater detail to understand two key questions. First, what kind of advantage could be taken by using the uncovered hidden behaviors such as backdoors? Second, what are the detailed items in a blacklist, and why they are blocked? To this end, we have manually inspected the top apps in each category, and we present here a detailed security analysis. Note that the top apps from Google Play and Baidu Market can be easily identified based on the download numbers from their app stores, but we cannot

[7] https://www.sammobile.com/

Usage Description	# Installs	Category	Package Name	Access Keys
Hidden Admin Interface Login	5,000,000 - 10,000,000	Sports	air.**.l****	U***S
	5,000,000 - 10,000,000	Dating	e**.o****	i***g
	1,000,000 - 5,000,000	Social	co.**.g****	$***n
	1,000,000 - 5,000,000	Travel	com.**.j****	J***!#
	1,000,000 - 5,000,000	News	com.**.a****	w***#
Arbitrary User Password Recovery	10,000,000 - 50,000,000	Health	org.**.p****	8***8
	5,000,000 - 10,000,000	Personal.	com.l*.k****	0**9*
	1,000,000 - 5,000,000	Games	com.**.c****	q***3
	1,000,000 - 5,000,000	Product.	com.**.cu***	h****
	1,000,000 - 5,000,000	Lifestyle	com.**.p****	*6**0
Advanced Service Payment Bypassing	1,000,000 - 5,000,000	Product.	com.**.n****	#***+
	1,000,000 - 5,000,000	Books	com.g*.d****	q***d
	1,000,000 - 5,000,000	Books	com.g*.d****	q***d
	1,000,000 - 5,000,000	Books	com.g*.t****	q***d
	1,000,000 - 5,000,000	Product.	vo*.**.tr***	q***d

TABLE IV: Detailed results of top inspected mobile apps containing secret access keys.

# Installs	Category	Package Name	Master Pwd
10,000,000 - 50,000,000	Tools	com.a****	9***8
5,000,000 - 10,000,000	Tools	s**.c*.g****	1*6**
5,000,000 - 10,000,000	Health	in.p*.l****	*9**2
1,000,000 - 5,000,000	Tools	com.m*.p****	4***2
1,000,000 - 5,000,000	Entert.	com.kiddoware.kidsplace	5493
1,000,000 - 5,000,000	Finance	com.v*.p****	o*f*s
1,000,000 - 5,000,000	Tools	it.v*.d****	1v**3
500,000 - 1,000,000	Parenting	com.*s.m****	****1
500,000 - 1,000,000	Product.	com.movinapp.quicknote	1349100416
500,000 - 1,000,000	Tools	com.s*.h***s	b*r*1

TABLE V: Detailed results of top inspected mobile apps containing master passwords.

identify the top apps from the pre-installed dataset since all of them are installed when users purchased the phone and likely have the same distribution. We therefore only focus on the apps from the app stores in our case studies below, though we have also observed similar patterns in pre-installed apps.

1) **Hidden Backdoor Behaviors:** Since INPUTSCOPE has discovered three types of input-triggered hidden behaviors using (*i*) secret access keys, (*ii*) master passwords, and (*iii*) secret commands, we present detailed analysis for each of these categories in the following.

(I). Hidden Behaviors Triggered by Secret Access Keys. To better understand why such hidden behaviors exist, we have manually inspected a set of 30 apps that are randomly selected from the apps with more than one million installs and summarized the three most common types of usage that we can recognize. In addition, we present the detailed results of the top five apps for each usage, 15 apps in total, in Table IV, where the first column describes the type of usage, the next three columns provide the number of downloads, its category, and its package name, respectively, and the last column shows the identified secret access key for each app. We have summarized the following three types of usages with our best effort as also listed in Table IV:

- **Logging into administrator interfaces.** We have identified access keys that can be used to log into an app's administrator interface, which is invisible to normal users and allows users to change the configuration of the app. An example in this case is a very popular sports live streaming app with more than 5 million installs. In particular, it allows anyone to login as an administrator with the access key "U***S" from the hidden administrator interface in its "`Setting`" menu. After the successful login, the administrator interface allows an attacker to perform privileged actions such as changing configuration URLs, changing network IDs, or resetting a "temporary pass".
- **Resetting arbitrary user passwords.** We have also discovered access keys to trigger the hidden behaviors of recovering or resetting normal users' passwords. We take a popular app providing screen-locking services with more than 5 million installs as an example. To launch this attack, first, attackers can simply trigger a hidden button after multiple trials with a wrong password. Then, attackers can click the hidden button to get a new interface where a special code is requested. After providing the code 0**9*, attackers can reset the password to unlock the screen.
- **Bypassing advanced service payment.** We also have verified there are access keys that can purchase in-app advanced services for free. For instance, we have extracted an access key, q***d, from a popular translation app with more than one million installs. Similar to the motivating example, by simply typing this code in the `EditText` which accepts text for translation and clicking the `translate` button, one can remove the advertisement displayed in the app for free. However, removing advertisements is a service available for purchase with a fee of $12.99.

From these detailed case studies, we can notice that the user input validations in apps can expose their secret access keys and can be used to launch various attacks against both the users of the mobile app (*e.g.*, resetting their passwords) and also the service providers of the app (*e.g.*, bypassing their service payment). Also, surprisingly, these types of mistakes can even occur in popular apps with millions of installs. In addition, we observed that the same group of developers could make the same mistake across all of their apps.

(II). Hidden Behaviors Triggered by Master Passwords. INPUTSCOPE has identified 501 master passwords among the tested apps. We also randomly selected 10 popular apps to understand the hidden behaviors triggered by these master passwords, and the result for these apps is presented in Table V. Since a master password can be used to hijack/override another compared target, it is extremely dangerous. During our manual investigation with these apps, we found a security related app with more than 10 million installs, which is designed to help a user lock their smartphone when it is lost by allowing legitimate users to control the phone remotely. While this app provides many different security mechanisms to protect its users, *e.g.*, remotely wiping the phone via SMS, it contains a master password `9***8` to bypass its protection on the privacy apps that are set to be locked when the phone is lost. Another interesting case is a diary app where users can lock the diary with their passwords. However, an attacker can use the password lv**3 to unlock the secret diary, although the app will display a text at the bottom of the screen saying "`wrong password`".

(III). Hidden Behaviors Triggered by Secret Commands. INPUTSCOPE identified 6,013 mobile apps containing secret

# Installs	Category	Package Name	Commands
10,000,000 - 50,000,000	Tools	com.*.whe*d	w***l, d***n, B***u, w***k ...
10,000,000 - 50,000,000	Music&Audio	com.th.ringtone.maker	enableartistalbum, disableartistalbum ...
10,000,000 - 50,000,000	Games	ru.c*.s****	8***4, 82***, 6***9 ...
5,000,000 - 10,000,000	Education	w*.*n.****g	t***e on, b***l, f***e, b****1, d***g ...
1,000,000 - 5,000,000	Shopping	com.b*.a***y	p***f, d***p, c***f, p***n, d***s ...
1,000,000 - 5,000,000	Education	com.*a.b**n*	S***D#, M****, G***S, G***I, D***P, C***R ...
1,000,000 - 5,000,000	Games	com.c*.f***s	c***h, e***t ...
1,000,000 - 5,000,000	Social	com.c*.s****	*#0*#, *#*1#, *#*3#, *#*5#, *#*2# ...
1,000,000 - 5,000,000	Games	com.h*.e****	un**s, lo**l, lo**s, un*** ...
1,000,000 - 5,000,000	Productivity	com.lfantasia.android.outworld	(maroonAuth), (amberAuth), (darkCyanAuth) ...

TABLE VI: Detailed results of top identified mobile apps containing secret commands.

commands. As before, we manually inspected the commands in the top 10 mobile apps according to their number of installations and summarized their common usages; this detailed result is presented in Table VI. The first column shows the number of installs, followed by its category, its package name, and the uncovered secret commands. We found that these commands can be classified into two categories: debugging and non-debugging, based on whether the commands are for developer use or not.

- **Commands for Debugging.** The most common use of these commands is to drive the app into debug mode and test the app's low level functionality. Many of the identified secret commands belong to this category. For example, as presented in Table VI, the shopping app can debug HTTP connections and proxy via the `d***p` and `p***f` commands, respectively. An education app can activate test mode using the command `t***e on`.
- **Commands for Other Functionality.** Other than debugging, which can be easily identified, the remaining commands fall into other categories, such as triggering hidden functions that are unknown to normal users. For instance, a social app contains various commands such as `*#0*3#` and `*#*2#` to trigger various hidden functions such as clearing all cached data and account settings. Similarly, an education app can use `C***R` to clear users' settings.

2) **Hidden Blacklists Secrets:** Given the diverse content a blacklist may contain, to understand why there exists such blacklist, we manually investigated the top 20 popular apps that expose their blacklist secrets based on the size of their blacklists. In the following, we provide our analysis of the blacklists with these apps at both the aggregated (macro) level and fine-grained (micro) level.

Aggregated Macro Results. We show the aggregated macro-level results of these apps in Table VII, where the first column shows the market to which the app belongs, the second column shows where its blacklist is stored, followed by its number of installs, its blacklist's content languages, and its blacklist size in terms of number of items.

- **Languages.** We found that the content comes from three different languages: Chinese, English, and Korean. This indicates that the usage of blacklists is not restricted to a specific country or language. Interestingly, we found that even when the primary language is not English, the blacklist usually involves several English words; however, if the primary language is English, then we did not see any case where a blacklist contained words in other languages.

M	S	# Installs	Package Name	Lang.	Size
Google Play	Local Storage	10,000,000 - 50,000,000	com.*.p**r*	E	324
		10,000,000 - 50,000,000	c**.f****	E	1,000
		5,000,000 - 10,000,000	com.w*.s****	C E	10,439
		500,000 - 1,000,000	com.k*.j****	E	1,594
		100,000 - 500,000	com.p*.p****	E	78
	Hardcode Str	5,000,000 - 10,000,000	com.s*.c***t	E K	27
		1,000,000 - 5,000,000	com.q***k	E	13
		100,000 - 500,000	com.b*.l***y	E	7
		100,000 - 500,000	in.*.l*.v***t	E	16
		50,000 - 100,000	kr.**.z*.d****	E K	562
Alternative	Local Storage	50,000,000 - 100,000,000	com.*.t****	C	1,958
		50,000,000 - 100,000,000	com.y*.t****	C	3,366
		10,000,000 - 50,000,000	com.i**i**	C	1,960
		1,000,000 - 5,000,000	com.y*.w****	C E	3,966
		1,000,000 - 5,000,000	com.m*.i****	C E	4,154
	Hardcode Str	10,000,000 - 50,000,000	com.z*.h****	C E	145
		10,000,000 - 50,000,000	com.**.q****	C	372
		5,000,000 - 10,000,000	com.a*.*****	C	87
		1,000,000 - 5,000,000	com.j*.s****	C	93
		1,000,000 - 5,000,000	y**.E**n**	C E	451

TABLE VII: Aggregated results of top tested apps containing black-lists: M for Market, S for Source of a blacklist, E for English, C for Chinese, K for Korean.

- **Sizes.** We observed that the size of the blacklist varies across apps regardless of their popularity, from more than 10,000 items to only 7 items in the list. In general, blacklists read from local storage contain more items than those hardcoded in the code, and Chinese blacklists contain many more items than Korean or English blacklists, where the size of English blacklist is relatively smaller than the other two languages. That might be result of the fact that Chinese blacklists cover more

Category	Detailed Blacklist Type
Drug	01-Addictive Drug, 02-Aphrodisiac, 03-Hallucinogen
Cult	04-Cults Name, 05-Malignant Event
Fraud	06-Fake Certificates, 07-MLM
Gamble	08-Chess & Card, 09-Lottery, 10-Jockey
Insult	11-Bullying, 12-Racial Discrimination, 13-Obscenity
Password	14-Weak Password
Politics	15-Leaders Name, 16-Mass Incident, 17-Rebel 18-Parade, 19-Separatist
Pornography	20-Adult Video, 21-Escort Service
Website	22-Anti-government, 23-Fake News, 24-Pornography 25-Criminal

TABLE VIII: Blacklist types

			Drug			Cult		Fraud		Gamble			Insult			PWD	Politics					Porn		Website			
Market	**Category**	**Package Name**	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Google Play	Games	com.*.p**r*	○	○	○	○	○	○	○	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○
	Social	c**.f****	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○
	Games	com.w*.s****	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●
	Entertainment	com.k*.j****	○	○	○	○	○	○	○	○	○	○	●	●	●	○	○	○	○	○	○	○	●	○	○	○	○
	Social	com.p*.p****	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○
	Games	com.s*.c***t	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○
	Lifestyle	com.q***k	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○
	Lifestyle	com.b*.l***y	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○
	Social	in.*.l*.v***t	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	●	○	○	○	○
	Communication	kr.**.z*.d****	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	●	○	○	○	○
Alternative	Education	com.*.t****	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●
	Education	com.y*.t****	●	●	●	●	●	●	○	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	○	●	○
	Social	com.i**i**	●	●	●	●	○	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	○	●	●
	Shopping	com.y*.w****	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	●	●	●
	Entertainment	com.m*.i****	●	●	●	●	●	●	○	●	●	●	●	●	●	○	●	●	●	●	●	●	●	●	○	●	●
	Productivity	com.z*.h****	○	○	○	●	●	●	●	○	○	○	○	○	●	○	●	●	○	●	●	○	●	○	●	○	○
	Entertainment	com.**.q****	○	○	○	●	●	●	○	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	○	●	○
	Social	com.a*.*****	○	●	○	●	○	●	○	○	○	○	●	○	●	○	●	●	●	○	●	●	○	○	○	○	●
	Entertainment	com.j*.s****	○	○	●	●	○	●	●	○	○	○	●	○	●	○	●	●	○	●	●	●	●	○	○	○	○
	Education	y**.E**n**	●	○	●	●	●	●	○	●	●	○	●	○	●	○	●	●	○	○	●	●	●	●	●	●	●
Statistics			7	7	8	11	8	11	6	7	8	6	19	9	20	1	11	11	8	8	11	10	13	8	5	8	7

TABLE IX: Fine-grained results of blacklist for top tested apps: ● for presence, and ○ for absence.

categories than Korean and English blacklists. More details will be presented in the following analysis.

Fine-grained Micro Results. After understanding the content of blacklist at the aggregated macro level, we then zoom into each blacklist to understand them at fine-grained micro-level. We created a best-effort classification of their content into 9 semantic categories including *drug, cult, fraud, gambling, insult, password, politics, pornography*, and *website*. In addition, we also list the recognized 25 micro-level types of content for each semantic category in Table VIII. Note that we only present the micro-level classification of the keywords of the blacklist instead of the exact words at Table IX, given their inappropriate content and large size.

- **Commonly blocked content in three languages.** We can observe that all blacklists in three languages filter keywords in the category of *insult* and *pornography*, according to Table IX. In particular, in the category of *insult*, there are 20 blacklists that filter keywords related to the concept of *obscenity*, 19 blacklists (except one blacklist in Chinese) that also block keywords used for *bullying*, and 9 blacklists that filter expressions related to *racial discrimination*; three contain only English keywords, three contain only Chinese words, and three blacklists contain content in both of these languages. Meanwhile, in the category of *pornography*, there is one blacklist containing both English and Korean content, four blacklists containing English and Chinese words, as well as two English exclusive blacklists and 7 Chinese exclusive blacklists that block keywords related to *escort services*. Finally, there are 7 Chinese-exclusive and 4 Chinese and English combined blacklists that block *adult video*.
- **Uniquely blocked content in each specific language.** Besides the commonly blocked content, we noticed one English blacklist containing items that we classify as *weak passwords*, while no blacklist in the other two languages filters such passwords. As for blacklist content in Korean, first, we did not witness a blacklist containing Korean content exclusively, and second, blacklists with Korean content in our dataset block no unique content other than porn and insults. However, we did find blacklists consisting of Chinese keywords covering 6 unique semantic categories (*i.e.*, *drug, cult, fraud, gamble, politic*, and *website*) with 19 micro-level types defined in Table VIII. Specifically, in the *drug* category, 8 blacklists block keywords relating to *hallucinogens*, 7 blacklists filter *addictive drug*, and also 7 blacklists forbid content related to *aphrodisiacs*. In the category of *cult*, we have 11 blacklists that disallow *cult name* and 8 disallow mentioning *malignant event*. As for the category of *politics*, keywords relating to *leader names*, *mass incidents*, and slogans for *separatism* are blocked by all Chinese blacklists. In addition, there are also 8 blacklists that forbid words from the *rebel* and *parade* categories. Interestingly, only Chinese blacklists try to filter information about *fraud* and *gambling*. In particular, 11 of them block content for forging *fake certificates*, and 6 of them disallow advertisements about *multi-level marketing (MLM)* organizations. For gambling, keywords related to *lottery* are blocked by 8 blacklists, names of *chess & card* games are disallowed by 7 blacklists, and information about Hong Kong *Jockey* (an organization that allows betting on horse racing and other sports) is also forbidden in 6 blacklists. Finally, there are 8 blacklists that disallow sharing the URL of websites whose content includes supporting *anti-government* and showing *pornography*, 7 also forbid criminal websites, and 5 filter websites disseminating *fake news*.

From these results, we can make several interesting observations. First, a keyword might be forbidden on one platform but would be accepted on another platform, even if these platforms intend to filter the same semantic category of words. For example, there are 9 platforms that block words related to *racial discrimination*, while the other 11 won't, even though all platforms in this study try to filter *insult* expressions. Second, Chinese blacklists cover many more semantic categories than the blacklists consists of other two languages. Besides filtering keywords in semantic categories such as *politic*, *cult*, and *gamble* that could be a result of political or law enforcement reasons, they also try to exclude content that might cause damage to people's lives, *e.g.*, *drug*, *fraud*, and criminal *website*. Moreover, another interesting observation is that mobile apps may use blacklist-based methods to validate weak passwords, though we only encountered one such case in our manual investigation.

From a security perspective, blacklist identification and extraction has two benefits. First, developers may have an interest in preventing abuse and harassment on their platforms, and may be unaware that client-side enforcement is ineffective at providing this capability. Second, users may be unaware that an app is limiting their freedom of expression, and exposing types of content being filtered can help them make more informed choices about what platforms they participate in.

VI. Discussion

A. Accuracy of Secrets Uncovering

InputScope relies on static analysis with a set of security policies to identify a variety of secrets that can trigger hidden behaviors within an app. To better understand these behaviors and evaluate the accuracy of our secret uncovering policies, we manually analyzed the top popular apps. More specifically, we first decompiled each app and inspected its code to identify whether the secret values we discovered can actually trigger actions (*e.g.*, invoking methods). If so, then we moved on to understand the purpose of this action by reading the code as well as finding the correct way to navigate the app and try to trigger the action for dynamic verification. Among the total number of 70 apps we have manually analyzed with our best effort and understanding, we have identified 1 misclassification and 8 false positives, resulting an accuracy of 87.14%.

In particular, a false positive in this study refers to an extracted value that (i) cannot trigger actions, (ii) triggers behaviors that can be achieved by normal operations, or (iii) where the triggered action is benign even though it cannot be triggered normally. In our manual analysis, we have identified 8 false positive cases where 6 of them are flagged as backdoor secrets of access keys and 2 as secret commands. Specifically, three false positives occur because the identified values will not trigger actions in practice because of conflicting constraints along the execution path; the other three false positives are caused by misclassifying benign behavior: two cases where the values are used for benign "Easter eggs", and one where they are used to provide (benign) special location-based services. The remaining two false positives were both identified as hidden commands: the identified commands for one app are a set of shortcuts for normal operations, and the other one uses hidden commands to change UI rendering. In addition, we also noticed 1 misclassification case where a set of secret commands has been flagged as blacklist secrets.

B. Limitations and Future Work

In the following, we discuss limitations and future challenges to improve the accuracy of the analysis performed by InputScope:

- The first challenge for InputScope is supporting the WebView component. Mobile apps using the WebView component may rely on JavaScript routines to collect and validate user inputs. As InputScope currently operates at the Java bytecode level only, it may not be able to analyze these apps and unveil potential hidden behaviors.
- The second challenge for InputScope is handling custom-defined string operations. InputScope currently relies on the system API functions for string comparison. However, apps may use customized or third-party string comparison operations, and InputScope will not be able to identify them.
- The third challenge for InputScope is when apps validate user input via database queries, *e.g.*, SQL queries. Extracting the execution context of data flows crossing the boundaries of the database API requires inferring the semantics of both queries and the database structure. Such a challenge has been partly solved only when additional artifacts, *e.g.*, initialization scripts, are found in the code [17]. However, this is not necessarily the case in the mobile app setting, and it requires a more general solution.
- The fourth challenge comes from our manual classification of the blacklists, which may result in misclassifications caused by our unfamiliarity with the topics and the language gap. In future work, we hope to perform a deeper analysis of these blacklists with a broader diversity of researchers from different backgrounds.
- Finally, InputScope has false positives for various reasons such as ignoring path constraints and failing to distinguish "benign" cases. We plan to address these issues by combining other techniques such as symbolic execution to prune impossible paths and machine learning to infer developers' real intention.

C. Why Hidden Behaviors Exist and How to Address Them

InputScope has uncovered a number of serious security issues from user-input validation implementations. In the following, we analyze their root causes and provide practical solutions accordingly.

Misplaced the Trust in Untrusted Client Software. InputScope has identified 7,584 apps containing secret access keys to trigger various hidden logic, such as bypassing payment. Our findings suggest that, to date, developers still wrongly assume that reversing the code of their apps for inspection is not a real threat. Accordingly, developers tend to implement high privilege interfaces in the mobile apps, mistakenly trusting untrusted client apps. To really secure their apps, developers need to perform security-relevant user-input validations on the backend servers. When enforcing server-side checks is not feasible, then developers should consider using trusted hardware components available on modern mobile devices (*e.g.*, TrustZone).

Removing Debugging Code Before Releasing the Software. InputScope has also discovered thousands of apps containing debugging features. These features need to be removed before deploying a mobile app in the store or in the device firmware. In fact, motivated users can reverse engineer the code of the apps to discover these hidden interfaces. One use of InputScope is to raise developer awareness and demonstrate the reverse engineering process can be fully automated. Therefore, our recommendation is to always remove unnecessary code, including debugging mode code, prior to software release.

Defending against our Secret-uncovering Analysis. We have demonstrated that with INPUTSCOPE a variety of app secrets can be discovered. In certain cases, there may be a need to protect these secrets against our analysis. For instance, an app may consider its blacklist a secret, and developers cannot use the trusted server or TrustZone to perform the input validations, *e.g.*, client-side blacklist filtering is inevitable in time-sensitive services such as live-streaming media. To defeat our analysis, there could be a number of possible avenues. For instance, an app can use obfuscation, or implement secret input validation in the native code, or dynamically load the secrets from remote servers to thwart our secret discovery. However, we note that many of these countermeasures could themselves be bypassed with additional implementation effort.

D. Ethics and Responsible Disclosure

We have taken ethical considerations seriously in every step of our research. First, we only validated the vulnerabilities on our own accounts and our own smartphones (during our deep case studies), and we never try to compromise other users' accounts and smartphones. Second, we did not intentionally manipulate or send forged requests to test the security mechanisms on the server-side.

The hidden functionality that INPUTSCOPE has identified can have severe consequences to either app users or developers, and these apps need to be patched by app developers. Therefore, we have contacted developers for each manually verified app to disclose our findings. Our disclosure process includes two steps: first we used the contact information left in the related market to ask for the correct contact information to disclose vulnerabilities, and then we disclosed the details to the correct security contact. For those vulnerable apps that have not yet been patched at the time of this writing, we redacted their package names as well as their secret values with the symbol "***", in order to avoid negative impacts (*e.g.*, economic hardship from disclosure of advertisement removal keys). We will continue to engage with the app developers to offer help with our best efforts.

VII. RELATED WORK

Static Taint Analysis. Our approach is based on static analysis to detect the user input validation behaviors within a given mobile app by tracking the user input data flows and their related operations. In the past several years, there have been many efforts that use static analysis for vulnerability discovery by tracking sensitive data flows in mobile apps. For instance, Flowdroid [7] and Amandroid [37] are generic approaches to track security-related data flows. WARDroid [26], Extractocol [15], and SmartGen [45] focus more on the data flow related to network communications. PlayDrone [35] and LeakScope [46] extract hard-coded secret keys that are used by apps to retrieve cloud-based services. Inspired by this work, INPUTSCOPE tracks only local user input through `EditText` to solve our particularly targeted problem.

Input Validation. Input validation has been well studied in the literature. However, previous studies either focus on the web applications [4], [8], [9], [16], [24], [27], [28], [34], [36], including XSS and SQL injection, or primarily target security issues on the server-side (e.g., [5], [48]). For Android mobile apps, recently WARDroid [26] analyzed issues caused by both the client-side and the server-side. There are also efforts focusing on input validation in Android system services [12], [40], [41], or IoT apps for vulnerability discovery [14], [47]. Different from these works, our study intends to recognize hidden behaviors (or secrets) unknown to normal users in Android mobile apps.

In addition, there is also a body of research focusing on how to generate inputs based on UI information of the apps. For example, AppsPlayground [30], SmartDroid [44], Dynodroid [25], and SMV-Hunters [33] are capable of exploring mobile app behaviors by recognizing UI elements and generating appropriate user input accordingly. However, this work generates input dynamically. In our work, we leverage static analysis and only focus on string related input generation.

User-input Analysis. There are also numerous works to detect security issues related to user input in Android apps. For instance, AsDroid [22] detects stealthy malicious behavior by monitoring the differences between program behaviors and the semantics inferred from the UI text, which includes descriptions for user input. In addition, SUPOR [21] and UIPicker [29] both apply NLP techniques and supervised classification to detect sensitive privacy data from user input. Unlike leveraging UI text to detect malicious behaviors, our work focus on user input in general to recognize its hidden behaviors through carefully defined validation context that is recovered from the code of mobile apps.

Malware Detection. Prior efforts also focus on finding hidden malware behaviors. For example, TriggerScope [18], IntelliDroid [38], and [10] use symbolic execution to generate external input (*e.g.*, GPS, messages) for malware detection. Crowdroid [11], MAMA [31], DroidAPIMiner [3], DREBIN [6], ICCDetector [39], DroidDetector [42], as well as [13], [19], [23], [32], [43] use feature-based algorithms to detect hidden malicious behaviors in Android apps that effect the OS or servers. Unlike these works that extract their features from system execution context (*e.g.*, ICC, system events, permissions), INPUTSCOPE intends to uncover hidden behaviors are triggered by user input at the Java bytecode level and our detection policy is built upon the execution context of user input validation.

VIII. CONCLUSION

While input validation has been well studied in vulnerability discovery, in this paper we have demonstrated that input validation can also have another important application, namely exposing input-triggered secrets such as backdoors (*e.g.*, secret access keys, master passwords, and secret privileged commands) and blacklists of unwanted items (*e.g.*, censorship keywords, cyber-bulling expressions, and weak passwords). To understand the severity of such input validations in mobile apps at scale, we developed a tool, INPUTSCOPE, to automatically detect both the execution context of user input validation and the content involved in the validation to automatically expose hidden functionality. We have tested INPUTSCOPE with over 150,000 mobile apps and uncovered 12,706 mobile apps containing backdoor secrets and 4,028 mobile apps containing blacklist secrets.

Acknowledgment

This research was supported in part by National Science Foundation (NSF) Awards 1657199, 1834215, and by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762). Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the BMBF and NSF.

References

[1] "OWASP - Input Validation Cheat Sheet," *https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet*.

[2] "Soot - A Java optimization framework." *https://github.com/Sable/soot*.

[3] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[4] M. Alkhalaf, T. Bultan, and J. L. Gallegos, "Verifying client-side input validation functions using string analysis," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 947–957.

[5] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, and B. Saltaformaggio, "The betrayal at cloud city: an empirical analysis of cloud-based mobile backends," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 551–566.

[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," vol. 49, no. 6. ACM New York, NY, USA, 2014, pp. 259–269.

[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008, pp. 387–401.

[9] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "Notamper: automatic blackbox detection of parameter tampering opportunities in web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 607–618.

[10] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Springer, 2008, pp. 65–88.

[11] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011, pp. 15–26.

[12] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards analyzing the input validation vulnerabilities associated with android system services," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 361–370.

[13] P. P. Chan and W.-K. Song, "Static detection of android malware by using permissions and api calls," in *2014 International Conference on Machine Learning and Cybernetics*, vol. 1. IEEE, 2014, pp. 82–87.

[14] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing." in *NDSS*, 2018.

[15] H. Choi, J. Kim, H. Hong, Y. Kim, J. Lee, and D. Han, "Extractocol: Automatic extraction of application-level protocol behaviors for android applications," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 593–594.

[16] A. Ciampa, C. A. Visaggio, and M. Di Penta, "A heuristic-based approach for detecting sql-injection vulnerabilities in web applications," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, 2010, pp. 43–49.

[17] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 989–1003.

[18] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 377–396.

[19] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013, pp. 45–54.

[20] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[21] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "{SUPOR}: Precise and scalable sensitive user input detection for android apps," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 977–992.

[22] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1036–1046.

[23] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *2011 seventh international conference on computational intelligence and security*. IEEE, 2011, pp. 1011–1015.

[24] M. Liu, K. Li, and T. Chen, "Security testing of web applications: a search-based approach for detecting sql injection vulnerabilities," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019, pp. 417–418.

[25] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[26] A. Mendoza and G. Gu, "Mobile application web api reconnaissance: Web-to-mobile inconsistencies & vulnerabilities," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 756–769.

[27] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan, "Mace: Detecting privilege escalation vulnerabilities in web applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 690–701.

[28] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch, "Flowwatcher: Defending against data disclosure vulnerabilities in web applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 603–615.

[29] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 993–1008.

[30] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 209–220.

[31] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. Álvarez Marañón, "Mama: manifest analysis for malware detection in android," *Cybernetics and Systems*, vol. 44, no. 6-7, pp. 469–488, 2013.

[32] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *2009 IEEE International Conference on Communications*. IEEE, 2009, pp. 1–5.

[33] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "Smvhunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps," in *In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14*. Citeseer, 2014.

[34] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," *Acm Sigplan Notices*, vol. 41, no. 1, pp. 372–382, 2006.

[35] N. Viennot, E. Garcia, and J. Nieh, “A measurement study of google play,” in *The 2014 ACM international conference on Measurement and modeling of computer systems*, 2014, pp. 221–233.

[36] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross site scripting prevention with dynamic data tainting and static analysis.” in *NDSS*, vol. 2007, 2007, p. 12.

[37] F. Wei, S. Roy, and X. Ou, “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1329–1341.

[38] M. Y. Wong and D. Lie, “Intellidroid: A targeted input generator for the dynamic analysis of android malware.” in *NDSS*, vol. 16, 2016, pp. 21–24.

[39] K. Xu, Y. Li, and R. H. Deng, “Iccdetector: Icc-based malware detection on android,” *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, 2016.

[40] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, “Intentfuzzer: detecting capability leaks of android applications,” in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, 2014, pp. 531–536.

[41] H. Ye, S. Cheng, L. Zhang, and F. Jiang, “Droidfuzzer: Fuzzing the android apps with intent-filter tag,” in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, 2013, pp. 68–74.

[42] Z. Yuan, Y. Lu, and Y. Xue, “Droiddetector: android malware characterization and detection using deep learning,” *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[43] W. Z. Zarni Aung, “Permission-based android malware detection,” *International Journal of Scientific & Technology Research*, vol. 2, no. 3, pp. 228–234, 2013.

[44] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, “Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications,” in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012, pp. 93–104.

[45] C. Zuo and Z. Lin, “Smartgen: Exposing server urls of mobile apps with selective symbolic execution,” in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 867–876.

[46] C. Zuo, Z. Lin, and Y. Zhang, “Why does your data leak? uncovering the data leakage in cloud from mobile apps,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1296–1310.

[47] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, “Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1469–1483.

[48] C. Zuo, Q. Zhao, and Z. Lin, “Authscope: Towards automatic discovery of vulnerable authorizations in online services,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 799–813.