Hunting Password Leaks in Android Applications

Johannes Feichtner

Institute of Applied Information Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a, 8010 Graz, Austria
`johannes.feichtner@iaik.tugraz.at`

Abstract. A wide range of mobile applications for the Android operating system require users to input sensitive data, such as PINs or passwords. Given the ubiquitous and security-critical role of credentials, it is paramount that programs process secrets responsibly and do not expose them to unrelated parties. Unfortunately, users have no insight into what happens with their data after entrusting it to an application. In this paper, we introduce a new approach to identify and follow the trace of user input right from the point where it enters an application. By using a combination of static slicing in forward and backward direction, we are able to reveal potential data leaks and can pinpoint their origin. To evaluate the applicability of our solution, we conducted a manual and automated inspection of security-related Android applications that process user-entered secrets. We find that 182 out of 509 (36%) applications insecurely store given credentials in files or pass them to a log output.

1 Introduction

A multitude of mobile applications perform security-critical tasks and require that user inputs are processed reliably. For this to achieve, a correct implementation is indispensable, ensuring that no sensitive data can be leaked within the data flow and that cryptographic systems are applied correctly, in case their use is appropriate. Sadly, there is little information on how responsibly applications treat critical user inputs. Usually, it is unknown whether an input undergoes a cryptographic transformation and if it is safe for a user to enter secrets. If the source code of a mobile application is not made available, the correct implementation can only be verified by reverse-engineering the final product.

The analysis of security aspects on mobile platforms has attracted a lot of attention in the past years. A majority of publications in this field focus on the Android ecosystem where the openness of the platform promotes program inspection. Supported by the fact that Dalvik bytecode in Android applications can be decompiled to Java code, existing tools for static analysis are easily applicable [1, 2]. Unfortunately, even state-of-the-art analysis solutions miss the opportunity to trace user inputs statically. Self-contained implementations and specific output formats make it difficult to extend existing tools with new capabilities. In experiments, we also noticed that many solutions are powerful in general but each present different drawbacks when it comes to aiming them at a specific purpose, such as following the trace of user-entered secrets.

In this work, we bridge this gap and present a new static analysis framework that focuses on identifying and tracking user input fields in Android applications. By implementing a combination of forward and backward slicing for use with reverse-engineered Dalvik bytecode, our solution aligns to inspecting the data flow of input fields for sensitive user data. Starting at predefined lookup patterns, the automated analysis first aims to derive concrete slicing criteria. We then follow the data flow throughout an application and obtain all execution paths that influence an input field under consideration. To determine whether user-provided secrets are passed to potentially problematic functions, it is verified whether the encountered statements comply with predefined security checks. After checking several hundred security-related applications, we find that 36% of them process secrets insecurely. Our insights stress the need for a framework to automatically analyze applications regarding the leakage of sensitive user input.

2 Background And Related Work

Most publications in the field of Android application security involve either dynamic or static analysis. Dynamic approaches work by monitoring the live execution of an application after hooking into the Dalvik virtual machine. Resilient to dynamic code loading and code obfuscation, solutions like TaintDroid [3] or Mobile Sandbox [4] can analyze and detect privacy leakage in the current execution path. Nevertheless, they inherently miss code paths that are not visited at runtime. Leaks of password inputs would, thus, only be detectable for input fields where a password was actively provided by the user. The solution of Cox et al. [5] mimics this task and inspects the flow of sensitive data in a sandbox. Other works, also based on TaintDroid, uncover privacy leaks based on used permissions [6] or by enforcing previously elaborated policies [7].

Methods for static analysis, as an alternative, typically apply taint tracking on a reverse-engineered representation of Dalvik bytecode. Smali is a mnemonic language to represent Dalvik bytecode in a parseable format. As it keeps the semantics of code very close to the original, it is often a preferable choice over a more intricate decompilation, e.g. to Java code. Having a source-code like representation, the primary challenge then is to follow arbitrary execution traces as sound and precisely as possible. This objective is tackled both by fully-fledged frameworks, such as FlowDroid [8] and IccTA [9], as well as solutions for individual issues like Implicit Control Flows [10]. Unfortunately, the general design of these tools prevents them from being specifically applicable on input fields for sensitive data. More targeted solutions for similar challenges [11, 12] are tailored to their specific use case and cannot handle the characteristics of both XML resources and dynamically generated input fields. The same applies to the subsequently conducted analysis of potential security-relevant problems where all possible execution paths have to be checked individually.

Tracking the data flow of program statements is commonly referred to as program slicing. The concept can be used to determine all code statements of a program that may affect a value at a specified point of execution (*slicing criterion*). The resulting *program slices* cover all possible execution paths and allow conclusions to be drawn about the functionality of the program. In our work, we adopt the algorithm of Weiser [13] to create slices of Smali code in order to find paths from the origin of an input field to its use in the application code. Technically, our solution for forward slicing is inspired by Hoffmann et al. [14] who proposed a general approach for static backtracking on Smali code.

3 Static Slicing of Smali Code

The ability to trace information in both forward and backward direction is a core component of our framework in order to isolate those parts of an application that are relevant with regard to a specific slicing criterion. In the following, we present the implemented techniques for static slicing and highlight practical challenges.

3.1 Slicing Patterns

The slicing process naturally depends on a slicing criterion referencing a specific line of program code. Considering our objective to track arbitrary input fields matching predefined criteria, a more generic representation is needed. Therefore, we propose so-called slicing patterns that conceptually describe a type of resource or object to track in XML format.

Since a pattern includes no reference to a specific program statement, it does not represent a slicing criterion by itself. Instead, it comprises all necessary data to dynamically build slicing criteria corresponding to the pattern. Assuming that the data of interest occurs multiple times within an application, a multitude of slicing criteria is deduced and subsequently tracked. Depending on the defined focus and level of granularity, a pattern might be applicable to either only one specific application, or be generally suited for a large set of targets. In order to comply with different requirements, patterns support the description of different types or features that can be tracked.

Method Invocations To follow the trace of dynamically generated input fields, we need to be able to address particular method invocations. In the default behavior, slicing criteria are determined by searching for all `invoke` statements matching the given pattern. Therefore, all code lines of an application are scanned, looking for the provided method signature. For each match, the appendant program statement is considered as a starting point for slicing. Subsequently, the name of the register to track is located by associating the index of each occurring register with the given parameter (index) of interest. As a result, a set of suitable slicing criteria is delivered.

Resource Objects Resources in Android applications, such as the user interface, layouts and strings, are usually externalized from the program code. For every outsourced element, the developer has to assign a unique resource ID which can later be referenced in code. Tracking concrete IDs would require us to manually modify the slicing pattern for every inspected application. As a remedy, we propose to address specific resource objects using generalized XPath queries.

As Android resources are typically denoted in XML format, XPath comes in handy to select elements by means of their node type and a variety of predicates. Accordingly, it is feasible to assemble slicing patterns that focus on particular resources in a multitude of applications. In contrast to tracking specific resource IDs, the used XPath queries are intended to cover one or multiple resource elements. By leveraging the flexibility of XPath, queries can be adapted to select arbitrary resource elements that match given properties. In practice, this benefit enables slicing patterns to be generalized to such an extent that the characteristics of individual applications become entirely extraneous.

3.2 Static Slicing

By performing static slicing on Smali code, our framework is capable of determining the control and data flow of relevant code segments in Android applications. Based on a given pattern, an analysis is conducted in forward or backward direction, storing the results in an object-based graph representation.

Having derived one or multiple slicing criteria from a given pattern, they are initially added to an internal FIFO queue. This to-do list serves as input for both the forward and backward slicer and collects all registers, fields, return values, and arrays that are subject to tracking. Moreover, it holds a reference to all objects that have already been followed and excludes them from being reprocessed. When requested by the slicer, the queue returns the next object to track, which includes the register to track and the location of the corresponding opcode. On the basis of this approach we are effectively able to control the slicing process and prevent the repeated analysis of already investigated data flows.

Forward and backward slicing are conceptually separated components that process the input from the to-do list and output slicing results to a dynamically built tree. Initially, the slicing criterion is set as root node, followed by all code statements that are contained in the slice. The generated graph is suited for further analysis, such as security checks for password fields.

3.3 Graph-Based Output

The initial idea consisted in visualizing all data flows in one graph per slicing pattern. Due to the fact that a pattern can lead to multiple slicing criteria, this approach would cause incoherent flows of various criteria to be collated into a single representation. Aside from impeding the meaningfulness of the resulting graph, it would also lead to inconsistent results since overlapping data flows might occur multiple times. As a remedy, one graph is generated per slicing criterion. The top node is always the criterion, deduced from the pattern, since it represents the root of all possible execution paths that can be modeled. Subjacent nodes stand for all code lines which are contained in the slice. In case there are multiple execution paths, e.g. an `if-else` statement, a slice node might have links from multiple predecessor nodes. When code statements are iterated multiple times, e.g. via `for` or `while`, loop cycles are induced between vertices. Each (intermediate) node involves a list of all predecessor nodes, including the originating registers and the registers, related to the current program statement.

A slice tree can comprise one or multiple leaf nodes whereas each describes either a constant or indicates an abruptly ended slicing process. Assuming that a constant value, such as an integer, an array, or a string, is copied into the tracked register, slicing may stop since the register value is redefined. For backward slicing this signifies that the tracking process has led to one or more values that affect the slicing criterion. In contrast, for forward slicing it means that the currently tracked register will not affect any subsequent operation and, thus, the data flow has reached an endpoint. Leaf nodes are also inserted in case slicing loses track. This happens, for instance, when registers are set as parameters in calls to unresolvable methods.

3.4 Slicing Accuracy

The aforementioned queue ascertains accurate analysis results by filtering registers that exceed a predefined threshold of fuzziness. Each tracked register is assigned a *fuzzy level* which indicates its accuracy in accordance with the slicing criterion. In other words, it expresses the likelihood that the value of the currently tracked register still equals the value of the initial register. Accordingly, the fuzzy level is also attached to found constants and nodes within the slice tree in order to highlight their relevance with respect to the slicing criterion. A value of 0 means that the result is completely accurate and has not been modified on its way to the slicing criterion. Higher values indicate less accurate results and a reduced expressiveness of the results.

Although the fuzzy level enables us to measure uncertainty in analysis results, it makes no indications about the quality of found constants. For example, a high value does not necessarily imply that a constant has only marginal impact on a slicing criterion. Similarly, it is probable that a register has a low value but does not correlate with the initial register at all.

4 Passwords on Android

The analysis of data flows from input fields for passwords starts with the definition of a suitable slicing pattern. Based on the provided parameters, concrete password field usages are searched in program code, added to slicing criteria and can then be tracked in forward direction. In view of our analysis objectives, the following case study illustrates the derivation of an eligible pattern. With the intention of tracking any password field occurring in practice, we also identify possible shortcomings of an elaborated pattern.

Basically, password fields in Android applications are either statically defined as XML resources or generated from program code during runtime. Since both options refer to the same implementation internally, their capabilities and produced outputs are identical. As an initial trigger for slicing, however, it is not feasible to cover both forms by a single slicing pattern. This is also reflected by our slicing patterns' types which focus either on resource objects or invocations. In the following, we will examine both cases and highlight their characteristics.

4.1 XML Resources

Password input fields in XML resources typically make use of the element class `EditText` that enables editable input fields to be displayed. Depending on the provided attributes, differently shaped fields and keyboards are presented to the user during interaction. Concise XPath queries facilitate the selection of corresponding input fields for analysis purposes.

Until the release of Android 1.6 (API level 4), the default way to declare password input fields consisted in adding the property `password=true` to an `EditText` element. Although considered deprecated now, the technique can still be found in applications that maintain compatibility with the eldest versions of Android. Referring to the previous section, an XPath statement is suited to specifically match this password input field description. The first-mentioned slicing pattern in Listing 1.1 illustrates the assembled XPath query.

On current versions of Android, password fields are declared by setting a corresponding constant value to the `EditText` element property `inputType`. Alongside with other input types, the change also introduced more fine-grained descriptors for password input fields. For instance, developers can specify the type `numberPassword` in order to restrict possible user input to numerical values only. For the subsequent static slicing process, this implies that the initially tracked value is also numeric and, hence, likely to be subject to integer transformations. If the property `maxLength` is also set, conclusions about the achievable security grade could be drawn even without slicing.

The most obvious descriptor for an arbitrary password combination is the input type value `textPassword`. Considering the previously formulated pattern, the same scheme is applicable to the input type property. The resulting adaptation is depicted in Listing 1.1. In the current state the XPath statement is designed to match *exactly* the given predicate and fail for any deviation. Although it is suited for practical application, the precision is comparably low as other relevant and legitimate input type values are not taken into account. In particular, this concerns all other descriptors, designated for password input, such as `textWebPassword`, `textVisiblePassword`, and `numberPassword`. A possible remedy is to add the listed options to the XPath statement accordingly. The resulting query is now capable of delivering all elements with an exactly matching input type value.

Another possible application scenario is the combined use of multiple input types. For example, the value `textNoSuggestions|textPassword` causes the user-shown keyboard to omit the display of any dictionary-based suggestions. Without adaptation to this circumstance our XPath query would not match input type combinations at all. A pragmatical approach to this issue consists in refining the pattern in a way that it focuses on verifying the occurrence of a password type, disregarding further options. This can be achieved by simply checking whether the property *contains* a known value. In contrast to the previously stipulated exact conformity, we weaken the statement to a containing match. The final slicing pattern is denoted in Listing 1.1. It covers all relevant forms of password types while refraining from matching unrelated values.

Listing 1.1. Forward slicing pattern: Password fields

```
<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
  pattern="//EditText[@password='true']" description="EditText XML fields with attribute
  'password'" />

<forwardtracking-pattern enabled="true" type="XPATH_QUERY" pattern="//EditText[@inputType,
  'textPassword']" description="EditText fields with inputType = textPassword" />

<forwardtracking-pattern enabled="true" type="XPATH_QUERY"
  pattern="//EditText[
    contains(@inputType, 'textPassword') or
    contains(@inputType, 'textWebPassword') or
    contains(@inputType, 'textVisiblePassword') or
    contains(@inputType, 'numberPassword')]"
  description="EditText fields with password inputType" />
```

4.2 Generated Input Fields

Another possibility to display password fields is to generate them dynamically during runtime. Rather than embedding monolithic `EditText` elements in XML resources, editable fields can also be defined using program code. Accordingly, a variety of properties and actions is assignable on each instance of the class `EditText`. A slicing pattern should, hence, be suited to identify generated password fields reliably and to convey slicing criteria for the subsequent tracking process. In order to achieve this, we have to cope with three essential problems:

- How is it possible to distinguish between ordinary `EditText` elements and those that are configured for password input?
- What are the implications of tracking the entire element instead of the password value only?
- Are we able to design a slicing pattern that adapts to the given constraints?

These questions were equally relevant for password fields in XML resources. Nevertheless, in the former case it has shown to be fairly simple to derive a pattern that matches particular properties of one corresponding XML element. With generated input fields, more complex prerequisites apply since password fields cannot be reduced to a single program statement, enclosing all relevant attributes. In the following, we will gradually answer the previously listed questions by examining the sample code provided in Listing 1.2.

Password Field Identification Initialized within the corresponding application context, a dynamically created input field is an instance of the class

`EditText`. In order to hide the user-entered text by asterisks, an input field has to be assigned an appropriate *password transformation method*. Similar to XML resources, an optionally added *input type property* restricts the possible input value to a predefined set of characters and advises the keyboard not to save the password for spelling correction. Although not recommended from a security-aware perspective, specifying the input type may be omitted. Consequently, we can conclude that the only irrevocable indicator for a password field (with asterisks) is the assignment of a `PasswordTransformationMethod` class instance. In order to identify an employed transformation object and input type constant, the arguments of `setTransformationMethod()` and `setInputType()` have to be tracked in backward direction.

With visible (non-hidden) password input fields, an entered text undergoes no transformation and, hence, in that case the value of the input type property remains the sole indicator for a password input field. As illustrated in Listing 1.2, the type is declared by a constant value which first points to the possible user-entered values (e.g. text or number) and secondly specifies the particular type of the input field. Accordingly, for visible passwords the second descriptor would be `TYPE_TEXT_VARIATION_VISIBLE_PASSWORD`. The constant states whether an input field is designed to handle passwords and indicates the processed type.

Being assembled at runtime, it might occur that the input type is not immediately assigned to the `EditText` instance upon initialization. Similarly, it is probable that the transformation method changes during execution. This is likely the case with Android applications that offer users the option to toggle the password visibility by clicking on a button. Internally, this is achieved by switching the transformation method, e.g. from `PasswordTransformationMethod` to `HideReturnsTransformationMethod` (or any other non-hiding option) and vice-versa. Unless the password transformation is already registered upon initialization, it is evident that *all* transformation method assignments to an `EditText` instance need to be backtracked in order to determine whether the element acts as an input for passwords at any point of execution. Of course, this process becomes redundant and can be skipped if an input type is set, already referring to a password or PIN code. Overall, the workflow to find generated password input fields can be summarized as follows:

1. Find instances of `EditText` objects and, using forward slicing, verify whether the methods `setTransformationMethod` and `setInputType` are invoked directly upon initialization.
2. Based on the obtained results, backtrack the arguments passed to the found methods. An input field for passwords is found if at least one of the following conditions is met:
   (a) The tracked transformation method is an instance of the class `PasswordTransformationMethod`.
   (b) The tracked input type constant value indicates a matching field type for a visible, numeric, web, or general password.
3. If still undecided, track all transformation method or input type assignments appendant to a particular `EditText` instance and perform the evaluation as outlined in the previous step.

Listing 1.2. Example of a dynamically generated input field.

```
 1 AlertDialog.Builder alert = new AlertDialog.Builder(context);
 2
 3 final EditText input = new EditText(context);
 4 input.setTransformationMethod(PasswordTransformationMethod.getInstance());
 5 input.setInputType(InputType.TYPE_CLASS_TEXT | InputType.TYPE_TEXT_VARIATION_PASSWORD);
 6
 7 input.addTextChangedListener(new TextWatcher() {
 8   @Override public void onTextChanged(CharSequence s, int st, int before, int ct) {
 9     String password = s.toString();
10   }
11
12   @Override public void beforeTextChanged(CharSequence s, int st, int ct, int af) {}
13
14   @Override public void afterTextChanged(Editable s) {
15     String password = s.toString();
16   }
17 });
18 alert.setView(input);
19
20 Button submitButton = new Button(this);
21 button.setText("Submit credentials");
22 button.setOnClickListener(new View.OnClickListener() {
23   public void onClick(View view) {
24     String password = input.getText().toString();
25   }
26 });
```

Tracking Passwords Having successfully identified an `EditText` element as a container for password input, the subsequent task consists in tracking the data flow of a user-entered password. Beforehand, a suitable slicing criterion is needed in order to trigger this process. In the following, we highlight the available options and point out possible implications on slicing results.

Basically, it is conceivable to compose a criterion from the previously found `EditText` instance and track the object in forward direction. The resulting slice would, in theory, comprise all code statements that refer to the input field or any of its properties. Applied to the sample code provided in Listing 1.2, the result *should* include the code lines 9, 15, 18, and 24 since they reference the input field object or a derivative. However, as opposed to the directly visible data flow from the `EditText` instance to the `AlertDialog` in line 18, the affiliation with the other code lines is not immediately obvious. To resolve these traces, our slicer is aware of *implicit control flows*, internally handled by the Android framework.

As depicted in Listing 1.2, `EditText` objects support the registration of event-triggered methods. They enable a predefined callback to be invoked whenever the event is signaled. The sample code demonstrates this feature by means of the `addTextChangedListener` listener. In practice, it causes the method `onTextChanged` (line 9) to be called with the current input field text wrapped as a `CharSequence`, as soon as the text of the input field changes. Another listener method is attached to a button (line 22) and brings the method `onClick` to access the value of the input field (line 24), once the button is clicked. The actual control and data flow in these two examples is carried out internally and beyond the scope of the underlying program code. For static slicing, this means that neither a consecutive nor a coherent data flow is determinable due to missing links in the execution chain. For instance, without being able to track into Android's `TextWatcher` class, a slicer cannot know that the `CharSequence` encloses the value of the input field. More generally, the slicer will miss all information flows that are handled within a listener-callback system, leading to considerable imprecision and false negatives in the overall output.

One way to address the shown issue consists in statically linking callbacks and their registrations. For instance, assuming that a call to `addTextChangedListener` is encountered by the slicing process, a previously learned mapping could disclose that the actual input value is made available through a `CharSequence` or `Editable` parameter. The downside of this approach, however, is that all probable associations have to be known in advance. Considering the extensive amount of possible listeners and callbacks on the Android ecosystem, a manually managed database is likely to cover only a subset of all implicit control flows.

Instead of tracking `EditText` instances, another approach is to track methods that are known to access the password value. E.g., by defining invocations of `EditText->getText()` as slicing criterion for forward tracking, it can safely be assumed that the initially sliced register holds the actual password value. Employing the same criterion for backward slicing reveals whether the originating `EditText` instance sets an appropriate transformation method or input type. Compared to the formerly described method, this combination of slicing into both directions enables the resulting slice to start with the password value itself (instead of the input field) and ascertains that it is not influenced by unrelated properties of the originating `EditText` object. However, the focus on specific methods, such as `getText()`, also causes other accessors to be excluded a priori.

The following key points can be concluded from the described approaches:

- The slicing criterion has to be assigned an `EditText` element or an access method, such as `getText()`, in order to track password input fields.
- Depending on the initial trigger, the slicing results may include code statements that are not related to the input field value at all.
- User-entered passwords are typically passed to event-triggered callbacks.
- By implicitly referring to an `EditText` instance, password values are made available via different data types and access descriptors. The slicing process has to know these characteristics in advance.

5 Finding Password Leaks

Evaluating the data flow of passwords regarding security aspects is challenging since the severity of problems may depend on the context of an application. For example, it might be inappropriate to flag an application insecure due to the fact that a password does not undergo a cryptographic transformation. Of course, the opposite can be true for applications where cryptography is inevitable in order to protect sensitive data.

Considering passwords as sensitive information, our security rule focuses on general misconceptions that substantially affect its secrecy. For instance, one paradigm states that passwords must not be written to a logging function. This emerges from the fact that the mandatory confidentiality is no longer given as soon as an unintended party is able to learn secret credentials. By analyzing the data flow between a password field and one or multiple endpoints, we aim to answer the following questions:

– Is an entered password written to an output file?
– Is a password leaked to a logging function / logfile?
– Is a cryptographic transformation applied to an input?

If one of the first two conditions is satisfied, the security of an entered password is clearly impaired. The latter question specifically depends on the investigated application. For example, under normal circumstances there is no need for a Mobile Banking application to transform a password in order to login to the service behind. In contrast, a program intending to securely store data protected by a user password undoubtedly should apply cryptography for key derivation and data encipherment.

5.1 Detection Strategy

Using the following workflow, we intend to evaluate the questions listed before:

1. Identify available password fields by applying the patterns, elaborated in Listing 1.1. For each occurrence, track all resource usages in forward direction.
2. From each computed slicing graph, extract all feasible execution paths and evaluate the following conditions:
   (a) Raise an alert if the data flow includes calls to `write(...)` methods of the (sub)classes of `java.io.OutputStream` and `java.io.FileWriter`. Also detect when passwords are exposed using `java.io.PrintWriter`.
   (b) Check if a password is sent to log output or leaked to a logfile using methods of the `android.util.Log` API. Issue a warning if corresponding calls have been found.
   (c) Verify if a password is processed by security-related APIs, exposed in `java.security.*` and `javax.crypto.*`. If found, emit a notification.

The detection workflow starts by obtaining the slicing graphs for all password fields. Initially containing the offset of the password resource, the data flow of an execution path models all program statements that are affected by the input field. Inspecting the graph enables us to search specific accessors that are known to implement the questioned behavior.

6 Evaluation

The goal of this evaluation is twofold. First, we intend to assess the practical feasibility of our analysis solution. Therefore, we manually contrast the output of our framework with the actual source code of real-world applications. This helps us to identify possible weaknesses in our approach and implicitly highlights the framework's reliability. Second, by applying our tool on a larger number of current applications that include password inputs, we gain a valuable insight into the prevalence of potential security problems.

For the evaluation, we conducted both a manual and an automated analysis on the same dataset. In the following, we explain the applied methodology, the individual goals, and what applications were analyzed. Lastly, we combine both approaches into a single representation and point out notable findings.

6.1 Methodology

Before testing the automated analysis, we manually reverse-engineered and examined the source code of 522 applications that use input fields for secrets. All of them were downloaded from the official Google Play Store and had at least 10,000 installations. 206 applications were “password managers”, intended to protect user-entered credentials by means of cryptography. The remaining applications served different purposes: mobile banking (145), cloud storage (68), secure data container (12), messenger functionality (91).

The idea of the manual analysis was primarily to collect a ground truth about what our framework should later find automatically. Meanwhile, we repetitively refined the implementation where we recognized deficiencies and ensured all components would interact well enough with each other. Besides identifying opportunities for future improvement, we also benefited from seeing what our security checks would be able to (not) cover in a real-world scenario.

In the second step, we applied our framework on the dataset. For each automatically inspected application, we obtained a generated report that included all found input fields for secrets, for each of them the possible execution paths and the result of the performed security checks.

6.2 Results

In total, we applied our framework to 522 selected Android applications. As listed in Table 1, among the investigated programs, 10 could not be analyzed automatically as the slicing process was either aborted after the defined threshold of 25 minutes or it surpassed the limit of 80,000 tracked registers. The analysis of another set of three applications failed due to limitations in the amount of usable memory. Precisely, during the pre-processing step, the automated analysis ran out of memory while parsing the Smali code into an object-oriented representation. A manual review of the affected programs revealed that their Dalvik bytecode contained tricks to hamper reverse-engineering. Apart from that, we could verify that these apps process secrets safely. As a result, for 97% or 509 out of 522 applications the analysis workflow terminated successfully.

During our evaluation, we disclosed a total of 2,874 input fields for passwords or PINs. The manual review revealed that the amount of fields used correlates with the program’s category. While, on average, mobile banking applications include 2, messengers provide up to 8 input fields for secrets.

Overall, we found that 41% or 1,181 entered secrets were processed by security-related APIs. Clearly, it depends on the purpose of the individual input field whether a cryptographic transformation is appropriate. However, of 206 inspected password managers, we observed that in 38% or 78 applications none of the

Table 1. Framework evaluation with selected applications

	Count	[%]
Downloaded from Google Play Store	522	
Failure during static slicing	10	2%
Out of memory	3	1%
Analyzable with password inputs	509	97%
Input fields for secrets	2,874	
Secrets passed to crypto-related functions	1,181	41%
Secrets leaked through `android/util/Log`	577	20%
Secrets written to a file output	346	12%
Input fields leaking secrets	923	32%
Apps with unsafe input fields	182	36%

available input fields for secrets was linked to security-related APIs. Although this does not immediately imply security issues in all affected programs, a further inspection seems advisable.

The secrecy of the user-entered data is only preserved if the associated data flows do not allow an attacker to learn credentials. Unfortunately, we found that in 20% or 577 input fields the secret was passed to a log output. Likewise, the input to another 12% or 346 fields was written to files. Interestingly, as also confirmed by the manual analysis, no credentials were leaked both to log output and files. In summary, we observed that 32% or 923 out of 2,874 inspected input fields leaked input data either to files or log output. With regard to the set of 509 investigated applications, it can be subsumed that 36% or 182 are subject to an issue that substantially affects the secrecy of entered passwords.

Evidently, the precision of our analysis results is strongly linked to the accuracy of the inspected data flow graphs. The manual analysis step ensured that there are neither false positives, nor false negatives with regard to our dataset. Nevertheless, from the obtained results we conclude that our solution qualifies for use with an arbitrary dataset. Of course, this does not imply the security checks are complete and there is no more room left for improvement. In fact, additional patterns and checks could be suited to reveal further misconceptions.

7 Conclusion

In this paper, we presented a target-oriented approach to track the data flow of input fields in Android applications by means of static analysis. Based on the proposed concept of slicing patterns and a combination of static slicing in forward and backward direction, our solution excels in following user-provided input right from the point where it enters an application. We assessed our framework by analyzing 509 applications manually and automatically. We detected that 36% or 182 applications leak sensitive user input either to files or log output. This result does not only highlight the viability of our solution but also underlines that misconceived processing of secrets is a common issue in Android applications.

References

1. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L.J., Lam, P., Sundaresan, V.: Soot - a Java bytecode optimization framework. In: Conference of the Centre for Advanced Studies on Collaborative Research – CASCON, IBM (1999) 13
2. Bartel, A., Klein, J., Traon, Y.L., Monperrus, M.: Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot. In: State of the Art in Java Program Analysis – SOAP, ACM (2012) 27–38
3. Enck, W., Gilbert, P., Chun, B., Cox, L.P., Jung, J., McDaniel, P.D., Sheth, A.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Symposium on Operating Systems Design and Implementation – OSDI, USENIX Association (2010) 393–407
4. Spreitzenbarth, M., Freiling, F.C., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into android applications. In: Symposium on Applied Computing – SAC, ACM (2013) 1808–1815
5. Cox, L.P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., Cheemalapati, S.: SpanDex: Secure Password Tracking for Android. In: USENIX Security Symposium, USENIX Association (2014) 481–494
6. Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In: Trust and Trustworthy Computing – TRUST. Volume 7344 of LNCS., Springer (2012) 291–307
7. Mann, C., Starostin, A.: A framework for static detection of privacy leaks in android applications. In: Symposium on Applied Computing – SAC, ACM (2012) 1457–1462
8. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D.: FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Programming Language Design and Implementation – PLDI, ACM (2014) 259–269
9. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.D.: IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In: Conference on Software Engineering – ICSE, IEEE Computer Society (2015) 280–291
10. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In: Network and Distributed System Security Symposium – NDSS, The Internet Society (2015)
11. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Conference on Computer and Communications Security – CCS, ACM (2013) 73–84
12. Backes, M., Bugiel, S., Derr, E., Gerling, S., Hammer, C.: R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In: Asia Conference on Computer and Communications Security – AsiaCCS, ACM (2016) 129–140
13. Weiser, M.: Program Slicing. In: Conference on Software Engineering – ICSE, IEEE Computer Society (1981) 439–449
14. Hoffmann, J., Ussath, M., Holz, T., Spreitzenbarth, M.: Slicing droids: program slicing for smali code. In: Symposium on Applied Computing – SAC, ACM (2013) 1844–1851