Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications

Daniele Gallingani
University of Illinois at Chicago
Chicago, IL
Politecnico di Milano
Milano, Italy
Email: dgalli3@uic.edu

Rigel Gjomemo, V.N. Venkatakrishnan
University of Illinois at Chicago
Chicago, IL
Email: {rgjome1,venkat}@uic.edu

Stefano Zanero
Politecnico di Milano
Milano, Italy
Email: stefano.zanero@polimi.it

***Abstract*—Android's Inter-Component Communication (ICC) mechanism strongly relies on Intent messages. Unfortunately, due to the lack of message origin verification in Intents, implementing security policies based on message sources is hard in practice, and completely relies on the programmer's skill and attention. In this paper, we present a framework for automatically detecting Intent input validation vulnerabilities. We are thus able to highlight component fragments that expose vulnerable resources to possible malicious message senders. Most importantly, we advance the state of the art by developing a method to automatically demonstrate whether the identified vulnerabilities can be exploited or not, adopting a formal approach to automatically produce malicious payloads that can trigger dangerous behavior in vulnerable applications. We therefore eliminate the high rate of false positives common in previously applied methods. We test our methods on a representative sample of applications, and we find that 29 out of 64 tested applications are detected as potentially vulnerable, while 26 out of 29 can be automatically proven to be exploitable. Our experiments demonstrate the lack of exhaustive sanity checks when receiving messages from unknown sources, and stress the underestimation of this problem in real world application development.**

I. Introduction

Android applications are formed by logically separated components that communicate with each other through two message passing mechanisms: *Binder* and *Intents*. *Binder* is a lightweight remote procedure call mechanism, mainly used in service-to-service communication, while *Intents* are the most used inter-component and inter-application communication mechanism. *Intents* are used for data exchange, as well as for requesting the execution of a procedure to another application.

Unfortunately, the Android Intent Passing mechanism does not provide the receiving component with any information concerning the origin of an intent. This facilitates the creation of spoofed intents with malicious input data. If such malicious input is not properly validated or sanitized by an application before being processed, it may subvert its state and control flow in unexpected ways. This attack vector may lead to a wide range of attacks, not only against the application itself, but also against other applications that receive and process data from the vulnerable app.

Previous research works studied applications and the Android ecosystem to identify components that are exposed to receiving intents from untrusted applications [1]. Others studied how applications can circumvent Android's permission checking by delegating execution of operations to applications with elevated permissions [2]. [3] analyzed permission leaks in Android apps in order to identify permission leakage. Finally, CHEX [4] develop static analysis techniques to check whether there exist dataflows that could lead to component hijacking vulnerabilities.

However, a common shortcoming of prior literature is not being able to automatically verify the *practical exploitability* of component hijacking vulnerabilities. For instance, CHEX [4] identifies 254 apps with suspicious data flows. A subsequent manual analysis by the authors, however, identified that 48 out of these 254 apps were false positives. Such false positives are due to two main reasons:

- *Precision issues in static analysis*. Static analysis techniques approximate the behavior of programs. Usually, a sound approximation is sought, by including all possible behaviors. However, to do so, approximations err on the side of excess, including additional behaviors that are not really present, such as dead code (i.e. paths that are never feasibly exercised). Since such additional paths are considered during dataflow analysis, they may lead to false instances of suspicious dataflows.
- *Effect of security-critical actions of code*. Analysis techniques in state-of-the-art approaches to this problem only take into account the existence of potential suspicious paths. They ignore, however, the effect of the code along those paths, such as the use of input validation to mitigate intent spoofing vulnerabilities [1]. Since such techniques can effectively obviate the security issues, ignoring their effectiveness leads to a large number of false alerts.

In this paper, we improve the state-of-art by automatically developing proof-of-concept exploits against applications, to effectively prove that they are vulnerable to intent message vulnerabilities. Developing proof-of-concept exploits helps minimize the risk of false alarms, and thus it increases the usability of the approach.

To do so, we statically analyze the application to identify data-flows under an attacker's (indirect) control. We design an analyzer that is able to follow such flows and identify Intent data that may affect either directly or indirectly the results that a component produces and sends in output. We formulate the problem as an Interprocedural Distributive Environment one, which allows us to efficiently deal with inter-procedural flows. Once such suspicious flows are identified, we develop techniques to analyze the operations (e.g., sanitization) along the identified flows. At this point, we use a constraint solver to develop concrete proof-of-concept exploits, thereby confirming the presence of the vulnerability. Finally, we are able to demonstrate the vulnerability by developing an *attacker app*, capable of launching these exploits on actual applications. It is important to note that, similarly to [4], our approach works on unmodified Android apps, without requiring any special information, nor access to the source code.

We test our approach on 64 popular applications from the Google Play store. Of these, 29 exhibit potential vulnerabilities, and for 26 of these, we are able to automatically generate an exploit, i.e. spoofed intents that trigger and demonstrate those vulnerabilities. We discuss in depth the results of this evaluation and analyze manually the applications to confirm them. We can thus confirm that many popular applications do not implement appropriate security countermeasures for Intent communications. Indeed, from our analysis we observed that most applications only check if malformed Intent payload tuples are received, but such checks are meant to avoid errors, and are thus far from sufficient to stop a determined attacker.

In summary, we make the following novel contributions:

- We provide a static analysis method to automatically detect data flows that potentially allow Intent data to affect the results sent in output by the applications, i.e. to identify potential vulnerabilities (Sections II, III).
- We provide a formulation of this problem as an Interprocedural Distributive Environment problem(Section IV).
- We describe an approach to automatically generate Intent examples that trigger a malicious behavior, thus automatically validating the discovered vulnerabilities (Section IV).
- We develop an *attacker app*, that is capable of delivering malicious exploit inputs to exercise these vulnerabilities (Section IV).

We report our evaluation results in Section V. In Section VI we review related works. Finally, in Section VII we draw our conclusions.

II. Problem Statement

In this section, we provide a running example and illustrate the problem we are exploring.

Threat Model. In our threat model, an attacker first analyzes the manifest file to identify exposed components that can receive intent messages. An example of such a component is depicted in Listing 1, where the *onCreate* method (line 1) is called to start a component. Next, the attacker identifies statements inside those components whose execution may be subverted to the attacker' s advantage. These statements may include network operations, database operations, updates to GUI elements and so on, and their execution may be subverted by modifying their parameter values, e.g., URL-s where data are sent by network operations, database queries, and the text of GUI elements. One such statement, dealing with a network operation is the HttpGet object creation in line 19. If the attacker is able to modify the value of the *url* variable to a domain under the attacker's control or the value of the *httpPar* variable to a string containing a cross-site scripting attack, then the component can be used to leak data to the attacker's web site or to execute a cross-site scripting attack, respectively. Another statement that may be of interest to an attacker is the one on line 31, which sets the path of a file that is ultimately sent over the network. If the attacker is able to modify the value of the variable *p* (for instance, by including ".." in the path to perform directory traversal) together with the variable *url*, then the component can be used to send arbitrary files to a host under the attacker's control. In the rest of this paper, we call such statements targeted by an attacker *sink* statements.

Under this threat model, the only way in which the attacker can try to modify the parameters of the sink statements is by sending a specially crafted intent to the component via a malicious application installed on the user's phone. Thus, if the component is exposed, the attacker can control the values that that component receives in input (e.g., lines 2-5).

This attack vector has been recognized in the past by researchers and developers alike [1], [4]–[6]. Two recommended practices for limiting this type of attacks are to not expose components needlessly and to sanitize and validate the data in input to the components. However, as often happens in application development, secure coding practices are not always followed. In fact, as shown by two independent studies of the first practice, a large percentage of applications still exposes components needlessly [1], [7]. In addition, as shown by other studies, there may exist execution paths from (exposed) source statements to sensitive sinks [4], along which data may flow. However, the presence of a path does not necessarily imply that an attack is feasible. In particular, applications may perform several operations along that path, such as sanitizations and other business logic operations.

Listing 1: Source code of a vulnerable application

```
1  void onCreate(Bundle savedInstance) {
2   Intent intent=getIntent();
3   String host = intent.getStringExtra("hostname");
4   String user = intent.getStringExtra("username");
5   String file = intent.getStringExtra("filename");
6   String url="http://www.example.com";
7   if (host.contains("example.com"))
8     url = "http://" + host + "/";
9   if (file.contains(".."))
10    file = file.replace("..", "");
11  String userId = getUserID(user);
12  if (userId != null)
13    textView.setText(user);
14  String b64File = toBase64(file);
15  String httpPar = toHttpParams(b64File,user_id);
16  . . .
17  try {
18    DefaultHttpClient httpC = new DefaultHttpClient();
19    HttpGet get = new HttpGet(url+httPar);
20    . . .
21    httpC.execute(get);
22  }
23  catch(IOException e) {
24    e.printStackTrace();
25  }
26 }
27 String toBase64(String p) {
28  if(p=null || p.equals(""))
29    p = "/data/data/com.example/defaultFile.pdf";
30  else
31    p = "/data/data/com.example/public/" + p;
32    byte[] bytes = InputStream.read(p);
33    String b = Base64Encoder.toString(bytes);
34    return b;
35  }
```

In our example, we delineate three different operations that deal with the input variables. The first operation is a sanitization (lines 7-8) of the variable $host$. This sanitization is not sufficient, since an attacker can use any host name that contains the "example.com" string, e.g., "malicious.example.com" or "example.com.com". The value of *host* set by the attacker inside the intent therefore flows unmodified to the sink statement at line 19. The second operation is also a sanitization (lines 9-10), which removes eventual ".." substrings from the file name to prevent directory traversal attacks. This sanitized value is then used later in line 31, which also adds a predefined prefix to the path. Thus, even though an attacker may be able to control to a certain degree the value of the variable *p* at the sink statement (line 32), he cannot use it to perform a directory traversal attack. The third operation (lines 11-13) checks that the *user* exists and sets a GUI element to a default user name if it does not exist. After this point, the value of the variable *user* is not used by the program anymore.

As has been recognized by previous work, to detect this type of vulnerabilities, it is important to correctly identify paths that starting from the *source* statements enable an attacker to influence the variable values at the *sink* statements. However, the existence of a path does not imply that an attack is feasible. To precisely identify exploit opportunities and prevent them, the operations performed on the variable values along that path must also be considered. In fact, these operations may include sanitizations (e.g., lines 9-10), and other business logic operations (e.g., lines 11-13) that, while allowing an attacker to influence the values at the sink statements, make exploits unfeasible. An approach that includes these path operations in its analysis is therefore needed to precisely identify exploit opportunities and prevent them. Such an approach must also provide a vulnerability proof under the form of malicious input to the component, in order to verify the vulnerability. In the rest of the paper, we present a method for automatically detecting such vulnerabilities and providing proofs for them.

III. Approach

In this section, we provide an overview of our approach for automatically generating exploits as proofs of application vulnerabilities.

A. Problem Formulation and Approach Overview

We formulate the problem as follows. Let the *state* of an application at a particular point in the program be defined as a set of (variable, value) pairs $V = \{(v_1, a_1), (v_2, a_2), ..., (v_n, a_n)\}$ visible at that point during execution. To successfully launch an exploit on a specific *sink*, an attacker needs to induce a state of the attacker's choosing at that sink. We denote this state by *exploit state* and represent it with a set of (variable, value) pairs $V_E = \{(v_{e1}, b_1)(v_{e2}, b_2)..., (v_{em}, b_m)\}$, where the variables v_{ei} represent the parameters of the sink statement. Furthermore, to be able to induce state V_E at the sink, the attacker can only use the partial control over the program input state at the *source* statements defined as a set of (variable, value) pairs $V_I = \{(v_{i1}, c_1), (v_{i2}, c_2)..., (v_{in}, c_n)\}$. Therefore, from an attacker's perspective, the problem can be stated as follows: can he determine a state V_I at one or more *source* statements, which induces a state V_E at a targeted sink statement? For instance, if an attacker aims at inducing the state $V_E = \{url, \text{“}http : //malicious.example.com''\}$ in line 19 of Figure 1, what are the values of the variable $host$ (line 3) that (s)he must induce in input via sending an intent?

Therefore, we state the problem as follows. Given any potential sink p in the program and given an exploit state V_E in that point in the program, can we automatically determine if there exists a state V_I at the *source* statements that induces V_E in p when the program executes? We highlight at this point that V_E is determined by an attacker depending on the type of attack s(he) plans to carry out and on the particular sinks. That is, V_E may represent a general pattern of attacks on particular sinks present in many applications or a single state for a specific application. For instance, if the attacker decides to target commonly used database APIs as sinks with the goal of carrying out SQL injection attacks on the applications that use those APIs, the state V_E will contain the variables that appear as parameters of those APIs with values containing SQL injection attack patterns (e.g., 'or 1=1 #). If, on the other hand, the attacker decides to target UI components for a phishing attack on a specific application, V_E will contain variables whose values determine the look and feel of that application.

Therefore, in our approach, we take the state V_E and related sinks as input and try to automatically determine if one or more related states V_I exist. Once V_I is determined, it then serves as a proof of vulnerability for V_E. To answer these questions, the relationship between the application state at any point in the program and the application state V_I at the source statements must be made explicit. The discovery of such relationship and its modeling as a function F, such that we can automatically compute V_I as $V_I = F(V_E)$ is at the core of our approach.

At a high level, the steps of our approach are as follows:

1) **Path computation**. Considering that every program point p may be a sink statement, the paths between the *source* statements and every point in the program are computed using a combination of taint propagation and static data flow analysis.
2) **Symbolic execution**. After the path computation step, symbolic execution over the paths is performed to derive the set of constraints imposed over the variable values along those paths. At the end of this step, for every program point p, a logic formula F_p is created, whose variables correspond to program variables and whose terms correspond to the program statements that modify those variables. Thus, F_p represents the relationship between the input state V_I and the application state in p.
3) **Exploit generation**. Given a point p in the program with the corresponding formula F_p, and an arbitrary exploit state V_E for that point, a new formula is created as $F = F_p \wedge F_E$, where F_E is a formula representing V_E. Next, the formula F is translated into a form suitable for an off the shelf solver

and solved for the variables of the state V_I.

In the exploit generation step, the formula F_p, which represents the relationship between the state V_I and the program state in point p, is joined with the constraints over the variable values desired by an attacker at the point p. Thus, if a solution exists, it must satisfy both sets of constraints. In the rest of this section, we describe each of these steps and their challenges in more detail.

B. Path Computation

Path computation deals with the identification of all the possible execution paths from a source statement to any program point p. This task can be modeled as a data-flow problem where path information is collected inside facts associated with each point in the program. However, in the context of Android, such data-flow analysis must account for the peculiarities of the Android environment, described next.

Interprocedural data-flow analysis. Android applications written in Java use method calls heavily. Therefore, any analysis framework must provide strong support for interprocedural data-flow analysis. Providing such support may be challenging, especially when identifying paths through deep sequences of method calls as well as recursive method calls. An additional challenge is posed by calls to library methods, which can largely increase the amount of code to analyze. In addition, in Android this problem is further exacerbated by the presence of native method calls over JNI with the control flow involving native code.

To deal with these challenges, we first divide the methods into two sets: user-defined and libraries. Next, a control-flow supergraph of the user-defined methods is created by the data-flow analysis framework. In this supergraph, the call sites are joined with callee definitions and callees' exit sites are joined back with the call sites. We show a portion of the supergraph built for our example in Figure 1, where each node corresponds to a statement and is labeled with the line number from code listing 1.

The supergraph provides a uniform representation of the control flow across different methods and is therefore well suited for interprocedural analysis. In addition, library methods are represented as single nodes inside it, that is their body is not included in the supergraph. This choice, while potentially reducing the precision of the approach, considerably reduces the size of the supergraph and provides clear performance advantages. To limit this imprecision, we summarize the operations of the most commonly used library methods (i.e., string manipulations) with a library of constraints (described later in this section), which can be used in the formula F_p.

Path explosion. Path explosion is a common problem when performing data-flow analysis that additionally becomes even more acute in an interprocedural context. To further limit the size of the supergraph and reduce the number of paths, we perform a preliminary *taint propagation* step, which identifies the set of statements that use attacker-provided values or values derived by them and the set of statements whose execution is independent from an attacker. We include only the former set in the subsequent analysis.

In Figure 1, the nodes in green background represent statements that use tainted variables, while those in white background represent the rest of the statements. The source statements are represented in blue background. We note that for space reasons, not every statement is shown in Figure 1.

Android application model. An additional challenge is posed by the asynchronous nature of the Android framework and its event-guided application execution, including various callbacks. Common events are, for example, component stopping or destruction based on user-events or system-wide events, such as low memory or battery power. This execution model poses challenges in the determination of all the possible paths through which data may flow, by breaking some paths into different methods, which are not connected in the supergraph.

In our approach, we consider these methods independently: as described in detail in Section IV we consider each statement that extracts data from an intent as a separate *source* statement and do not follow possible paths across methods that cannot be linked in the supergraph. While this choice may miss some paths, we note that a large number of these event-based method calls deals with saving the state of a component and restoring it when the component is resumed. Thus, in these cases there is no loss of information about paths and variable values.

After the preliminary step of taint propagation, the data flow analysis framework proceeds to traverse the supergraph and to collect for every point in the program the paths reaching that point from the source. The supergraph traversal starts from the source statements and adds statements to each list until a fix point is reached, and no additional statements are added to any of the lists of any program point. During this traversal, the nodes that do not use tainted variables are ignored.

C. Symbolic execution

Once the data-flow analysis framework collects all the paths for every program point, symbolic execution is used to create a formula of constraints for that point. In particular, for every program point, the list of statements is consulted and every statement different from a branching statement is added as a constraint to a symbolic formula. At the end of this step, for each program point we obtain the symbolic formula F_p, which models the relationship between V_I and its state in that program point.

$$\begin{array}{lcl}
F_p & \rightarrow & F_p \vee conj \mid conj \\
conj & \rightarrow & (conj \wedge term) \mid term \\
term & \rightarrow & stat \mid \neg stat \\
stat & \rightarrow & statement \mid var == statement \\
statement & \rightarrow & statement + single\_stat \mid single\_stat \\
single\_stat & \rightarrow & var \mid constant \mid lib\_method \\
lib\_method & \rightarrow & solv\_stat \mid nonsolv\_stat
\end{array}$$

Fig. 2: Grammar of Symbolic Formula

The syntax of the symbolic formulas used in our approach is described by the grammar listed in Figure 2. In particular, the symbol $solv\_stat$ represents statements and library methods whose operations semantics can be modeled by the solver used in the exploit generation step (see Section IV-D). These include string manipulation methods (e.g.,

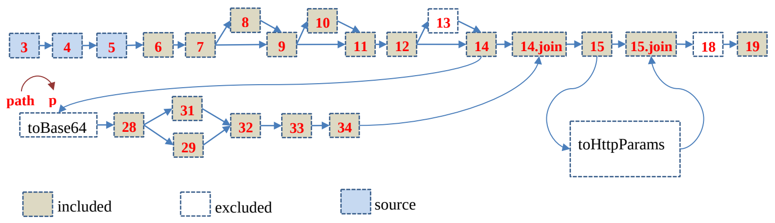


Fig. 1: Supergraph built by the analysis framework.

host.contains("example.com")). The symbol $nonsolv\_stat$ represents statements and library methods whose semantics cannot be modeled by the solver. Finally, var represents tainted variables, $constant$ represents constant strings, while the "+" symbol represents string concatenation. For instance, the formula F_p related to the sink statement on line 20 of Figure 1 is derived as:

$$(host.contains(\text{"}example.com\text{"}) \wedge url{==}\text{"}http://\text{"} + host + \text{"}/\text{"})\bigvee$$
$$(!\ host.contains(\text{"}example.com\text{"}) \wedge url{==}\text{"}http://www.example.com/\text{"})$$

We note that each term in the symbolic formula represents a statement along the path, while each new path created by a branching statements is represented using a disjunction. Assignment operations in the code are modeled using equality constraints, in order to capture the equality conditions between two expressions.

Loops. One of the main issues in symbolic execution is dealing with loops. In fact, even a single loop can generate a large number of execution paths depending on different number of iterations and paths inside the loop. Since the number of loop iterations is unknown during static analysis, a common strategy is to place an upper bound over the number of times a loop is executed symbolically. In our approach, we execute each loop symbolically one time. This choice allows us to cover the loop body statements while still having an acceptable performance.

D. Exploit Generation

The input to the exploit generation step, is a program point p, the corresponding formula F_p and a set of assignments representing the exploit state V_E. The first operation of this step is the translation of the symbolic formula F_p into the solver's language. In particular, for each member of the $solv\_stat$ statements, we create a set of constraints in the language of the solver, which model the behavior of that statement (for more details about this task, see Section IV). The members of the $unsolv\_stat$ statements are modeled with a particular operator in the solver's language that returns the whole domain of values for the variable.

IV. Implementation

In this section, we discuss the implementation of the different steps of our approach.

A. Implementation Background

In this subsection, we provide a short background on two techniques and tools that we used in our implementation, namely the IFDS framework and Kaluza.

IFDS framework. IFDS is a framework for interprocedural data flow analysis that transforms dataflow problems into graph reachability problems [8], [9]. This framework is particularly efficient in dealing with interprocedural data flow analysis, and highly customizable to represent different data flow problems. The framework takes care automatically of several general analysis tasks, such as determination of valid paths on the control flow supergraph (i.e., paths that can potentially be executed at runtime) and of fact propagation. However, to solve a specific analysis problem, it is necessary to formulate it appropriately as an instance of an IFDS problem. In practice, this means defining the analysis information contained in the data-flow *facts* as well as the *rules* that update that information for every node in the control flow supergraph.

Kaluza. Kaluza is an efficient solver for formulas containing string variables and constraints in the form of string equalities, substring operations, numeric constraints over string lengths, and so on [10], [11]. Kaluza natively supports a set of string operations, such as string concatenation, equality, and length equality.

B. Path Computation

As mentioned previously, the problem of path computation is a typical interprocedural data-flow problem. In our approach, we model this problem as an instance of an interprocedural, finite, distributive, subsets problem (IFDS). This framework is build on top of Soot, a Java optimization framework, due to the many analysis facilities it provides [9], [12], [13]. In addition, we use the Heros Soot plugin, which provides a fully context-sensitive implementation of the IFDS framework [12].

To prepare the application for use with Soot and HEROS, the Dalvik bytecode is first transformed into Soot's intermediate representation language Jimple. While in theory this transformation may be lossy and not retrieve the original code, these losses are negligible. Jimple is particularly suited to our task, since it provides a three-address and single assignment representation of the code, making it easier to derive the information about paths and perform symbolic execution. In addition, the Soot framework provides many ready-to-use capabilities for code analysis.

In our implementation, we model the preliminary taint propagation step as a data-flow problem as well, and incorporate it into the IFDS instance problem of path computation. This removes the need for running this step separately and improves the efficiency of our implementation. Before proceeding further, we provide a short review of the IFDS framework.

Source statements detection. The first step in defining the problem as an IFDS instance is the specification of the *source statements*, which constitute the IFDS analysis entry points. To detect these entry points, a full scan of the Jimple representation of the program is performed and the instructions that perform Intents and Bundle payload extraction (e.g. $getStringExtra$) are identified. Since the set of API calls that Android provides to extract Intent payloads is limited, we use an exhaustive list of method signatures for this task. A step further is made in order to reconcile extractions of different payload pieces conceptually belonging to the same Intent message. After this detection, the program variables whose value is defined in the entry points serve as the initial taint variables. These are indeed the variables appearing in the state V_I, whose value can be set by an attacker.

Path Computation and Taint Propagation. The next step in modeling our problem as an IFDS instance, is to define the information contained inside the dataflow facts and how this information is updated for the different nodes of the supergraph. In our implementation, we use a special definition of a *fact*, which contains both taint propagation and path information. Thus, a *fact* is defined as tuple $F = (input\_vars, tainted\_vars, statements, cond\_statments)$, where $input\_vars$ represents variables from the state V_I whose values is defined in the source statements (namely predefined Android API statements that extract values from intents), $tainted\_vars$ is a set of variables representing the tainted variables, $statements$, is the list of statements from the source statement to the current point in the program, and $cond\_statements$, is the list of all the conditional statements from the source statement to the current point in the program.

Thus, for every program point, the fact associated with that point contains the list of input variables, the list of tainted variables visible in that point, as well as a list of statements contained in the paths from the source statements to that program point. The taint is therefore propagated to a variable by adding that variable to the list.

During analysis, the IFDS framework takes care of traversing the supergraph and update the facts associated with each node using user-defined *rules*. These rules are different for every node and described below.

- *Normal Rules:* These functions define how fact information is updated for nodes different from method calls. In this case, we add a statement to the $statements$ list if: either the $input\_vars$ or one of the $tainted$ variables is used by the statement. A new variable is added to the $tainted$ set if its value is obtained by using one of the $input\_vars$ or a tainted variable.
- *Call Rules:* These rules define how fact information is updated for procedure calls. In this case, the call statement is added to the $statements$ list, if the $input\_var$ or one of the tainted variables are contained in the list of arguments. In addition, this rule is used to add to the set of tainted variables the formal parameters of the callee that correspond to tainted variables in the call.
- *Return Rules:* The purpose of a return rule is to propagate the information discovered inside the body of a called method to the caller. Using this rule, the taint information is therefore propagated to the variables at the caller site. For instance, since the return value of the method $toBase64$ is tainted, the variable $b64File$ is added to the list of tainted variables.

C. Symbolic Execution Implementation

Given the paths identified inside the facts by the previous step, we match each statement to one of the productions of the symbolic formula grammar described in Section III and add it as a term to the rest of the formula. In particular, the rules for each of the productions are described below. In this step, we also perform variable name rewriting, to $flatten$ the objects and to extract constraints among $String$ variables. This renaming is performed by prepending to the name of a variable, the name of the method it is declared in and the name of the corresponding object (if available). For instance, the variable p inside the function $toBase64$ is renamed to $toBase64\_p$.

- *Assignment statements:* Every assignment statement in the path is transformed into an equality constraint.
- *Branching statements:* For every branching statement, symbolic execution is split into two paths and the condition of the branching statement is added to the true branch, while its negation is added to the false branch. For instance, when the if statement in line 28 of Fig. 1 is encountered, the condition $toBase64\_p == null \| toBase64\_p ==$ "$''$ is added to the formula representing the path that contains the $then$ part of the statement, while the condition $!(toBase64\_p == null \| toBase64\_p ==$ "$'')$ is added to the path that contains the $else$ part of the statement.
- *User-defined procedure calls:* When a call to a user defined method is encountered, we first rename the local variables of the method by prepending the name of the function to avoid duplicates, then add equality constraints between arguments and formal parameters. Next, we proceed to symbolically execute the function. For instance, since the variable $file$ is passed as an argument to $toBase64$, we add the constraint "$file == toBase64\_p''$ to the formula. If the function returns a value that is assigned to a variable, we add an equality condition between them as well. For instance, the condition added after the return statement of $toBase64$ is $b64File == toBase64_b$.
- *Library method calls:* Unsolvable library method calls are replaced by a special term whose purpose is to not introduce any constraints to its arguments.

D. Exploit Proof Generation

As mentioned previously, to generate an exploit as a malicious state input state V_I, we chose to use the Kaluza constraint solver. Kaluza natively supports several string operations. For other operations, not natively supported by Kaluza, a translation system for a set of Java (Jimple) standard library methods was built, which focuses on string and integer constrains. This set together with the set of operations natively supported implements the *solvable* library methods previously discussed. Some examples of these custom translations are listed in Table I, using regular definitions.

.

For library methods that can not be translated directly in Kaluza (e.g., Base64Encoder.toString(bytes)), a report is

Java	**Kaluza formulation**
$a.contains("test")$	$a \in CB(/.*test.*/, 0);$
$a.indexOf("test")$	$a := T1.T2;$ $0x0 == Len(T1);$ $T2 := T3.T4;$ $T3 := T5.T6;$ $T6 == "test";$ $T5 \notin CB(/test/, 0);$ $a\_indexOf == Len(T5);$
$new\_a = a.replace("test",$ $"newTest")$	$a := T1.T2;$ $T2 := T3.T4;$ $T3 := T5.T6;$ $T5 \in CB(/test/, 0);$ $T7 \in CB(/newTest/, 0);$ $T8 := T9.T4;$ $T9 := T7.T6;$ $new\_a := T1.T8;$

TABLE I: Kaluza constraints formulation example (CB = CapturedBrack)

created in output, and either a Kaluza approximation of their functionality is manually built or they are represented by a special term that places no constraints on the values of their variables. An example of such approximation is the $split$ method, which is a utility function to divide a string into pieces separated by a substring given in input to the function. $Split$ returns an array, and arrays are hard to represent in Kaluza because they are defined as an unknown number of variables, while Kaluza accepts only defined numbers of variables. Our approximation consists in producing the entire string instead of an array of parts as returned value of the method.

After the constraint solver processes the translated formula, it provides a set of solutions for the ranges of variables in V_I for which the formula $F_p \wedge V_E$ is satisfiable, where V_E is also expressed using the Kaluza language, which provides the opportunity to define several patterns for the values of the parameters at the sinks. Conversely, an unsatisfiable result produced by the solver means that the vulnerable sink cannot be reached, in practice, at run-time. For instance, the following example shows a Kaluza formula derived from one of the studied applications. In this example, after ensuring that the input value $source$ length is greater than 0, the prefix is stripped out (if any) in tmp. This tmp variable is then matched against a regular expression for validation purposes. Our tool was able to traverse the control flow graph with the Jimple representation of this code fragment, and in the end obtain the solution 4444944944.

```
$source.length > 0
$IF(source.startsWith("+1")) { $tmp := $source.
      substring(2, $source.length - 1) }
IF((not $source.startsWith("+1"))) { $tmp := $source
      }
$tmp.match("/([2-9][0-8][0-9])\ *([2-9][0-9][0-9])\
      *([0-9][0-9][0-9][0-9])/")
```

E. Approximations and limitations

For simplicity, we used several approximations in our approach. We discuss their impact in the following paragraphs.

Untainted input variables. Motivated by efficiency considerations, we chose to ignore variables whose values cannot be affected by the variables input via intents. While allowing us to prune a portion of the control flow super-graph by removing statements that do not use tainted variables, this choice may also reduce the precision of our approach. In fact, the value of the outputs at the sink may also depend on these variables. The overall result of this choice is that the untainted variables may appear in constraints containing tainted variables. For instance, if a statement $x = s + t$; appears in the code, with s untainted and t tainted, the constraint $x == s + t$ will be added by the symbolic execution to the symbolic formula. However, since the solver has no constraints related to s, it assumes that s can have any possible value.

This approximation may lead to *false positives* where an input state V_I is computed statically starting from an exploit state V_E, while at run time, the presence of these untainted variables in the computation path may induce a state different from V_E at the *sink* statement.

Attack Effectiveness. The effectiveness of this attack depends partly on the Android communication system, on the installed apps in the device, as well as on user attention. In particular, when an intent is sent, if several activities have registered to as receivers for that intent type, Android will present a list of choices to the user. Thus, if malicious intents are caught in this way, the attack may fail. However, we note that this type of behavior may be bypassed by an attacker by sending explicit intents, where the receiving component is explicitly named.

F. Attack app construction

Having generated the exploit automatically, we take a step further, and generate an attack application that exploits the vulnerable application. An effective attack is automatically prepared in the form of a well-crafted application able to send Intent messages to the right target, at the right moment in time and carrying the malicious payload. It is entirely possible to present such attack application as a simple utility application (such as a torch), which runs a malicious *service* behind the scene.

Our malicious service embeds the exploit strings obtained from the previous steps in a list of pre-populated Intent messages ready to be sent, one for each demonstrated vulnerability. In order for the attack to be successful, we enhance the service logic and domain knowledge to obtain the best attack scenario possible: the user perception of the environment during the attack must be as transparent as possible.

For each of the messages, the service first needs to check whether the target vulnerable application is installed on the system or not. We do this through the Android *Package Manager API* offering the $getInstalledApplications$ method. The invocation with the $PackageManager.GET\_META\_DATA$ specified as argument returns a complete list of all the applications installed on the phone. The list contains several meta-data such as the application's package name or the application's launch Intent message.

In the next step of the attack the application checks if the application is currently active (in foreground) on the device. This is particularly important for two reasons: first we limit the context switching consequences of presenting to the user a completed unrelated content (e.g. a different application screen), then we can assume that the user uses the vulnerable application and hence that the user is logged in at the moment of the attack. Also in this case, Androids offers an API: the *Activity Manager API*, in particular the $getRunningAppProcesses$ method that returns a list of $RunningAppProcessInfo$ containing the foreground/background information along with the packages Application process package information, that are used to identify the application. When it ensures that a victim application is installed and running, the attacker app easily sends the malicious Intent message to trigger the attack.

V. Results

A. Experimental Setup

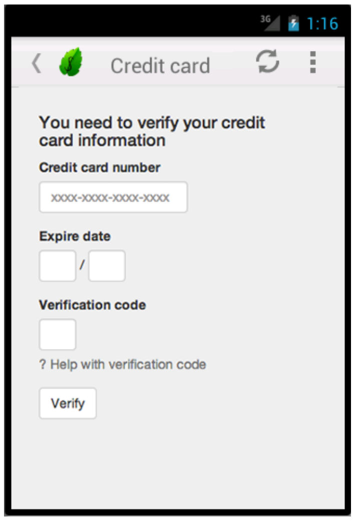
We evaluate our approach on 64 free applications of different sizes, picked from different categories of the Google Play Store. We targeted a class of easily verifiable vulnerabilities, which includes manipulation of UI components, which could lead to phishing attacks. For each application, we targeted the code of the UI components and determined the possible values for the state V_E in those components. Verification of different vulnerabilities is also possible. For instance, for SQL injection vulnerabilities, the state V_E can be set to contain a malicious query and the results of the query may be checked after the attack. For cross site scripting attacks targeting sinks with network operations, V_E may be set to contain cross site scripting code and the effects can be checked on the remote server.

The evaluation was performed on a quad-core machine equipped with 16 GB of RAM. The need of a large amount of memory was driven by the application call graph generation performed by Soot. The memory requirements during the IFDS run and super-graph construction varied depending on the code size, peaking at approximately 5 GB.

B. Experimental Results

The evaluation detected paths from sources to sinks in 29 of the 64 applications. Out of these 29 applications, the string solver was able to produce V_I exploit strings for 26 of them. These exploits were manually tested and confirmed. We divide these vulnerabilities according to the class of UI elements that can be targeted by an attacker:

- *Entire screen:* vulnerabilities in which an attacker could change the content of the main portion of application screens, keeping visible parts of UI that identify the application such as the action bar. Applications such as Mint, which load arbitrary web pages or Booking that let populate a text area belong into this category.
- *Alert screen:* vulnerabilities involving pieces of code used that populate application-specific alert screens. Applications such as Poste Italiane or Poste Pay can be induced to prompt to users arbitrary text content in UI areas usually assigned to communicate operations status.
- *User Inputs:* applications filling user inputs, such as text fields, with Intent payload pieces. An example could be found in GoSMS where the SMS creation form (recipient and text body) can be arbitrary populated.
- *Other components:* we grouped in this category all the vulnerabilities that involve the manipulation of minor interface components such as search input fields, screen titles or buttons. Vulnerabilities found in applications such as Craiglist belongs to this category.



You need to verify your credit card information

Fig. 3: Example of phishing attack on the Mint app.

The results of the analysis are summarized in Table II, Table III, Table IV and Table V. For each of the applications found to be vulnerable, we describe the related vulnerability and the implications of a possible attack.

C. Detailed description of attacks

In this section, we describe in detail four attack scenarios chosen from the vulnerable findings in the sample set.

Mint aggregates and presents to users detailed reports of all their incomes and expenses from different financial sources such as credit cards and bank accounts. Our tool reported a *entire screen* vulnerability in Mint. The UI integrity attack consists in showing to the user an arbitrary web page inside the application visual context. A phishing attack scenario consists in presenting to the user a crafted web page resembling Mint's look and feel inside the application. In the page, the user is asked to reinsert her credit card details complete with verification code because of identity verification purposes. Reasonably, the user would act as asked, since they already inserted such information in the Mint system. The loaded web page is now free to send the collected credit card information to any server. It is also important to point out that the URL of the currently visualized page is never presented to the user. An example of this attack can be seen in Figure 3.

Poste Pay manages the popular Poste Pay rechargeable debit card service, allowing users to recharge prepaid cell phones and any other Poste Pay card. Our tool reported a *alert screen* vulnerability in Poste Pay. Since the content of an alert screen is completely specified by an attacker, one could mimic traditional phishing attacks, prompting users to submit

TABLE II: Experimental Results - Entire screen

Application	Implication
Booking	*Important Information* Activity screen content can be filled with text
IM+	Display an arbitrary web page inside an Activity && Change the Activity title. An attacker can thus inject arbitrary conversation threads
Mint	Display an arbitrary web page inside an Activity
PromoQui	Induced an Activity to load an arbitrary web page and to change the common purchase process
SwissQuote	Populate an Activity with arbitrary text content

TABLE III: Experimental Results - User Input

Application	Implication
GoSMS	Prompt to the user notification about a new message received. Can set an arbitrary sender and SMS content
Imo	Populate registration screen by inserting custom fields such as email, password
Yelp	Modify the fields contained in the venue review draft screen. A successful attack leads in a random review by the user to a specific venue.

TABLE IV: Experimental Results - Alert screen

Application	Implication
Poste Italiane	Modify and show the application prompt screen usually used by the application show notifications to users
Poste Pay	Modify and show the application prompt screen usually used by the application show notifications to users

TABLE V: Experimental Results - Other components

Application	Implication
Craiglist	Change the Action Bar title, compromising the interface integrity
Hollister	Fill the search box used to search between book contents
OpenTable	Populate the editable text field used for restaurant searching
SnapChat	Arbitrary set the label of some buttons in the preference screen

their information to a malicious website or email address. Another scenario could ask the user to make a small transfer to a specific credit card in order to, for example, extend the expiration date of the card.

Booking is a hotel reservation system, allowing to browse hotels by geographical position, price and user rating and book them with the help of a credit card. Our tool reported a *entire screen* vulnerability in Booking. The application allows attackers to fill arbitrary text in the “Important Information” screen. An attack aimed to steal the reservation amount can be designed as follows: The user is informed of a (fake) server issue experienced by Booking itself, and told to proceed with a manual transaction to fake bank coordinates, or on an alternate (malicious) website.

GOSMS is a popular SMS replacement client for Android for which our tool reported a *user input* vulnerability, which can be used to fill the SMS creation screen with arbitrary recipient and message body. Exploiting this vulnerability, for instance, pre-populating the user inputs with a fake demand for charity as SMS body one can set up an effective SMS fraud.

D. Other experimental results

In Table VI we summarize several performance results, including information about analysis execution time, path lengths and components in the applications, and so on. As it can be noted, the execution time of our whole analysis depends on the number of components of an application. The per-application execution time range is very wide, varying from a minimum of 2.4 minutes for 3 analyzed components to a maximum of 33.3 minutes spent to analyze an application including 31 components, with an average of 12.3 minutes per application. The execution time is heavily dominated by the IFDS analysis which consists in 70-80% of the overall execution time.

We discovered 92 paths leading to a sink out of 537 analyzed with an average of 4.2 vulnerable paths per application. For each vulnerable path the analyzer collected an average number of 17.2 statements, with an average number of 5.8 statements including API calls. We approximated 31.2% of the supported API methods by not giving precise formulations due to Kaluza limitations, as described in Section IV-D. We found this not to be a limitation to our approach for two reasons: this apparently large number of approximations were performed, in practice, over the 7.7% of the total number of translated statements including library calls, meaning that approximated transformations are, in general, infrequent. Second, it is worth noting that within the analyzed paths, very few include validation checks over the data in input. In particular, our tool found at most 3 checks in those paths, usually checks on null values. In short validation checks involving Intent payloads are performed in the 23% of the vulnerable paths.

E. Discussion of Results and Recommendations

From our results, it is clear that only a few applications properly implement validation of data received from Intent messages. Most applications perform only basic checks on Intent payloads parts, such as null-checks, performed only to

	Min	Max	Avg
Per-application execution time	2.4 min	33.2 min	12.3 min
Per-application components	3	31	24.5
Per-application vulnerable paths	2	19	4.2
Per-path statements	5	81	17.2
Per-path if-statements	0	3	0.98

TABLE VI: Other experimental setup

protect against malformed Intents payload tuples. Such checks are not strict enough to defend application resources against more sophisticated Intent spoofing attacks as the ones we considered in this work.

In order to verify the intents' origins, currently developers may need to implement custom authentication mechanisms, given the absence of a system-provided intent origin verification feature. A recommended solution consists in exchanging keys between intent sender and receiver, and requiring signatures to be exchanged within intents as a mean of authentication. This solution is very similar to the one used in web application exchanging CSRF tokens, and it is very effective in practice, since it guarantees complete protection to application components possibly exposed to Intent attacks. However, it requires communication of key material among applications before normal communications can start.

A similar mechanism to provide trust among application components in Android is the *signature* permission type, where a permission is automatically granted to a request from a component, if both the sender and receiver components belong to applications signed with the same developer key. However, this mechanism provides trust among apps developed by the same developer and would limit communications among apps of different developers.

An OS-level enhancement to offer to final users a set of features in the Intent API to reliably retrieve an intent's origin or sign intent's data would also help mitigate this risk. A work that proposes something similar can be found in [14].

Most importantly, developers should always include strong data validation checks, even when dealing with intents, and should follow all the guidelines to securely implement intent communications by not exposing their application components unnecessarily. In addition, another good programming practice is to limit as much as possible the functionality of external components, which may be subverted by an attacker.

VI. Related Work

Android ICC security has received an increasing interest in the past years. The current state of the art is presented in [7]. This work aims to provide a precise specification of applications ICC interconnections by using IFDS analysis to determine communication entry and exit points and determine security breaches. Despite some common points in the techniques, our work is focused on the paths inside a single application rather than analyzing a network of intercommunicating applications. In addition, we characterize vulnerabilities and provide exploit proofs.

In [1], the authors analyze the inter-application message passing system and identify risks of intent-spoofing, such as broadcast eavesdropping and denial of service. Their work is however focused on developer practices that allow intents to be sent by malicious applications and received by victim applications. Our work goes more in-depth and focuses on how the attackers can use intent-spoofing and leverage insufficient intent data validation checks inside applications to carry out additional attacks. Also, our work goes in the direction of automatic detection and exploitation of the vulnerabilities.

In FlowDroid, the authors present an IFDS-based taint analysis to derive information flow inside Android applications and detect malicious data leakages [15]. Conversely, in our work, IFDS-based taint analysis is used only as a first step towards symbolic execution of an application component with the goal of finding vulnerable paths inside an application.

A work related to defenses against intent spoofing is [14]. Quire is a lightweight framework to enhance Android IPC mechanism by adding message provenance. The implementation relies on a signature scheme that allows message recipient validation before delivering. The class of attacks they want to prevent is closely related to the ours: *confused deputy attacks*. Their approach needs to modify the Android OS in order to guarantee the described security features, our approach aims to obtain a similar result by preventing behaviors that can potentially introduce vulnerabilities in applications.

Other related studies includes different techniques for statically testing security of Android applications: Schmidt *et al.* [16] used static analysis to extract a list of function calls from which they can perform data analysis for malware detection. Mirzaei *et al.* [17] applied symbolic execution techniques in order to generate test cases and thus differs significantly from our work. ScanDal [18] is a static analyzer that implements a formal approach to automatically detect private data leaks in Android applications rather than detect active attacks as in our work. AppIntent [5] uses symbolic execution to detect if dataflows from application GUI elements are not intended by the user and differs from our approach for detecting vulnerabilities and providing proofs for them.

Our approach shares some characteristics with similar research on automatics exploit generation and vulnerability proving [19], [20]. In particular, symbolic execution is a common technique to for finding inputs that exercise specific paths in a program. However, our work differs from them in that it is applied to the Android context and requires a heavy use of interprocedural analysis. In addition, our work differs from Mayhem [20] in that it uses static taint propagation as a "heuristic" for pruning the search space. Similarly to our approach, in Dowser [19], the authors use dynamic taint taint propagation to find paths that are most likely to exercise predefined vulnerabilities.

VII. Conclusions

In this paper, we describe an automated analysis framework to study validation checks over the data received via Intent messages in Android applications. Our method for automatically detecting these vulnerabilities relies on static taint analysis and symbolic execution, using the IFDS framework. Furthermore, we use a solver to automatically generate a proof-of-concept attacker app that validate the vulnerabilities. We evaluate our approach on 64 popular applications downloaded from the Google Play Store, finding 29 potential vulnerabilities, and automatically exploiting (and thus confirming) 26 of these. Our results confirm that a large percentage of commonly-used applications do not implement appropriate security safeguards for Intent communications.

References

[1] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[2] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *USENIX Security Symposium*, 2011.

[3] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *NDSS*. The Internet Society, 2012.

[4] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: Statically vetting android apps for component hijacking vulnerabilities," in *ACM Conference on Computer and Communications Security (CCS 2012)*, pp. 229–240.

[5] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *ACM SIGSAC conference on Computer & communications security (CCS 2013)*. New York, NY, USA: ACM, pp. 1043–1054.

[6] "Android: Intents and intent filters," http://developer.android.com/guide/components/intents-filters.html, Feb. 2013.

[7] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android: An essential step towards holistic security analysis," in *22nd USENIX Security Symposium*, Washington, DC, USA, 2013.

[8] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[9] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *ACM SIGPLAN International Workshop on State of the Art in Java Program analysis (SOAP 2012)*. New York, NY, USA: ACM, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2259051.2259052

[10] "Kaluza," http://webblaze.cs.berkeley.edu/2010/kaluza/, 2010.

[11] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 513–528. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.38

[12] "Heros ifds/ide solver," http://sable.github.io/heros/, 2013.

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Conference of the Centre for Advanced Studies on Collaborative research (CASCON 99)*. IBM Press, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[14] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *20th USENIX Security Symposium*, San Francisco, CA, Aug. 2011.

[15] S. Arzt, S. Rasthofer, 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," in *35th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2014)*, 2014.

[16] 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 *IEEE International Conference on Communications (ICC'09)*. IEEE, pp. 1–5.

[17] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–5, 2012.

[18] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "Scandal: Static analyzer for detecting privacy leaks in android applications," in *Proceedings of the Workshop on Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, 2012.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 49–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534772

[20] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560217.2560219