Automatic Generation of Inter-Component Communication Exploits for Android Applications

Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek
Department of Informatics
University of California, Irvine
Irvine, California, USA
{joshug4, hamadm, negargh, malek}@uci.edu

ABSTRACT
Although a wide variety of approaches identify vulnerabilities in Android apps, none attempt to determine exploitability of those vulnerabilities. Exploitability can aid in reducing false positives of vulnerability analysis, and can help engineers triage bugs. Specifically, one of the main attack vectors of Android apps is their inter-component communication interface, where apps may receive messages called Intents. In this paper, we provide the first approach for automatically generating exploits for Android apps, called LetterBomb, relying on a combined path-sensitive symbolic execution-based static analysis, and the use of software instrumentation and test oracles. We run LetterBomb on 10,000 Android apps from Google Play, where we identify 181 exploits from 835 vulnerable apps. Compared to a state-of-the-art detection approach for three ICC-based vulnerabilities, LetterBomb obtains 33%-60% more vulnerabilities at a 6.66 to 7 times faster speed.

CCS CONCEPTS
• Security and privacy → Software security engineering;
• Software and its engineering → Software testing and debugging;

KEYWORDS
Android, exploit, vulnerability, test generation, test oracle

1 INTRODUCTION
Mobile devices are ubiquitous, with billions of smartphones and tablets used worldwide [6]. Among these popular mobile devices, Android has emerged as the dominant platform [1]. Fueling the popularity of such devices is the abundance of applications (apps) available on a variety of app markets (e.g., Google Play). This abundance of apps arises, in large part, due to the Android platform’s low barrier to entry for amateur and professional developers alike, where a re-usable infrastructure enables relatively quick production of apps. However, this low barrier to entry is associated with an increased risk of apps with defects, particularly in the form of security vulnerabilities [19]. Consequently, developers and designers of such apps need to utilize appropriate approaches, tools, and frameworks that aid them in producing secure apps.

To identify security vulnerabilities in Android apps, a plethora of approaches have been constructed [37]. Most of these approaches rely upon static analysis of Android apps to identify such vulnerabilities [7, 11, 16, 19, 25, 29, 33, 34, 37]. Several approaches utilize dynamic analysis for identifying vulnerabilities in Android apps [13, 21, 27, 31, 43]. Other approaches use a combination of static and dynamic analysis to identify vulnerabilities [22, 35, 38, 39, 41, 42, 44]. Although these approaches and techniques have provided useful means for identifying vulnerabilities, the exploitability of those vulnerabilities often must be determined manually by security analysts. Such a manual task is cumbersome, time-consuming, and error prone. Furthermore, vulnerabilities that are identified by existing techniques that are not actually exploitable do not pose a true security problem. The time an analyst spends on such non-exploitable vulnerabilities should be minimized.

Ideally, an approach is capable of both identifying vulnerabilities within an Android app and determining if such vulnerabilities are exploitable, in an automated fashion. Achieving the latter goal reduces the false positives produced by a security analysis and the time a human analyst must spend examining a vulnerability. Furthermore, identifying vulnerabilities that can be automatically exploited (1) aids software engineers in determining which bugs they should prioritize first, (2) provides an input to help fix the security bug, and (3) keeps engineers ahead of malicious actors that may create zero-day exploits of vulnerabilities.

To enable automatic exploit generation (AEG) for Android apps, two challenges must be overcome. First, specific Android constructs must be taken into account, including the distributed event-based, or message-based, framework that serves as an app’s attack surface. In particular, inter-component communication (ICC) both within and across Android apps relies primarily on the exchange of asynchronous messages, called Intents in Android. Furthermore, the Android framework provides a set of pre-defined components that react differently to Intents they receive. Another challenge of applying AEG to Android is providing a means of automatically assessing whether a vulnerability has been exploited.

To address these challenges, and apply AEG for Android, we present an approach, called LetterBomb, that (1) models the Android framework, especially the ICC interface of Android apps; (2) provides test input generation, whose goal is to construct an ICC input that actually exploits a vulnerability; and (3) includes software test oracles that determine if a test input successfully exploits a particular vulnerability type. Specifically, we focus on three types of vulnerabilities—inter-process denial of service, cross application scripting, and Fragment injection—where each vulnerability corresponds to a single oracle type. Each oracle is realized as a combination of instrumentation at either the app or framework level, and the check of a property to determine if exploitation was successful. As a result, even though each vulnerability requires an oracle designed specifically for it, construction of each oracle only
needs to be performed once, either as an algorithm that automatically instruments an app, or a one-time modification to the Android framework. Thereafter, the oracle may be continually reused.

Given that test input generation is critical for AEG at the ICC interface of Android apps and their constituent components, LetterBomb relies upon a path-sensitive analysis of Android apps along the message-based Android ICC interface, i.e., Intents. Determining exploitability of a vulnerability at a particular statement is dependent on assessing the different program paths that may reach a statement. Certain paths may reach a statement without exploiting the vulnerability residing at that statement—or there may be more than one path in a program that may exploit a vulnerable program statement. As a result, it is important for our analysis to be path-sensitive to minimize the possibility of missing exploitable vulnerabilities. At the same time, path-sensitive analyses face the potentially exponential number of program paths to be analyzed. To address this problem, our approach analyzes program paths beginning from the points in the program that may be vulnerable, and utilizes information about the Android framework to reduce the information that needs to be considered for the analysis.

The main contributions of our work are as follows:

- We present LetterBomb, the first approach for automatically generating exploits of Android apps at their ICC interface.
- We perform an evaluation of LetterBomb on 10,000 Android apps in terms of its ability to generate ICC exploits, reduce the false positives of a conservative path-sensitive static analysis, and the efficiency of the different parts of our approach. We have identified 181 exploits from 835 apps, and have informed the authors about the exploits and associated vulnerabilities. We further compare LetterBomb to a state-of-the-art detection approach for three ICC-based vulnerabilities, and find that LetterBomb obtains 33%-60% more vulnerabilities at a 6.66 to 7 times faster speed.
- We describe LetterBomb's implementation and make it available online [5].

2 BACKGROUND AND RUNNING EXAMPLE

To aid in understanding LetterBomb, we first cover the necessary background regarding the Android platform and describe a running example of an Android app we will reuse throughout the paper. Finally, we cover the foundational aspects of AEG.

Android Background. The Android Development Framework (ADF) supplies developers with a set of customizable components and communication mechanisms that allow construction of mobile apps. In particular, Android includes four pre-defined components: Activities, Services, Broadcast Receivers, and Content Providers. An Activity represents a GUI screen that an app displays to a user and allows her to interact with the app. A Service runs operations in the background of an app. A Content Provider represents persistent data storage of an app. A Broadcast Receiver receives Intents that are, as its name implies, broadcasted by other apps or the Android framework itself (e.g., indicating that the battery is low or that the device has finished booting). Activities, Services, and Broadcast Receivers can exchange Intents.

Activities may consist of Fragments, where each Fragment may represent a partial or whole screen viewable by a user. For example, one Fragment of an email app may contain the list of messages to be displayed, while another Fragment may contain the body of an individual message.

As an event-based system [20, 30], Android components, i.e., Activities and Services, may have multiple entry points corresponding to their lifecycle. For example, an Activity has separate entry points for initial creation and being sent to the background to pause the Activity. Broadcast Receivers have a single entry point, but may be registered dynamically.

From our studies of Android apps, there are mainly three types of attributes of an Intent that an app uses to determine the manner in which it will utilize the Intent and perform operations based on it: the action of an Intent, its categories, and its extra data. The action of an Intent is an attribute that indicates the general operation to be performed in response to an Intent (e.g., display data to the user or deliver data to some person or agent). Categories of an Intent provide additional information as to the manner in which the Intent’s action should be performed (e.g., whether the Intent will allow launching of an application as referenced by a link in a browser). Extra data, also called Bundles, are a collection of key-value pairs in an Intent, allowing even more flexible attributes to be stored in it.

Running Example. By supplying actions, categories, or extra data with malicious payloads, or excluding these attributes, an attacker can exploit an ICC-based vulnerability in an Android app. Figure 1 illustrates two activities AdsActivity, which shows ads to a user, and FragmentActivity, which utilizes Fragments as part of the UI of the Activity. Both of these Activities contain exploitable vulnerabilities that are reachable from the app’s ICC interface.

AdsActivity shows a banner ad if it receives an Intent whose action contains the string “BANNER” (line 5) or an interstitial ad, i.e., an ad that fills the entire screen, if the Intent’s action contains the string “APPWALL” (line 7). For the interstitial ad to display, the Intent must have (1) an integer extra data containing the key “expirytime” and value greater than 0, and (2) must not include the default Intent category (line 8). To display the interstitial ad, the AdsActivity relies upon a WebView, which is a class that displays a web page from within an Activity. The contents of the web page depend on a boolean extra data, supplied by the Activities incoming Intent, with a key “cached” (lines 14-18). If the value of this extra data is set to true, the web page is loaded from a cached URL (line 16); otherwise, the URL used as part of the displayed web page depends on a string extra data, with key “url”, supplied by the incoming Intent (line 18).

Two vulnerabilities exist within the AdsActivity. First, by sending an Intent with no action to this Activity, a null pointer exception will be thrown when the Activity checks the Intent for a "Banner" action. Specifically, invoking the equals method on the action string is an invocation of a null object (line 5), which results in the app crashing as the thrown exception is not caught. This vulnerability can be leveraged by a malicious app to perform an inter-process denial-of-service (IDOS) attack on the AdsActivity by periodically sending an Intent with no action.

The second vulnerability occurs when the incoming Intent of the AdsActivity loads a web page supplied by an Intent. A malicious app can send an Intent to the AdsActivity and attempt to perform a spoofing attack by redirecting the user to a web page requesting sensitive personal information. Alternatively, the WebView may be redirected to a page with malicious JavaScript code. This vulnerability is similar to cross-site scripting vulnerabilities in web applications, and is referred to as cross-application scripting (XAS) vulnerabilities for Android apps [27].
3 LETTERBOMB OVERVIEW

Figure 2 depicts a high-level overview of LetterBomb. Vulnerability Identifier (VI) conservatively analyzes the inputted app to identify statements where exploitable vulnerabilities may exist, and passes those vulnerable statements to Attack Intent Generator (AIG) and Exploit Oracle Instrumenter (EOI).

AIG performs a backwards static symbolic execution (SSE) starting from the vulnerable statements identified by VI to determine the payload of Intents that, when sent to the vulnerable component, are likely to execute each vulnerable statement. By performing SSE from the vulnerable statements, the SSE reduces the possibility of path explosion faced by performing a symbolic execution. Based on the type of vulnerability identified, AIG determines the appropriate modification to the Intent needed to potentially exploit the vulnerability (e.g., supply a value to an extra data, or leave out an extra data). Each generated Intent is (1) an attack on the app, whose goal is to exploit a particular vulnerable statement, and (2) a security test case. VI and AIG together aim to satisfy $n_{bug}$ of the AEG equation in Section 2 and the attacker’s logic of $\pi_{exploit}$.

In the following sections, we discuss each of the three major components of LetterBomb: VI, AIG, and EOI.

4 VULNERABILITY IDENTIFICATION

VI’s main goal is to conservatively identify potentially vulnerable statements. By performing vulnerability analysis conservatively, the output of VI may be less precise, but such an analysis ensures the vulnerabilities are well-covered during SSE and security testing. In the remainder of this section, we will discuss the static analyses we perform to identify statements vulnerable to the three types of vulnerabilities that we focus on in this paper.

Inter-Process Denial-of-Service. To identify potential IDOS attacks, VI checks for uses of an Intent’s payload that may cause an unhandled null pointer exception. To that end, VI examines each use of an Intent attribute (i.e., an Intent action, extra data, or category) and performs a backwards data-flow analysis along the use-def chain [14] of the corresponding attribute to determine if there is a null check of the attribute [19]. A null check is a conditional comparison of an object against null.

Figure 3: Modified example from lines 5-6 of Figure 1

To illustrate, consider Figure 3, which contains a modified code snippet from lines 5-6 of Figure 1. Line 1 of Figure 3 contains a null check of the Intent action. If VI’s backwards data-flow analysis
along the use-def chain of an Intent attribute does not include a null check, VI marks the use as vulnerable to an IDOS attack, and adds that statement as a vulnerable statement to be outputted.

**Cross-Application Scripting.** VI identifies statements vulnerable to XAS attacks by first identifying invocations of WebView.loadUrl(...) in an application. Starting from such a statement, VI performs a backwards data-flow analysis along the use-def chain of arguments passed to the invocation WebView.loadUrl(...) at that statement. If any of those arguments are uses of a definition statement whose right-hand side involves the extraction of an Intent attribute, VI considers the statement vulnerable to an XAS vulnerability.

As an example, consider the statement at line 19 of Figure 1 that is vulnerable to XAS. VI follows the ur1 string argument at that invocation along the argument’s use-def chain. At line 16, the definition of ur1 does not involve data extracted from an Intent—recall that the URL is extracted from a cache. A summary of the getLastCachedUrl() method may be used to make this determination, or this information may be determined during runtime when generated Intent attacks are executed. On the other hand, at line 18, ur1 is assigned its value from a string extra-data attribute. As a result, VI marks line 19 as vulnerable to XAS.

**Fragment Injection.** To determine if a statement is vulnerable to FI, VI checks (1) if the statement invokes Fragment.instantiate(...) and (2) if the second positional argument of that method, i.e., the name of the Fragment to load, is controllable using an Intent attribute by following the use-def chain of that argument.

For example, consider the invocation of Fragment.instantiate(...) at line 25 of Figure 1. fragmentName is the second positional argument of that invocation, indicating the name of the Fragment to be loaded from within the app under analysis. By following the use-def chain of this object backwards, we find its sole definition at line 24 of Figure 1. At that statement, fragmentName is assigned the value of the extra data corresponding to the key “frag_name” of FragmentActivity’s incoming Intent. As a result, an Intent can control the Fragment loaded at line 25, making that statement vulnerable to a FI.

## 5 ATTACK INTENT GENERATION

AIG performs two key functionalities to exploit an ICC-based vulnerability: (1) computes Intents that can execute a vulnerable statement along all possible Intent-controlled paths, and (2) modifies Intent attributes to supply vulnerability-specific logic of an attack.

### 5.1 Reaching Intent Generation

To perform (1), AIG relies on an algorithm we refer to as Reaching Intents, which is a flow-sensitive, context-sensitive, object-sensitive, and path-sensitive backwards SSE and a backwards data-flow analysis over the app’s use-def chains [14], beginning at each vulnerable statement supplied to it by VI. By starting the SSE from vulnerable statements, the SSE prunes the space from which paths must be computed, as opposed to a forward symbolic execution starting from the ICC-based entry points of the app to all statements reachable from those entry points. This pruning significantly reduces AIG’s computation time. Each backwards SSE can be computed independently per vulnerable statement—allowing the backwards SSE to be parallelized, further improving scalability of AIG’s analyses of Intents needed to execute a vulnerable statement.

![Algorithm 1: intentControlAnalysis](image)

To obtain a call graph suitable for analysis of Android apps, the call graph must take into account the multiple entry points of an Android app and its lifecycle. To achieve this, Reaching Intents incorporates incremental callback analysis to construct a call graph as described in previous work [7], where the call graph is continuously updated with identified callback registrations until a fixed point is reached.

The main algorithm driving Reaching Intents’s analysis, intent-ControlAnalysis, is depicted in Algorithm 1. Similar to previous analyses [23, 32], intentControlAnalysis is a summary-based analysis that processes methods in the app’s call graph in reverse topological order, and takes as input targetStmts, i.e., the vulnerable statements identified by VI. By analyzing methods in that order, intentControlAnalysis ensures that a callee method’s summary is constructed and available before a caller method is analyzed, preventing the need to analyze a method more than once and improving intent-ControlAnalysis’s efficiency. intentControlAnalysis returns a map $\Sigma : M \rightarrow \text{targetExprs}$ summarizing the analysis results for each method $m \in M$. Essentially, this map contains the expressions describing the Intent, including its attributes, to be generated that may reach vulnerable statements in the app. Each $e \in \text{targetExprs}$ is a pair $(s_e, \text{exprs}_e)$. exprs$_e$ is a sequence of expressions describing Intents and path conditions in a program path $p$, and $s_e$ is a vulnerable statement, where backward symbolic execution initiates from, which is further elaborated in the next section.

Algorithm 1 analyzes each method $m$ by first constructing $m$’s use-def chains (line 3 of Algorithm 1). For each vulnerable statement $s_f$ of a method $m$, line 6 of Algorithm 1 constructs all the relevant program paths from the entry point of the program to the vulnerable statements by invoking constructBackReachPaths. To avoid analyzing paths that may not actually reach these vulnerable statements, and thus improving analysis efficiency, constructBackReachPaths builds these relevant program paths through a backward traversal algorithm over a method’s control-flow graph.

intentControlAnalysis determines whether to utilize $\Sigma$ for an invoked method in $s_p$ of path $p$, or construct entirely new Intent expressions for $s_p$. For invoked methods, lines 10-14 of Algorithm
Algorithm 2: generateExprsForStmt

Input: A statement $s_{p}$, method of statement $m$, path in $m$ containing $s_{p}$, user-def chain $useDef_{m}$ of method $m$
Output: expressions newExprs describing Intent and path-condition information at statement $s_{p}$

1. newExprs $\leftarrow \emptyset$.
2. if $s_{p}$ extracts extra data from an Intent $i$ of the form $r_{k} = i.get Państw(\text{Extra})$ then
   3. newExprs $\leftarrow genExtraDataExprs(s_{p}, p, useDef_{m}) \cup$ newExprs.
4. else if $s_{p}$ extracts an action from an Intent $i$ of the form $r_{k} = i.getAction()$ then
   5. newExprs $\leftarrow genGetActionExprs(s_{p}, p, useDef_{m}) \cup$ newExprs.
6. else if $s_{p}$ is of the form if ($r_{k}$ equals ($r_{j}$)) then
   7. newExprs $\leftarrow genAssertExprs(s_{p}, p, useDef_{m}) \cup$ newExprs.
   8. if $r_{k}$ is an Intent then
   9. newExprs $\leftarrow genCategoryExprs(s_{p}, p, useDef_{m}) \cup$ newExprs.
10. else if $r_{k}$ is an arbitrary object obtained from an Intent's extra data then
     11. newExprs $\leftarrow genObjEqualityExprs(s_{p}, p, useDef_{m}) \cup$ newExprs.
     12. else if $s_{p}$ is a conditional statement of the form if ($r_{k}$) then
     13. newExprs $\leftarrow genConditionalExprs(s_{p}, p) \cup$ newExprs.

1. utilize method summaries to determine context-sensitive Intent information at a call site by enumerating expression paths from $e$ and updating $\Sigma$ based on the current method $m$ under analysis. For other types of statements, lines 16-17 of Algorithm 1 utilize generateExprsForStmt to construct new Intent expressions. In that block of code, intentControlAnalysis stores Intent information (i.e., expressions describing the Intent’s attributes) in $\Sigma$. At this point, the computed path conditions and expressions describing Intents may be sent to a solver to check for feasibility, and to generate Intents that can execute specific program paths.

5.1.1 Generating Intent Expressions. The production of expressions describing message-controlling Intents occurs during the first phase, on lines 8-17 of Algorithm 1. For each statement $s_{p}$ in a path $p$ that reaches vulnerable statement $s_{k}$, Algorithm 1 at line 16 generates a set of expressions describing the Intent or the path conditions at $s_{k}$ by invoking generateExprsForStmt, shown in Algorithm 2.

generateExprsForStmt generates each expression in a language suitable for supplying to an SMT solver, i.e., the SMT-LIB language [12], allowing our analysis to use the SMT solver to determine feasibility of paths, and also the validity of the expressions describing Intents, their attributes (i.e., actions, categories, and extra data), and their relations to programming language-level constructs (e.g., object references, definition sites, etc.). In a post-processing phase, $\tilde{A}$ generates these SMT expressions to construct Intents that can be executed by an appropriate test bed. To support a variety of Intent usages, we model primitive comparison operators for numerics and booleans, i.e., $\equiv$, $\neq$, $<$, $>$, $\geq$, $\leq$, and $\geq$.

generateExprsForStmt takes as input a statement $s_{p}$ of method $m$, path $p$ in $m$ containing $s_{p}$, and user-def chain $useDef_{m}$ of method $m$. As output, generateExprsForStmt constructs expressions describing an Intent and path conditions that the Intent must satisfy to reach a vulnerable statement. By considering the path $p$ of $s_{p}$, generateExprsForStmt ensures that expressions generated for $s_{p}$ are relevant to $p$, thus maintaining path sensitivity. Each conditional block in Algorithm 2 handles a different type of program statement and generates expressions based on that statement type.

Extra Data. To handle extra data, genExtraDataExprs (at line 3 of Algorithm 2) produces symbolic variables for the following references: $r_{k}$, key of the extra datum extracted from the Intent $i$; $r_{k}$, containing the value of the extra datum; and $i$ for the reference of the Intent housing the extra datum. genExtraDataExprs further records the type of the extra datum at the programming language-level when declaring a new symbol by taking the API method’s type $\Psi$ into account. For example, in the case of the API method getIntExtra, $\Psi$ is an integer. To represent the new generated information, genExtraDataExprs creates expressions of the following form, with declarations removed for brevity:

$e_{1}$ (assert (= (containsKey $r_{k}$ $r_{k}$) true))
$e_{2}$ (assert (= (fromIntent $r_{k}$) $i$))
$e_{3}$ (assert (= (containsKey $r_{k}$ "expirytime") true))
$e_{4}$ (assert (= (fromIntent $r_{k}$) "intent") )

...
existence or absence of a category. Therefore, for existence of a category, genCategoryExprs generates the following expression:

\[
\text{assert}(\exists(x: \text{Int})(\text{select cats}_{\text{r}}} x) r))
\]

The above expression simply asserts existence of an element at index \(idx\) in the array \(cats_{\text{r}}\) that contains the value \(r\), using the existential quantifier. For absence, genCategoryExprs generates the following expression:

\[
\text{assert(∀(∀(\text{idx}\text{Int})(\text{not}(\text{select cats}_{\text{r}} \text{idx}) r)))}
\]

The expression asserts that for all elements in the array \(cats_{\text{r}}\) there is no element with the value \(r\). To relate the categories \(cats_{\text{r}}\) to an Intent \(i\), genCategoryExprs produces an expression using the fromIntent function.

For example, consider a partial path from lines 8-9 in Figure 1. At line 8, genCategoryExprs generates the following expression, where we elide the fromIntent expression due to space limitations:

\[
\text{assert(∀(\text{idx}\text{Int})(\text{not}(\text{select cats}_{\text{r}} \text{idx}) r)))}
\]

```
“android.intent.category.DEFAULT”))
```

**String and Object Comparisons.** Equality comparisons among strings, and to a lesser extent objects in general, are critical for determining the contents of Intents that control execution of different program paths. Although other forms of string manipulation may potentially affect execution, they are extremely rare, as found both in this study and previous work [8, 11, 23, 36]. Consequently, our analysis focuses on representing and handling string equality.

Specifically for Intents, determining string extra data and values of actions for an Intent are dependent on extracting equality comparisons. As an example, for any path that reaches line 9 of Figure 1, the equals comparison of strings at line 7 in that figure must evaluate to true. Furthermore, along that path, the action of the Intent must be “APPWALL”.

To extract Intent information from string comparisons, genStringAttrExprs (invoked on line 10 of Algorithm 2) analyzes string equality statements. For statements of the form shown on line 8, genStringAttrExprs creates symbols for references \(r1\) and \(r2\) declared as built-in strings and generates an expression of the form \(\text{assert}(\text{eq}(\text{select}\ r1)\ r2)\) if the comparison is true along the path under analysis, and generates the expression \(\sim\text{eq}(\text{select}\ not(\text{eq}(\text{select}\ r1)\ r2))\) otherwise. As in the case of non-conditional extra data extraction, expressions of the form \(e1\) and \(e2\) are generated as well, describing the key of the string extra datum and the Intent it belongs to.

To obtain potential values for actions of an Intent along a path, genStringAttrExprs need only generate the assertion expressions of the form \(\text{assert}\ or\ \sim\text{eq}\). These expressions combined with the expressions of the form \(a1\) and \(a2\) extracted by genGetActionExprs describe potential string values for actions of a particular Intent.

For example, in line 7 in Figure 1 with a path ending at line 9 of Figure 1, genStringAttrExprs generates the following expressions:

\[
\begin{align*}
&\text{a3)}\ \text{assert}(\text{eq}(r_a \text{ "APPWALL"}) ) \\
&\text{a4)}\ \text{assert}(\text{not}(\text{eq}(r_a \text{ "BANNER"}) )
\end{align*}
\]

The relevant information along the path for the action, generated by genGetActionExprs, are as follows:

\[
\begin{align*}
&\text{a5)}\ \text{assert}(\text{getAction}(i_j) \text{ r}_j) \\
&\text{a6)}\ \text{assert}(\text{fromIntent}(r_a) i_j)
\end{align*}
\]

In expression a5 and a6, the Intent symbol’s subscript represents the line number where the Intent is created. In that case, the Intent that starts AdsActivity is obtained at line 3 of Figure 1.

When comparing arbitrary objects, genObjEqualityExprs (invoked on line 12 of Algorithm 2) operates in a manner highly similar to that of genStringAttrExprs. genObjEqualityExprs still creates expressions of the form \(\text{assert}\ or\ \sim\text{eq}\). These objects are declared as our custom Object type and may also be assigned to an Intent using the fromIntent function.

A special case occurs when a string \(r_o\) is first compared with the null constant along a path and is later compared with a specific string. This case occurs in Figure 3, where \(r_o\) refers to the Intent’s action. To avoid type conflicts in such a case, we generate a symbol for a reference \(r_o\) as an Object and another symbol for \(r_o\) as a string. We then create a custom function objEqual1s that allows comparison of strings with objects for the SMT solver.

**Conditional Comparison Operators.** Lines 13-14 of Algorithm 2 handle conditional expressions involving comparison operators. This part of Algorithm 2 involves straightforward substitutions of operators and variables, which we exclude in this paper due to space limitations. Additionally, we model the null constant as its own special type in SMT, since null values are used often and are treated as a special case in Java code.

5.1.2 Constructing Context-Sensitive Results. Lines 10-14 of Algorithm 1 enumerates paths inter-procedurally by identifying call sites of summarized methods stored in \(\Sigma\). Specifically, intentControlAnalysis checks three criteria to determine where to enumerate paths for a statement \(s_p\) in path \(p\) under analysis: (1) \(s_p\) is a call site to a summarized method \(m_1\) in \(\Sigma\), (2) an argument \(a\) passed to \(m_1\) is an Intent, and (3) \(\exists\ m_j\) contains expressions indicating that information is generated from a parameter of \(m_j\) that matches \(a\). For example, line 9 of Figure 1 is a call site where the method doAppWallAd is invoked and is also summarized in \(\Sigma\).

To clarify, consider the following intra-method path \(p_{oc}\ =\ (2, 3, 4, 5, 7, 8, 9)\) of onCreate in Figure 1; and the two intra-method paths of doAppWallAd from that figure: \(p_{dc\_1}\ =\ (10, 11, 12, 13, 14, 15, 16, 19)\) and \(p_{dc\_2}\ =\ (10, 11, 12, 13, 14, 15, 17, 18, 19)\). In this example, intentControlAnalysis has already summarized method doAppWallAd and is now analyzing method onCreate, since intentControlAnalysis analyzes methods in reverse topological order. In these three paths, the numbers in the path represent the line numbers in the figure. The Intent \(i_{oc}\) for path \(p_{oc}\) has action “APPWALL”, an integer extra data with key-value pair (“expirtime”, > 0), and no default category. The Intent \(i_{dc\_1}\) for \(p_{dc\_1}\) has extra data (cached, true); Intent \(i_{dc\_2}\) for \(p_{dc\_2}\) has extra data (cached, false).

Once Algorithm 1 reaches lines 16-17, intentControlAnalysis combines these three paths into two context-sensitive paths. Specifically, \(p_{oc}\ \sim\ p_{dc\_1}\) forms a final context-sensitive path with a new Intent \(i_{oc\_dc\_1}\) that is simply a combination of Intents \(i_{oc}\) and \(i_{dc\_1}\). Similarly, \(p_{oc}\ \sim\ p_{dc\_2}\) forms a second final context-sensitive path with a new Intent \(i_{oc\_dc\_2}\) that is the combination of \(i_{oc}\) and \(i_{dc\_2}\). intentControlAnalysis stores info for both context-sensitive paths and their new Intents in \(\Sigma\). \(i_{oc\_dc\_1}\) is \(i_{oc}\) with extra data (cached, true). \(i_{oc\_dc\_2}\) is \(i_{oc}\) with extra data (cached, false). Note that if the combination of two Intents conflict, the solver, as part of the symbolic execution, would detect the conflict and determine that the path is infeasible.

For example, consider an Intent \(i_1\) has action \(a_1\), Intent \(i_2\) has action \(a_2\), and \(a_1\ \neq\ a_2\). Combining \(i_1\) with \(i_2\) is a contradiction, which the solver would detect.

5.2 Vulnerability-Specific Intent Modification

Once AIG computes the attributes of an Intent needed to execute a path leading to a vulnerable statement, an attack specific to a
vulnerability must be crafted. To that end, AIG accepts two modifications to an Intent computed using Reaching Intents: (1) removal of an attribute of an Intent and (2) modification of a value of an attribute. In the remainder of this section, we describe how these Intent modifications are applied for the three vulnerability types we focus on in this paper. Note that once this vulnerability-specific intent modification is designed and constructed, it can be reused for every app and vulnerability instance.

For an IDOS attack, the Intent modification is to exclude the attribute that would cause a crash to occur, which in our case is a null pointer exception. For the example of the IDOS vulnerability occurring at line 5 of Figure 1, the specific modification is to exclude the attribute with no null check, which is the Intent action in this case, and is determined automatically by VI.

To generate an exploit for XAS vulnerabilities, our analysis automatically identifies the key of the extra data to target, by examining the string passed to WebView.loadUrl(...) and supplies the value of the extra data corresponding to a URL within our control. For the example at line 19 of Figure 1, VI computes the extra data with key "url" as vulnerable, and specifies a URL under our control as the malicious value.

For FI, the Intent extra data’s key that controls the Fragment to be injected is identified by the second positional argument of Fragment.instantiate(...). At the same time, we determine potential Fragment values to inject by identifying any class that inherits from the android.app.Fragment class. In Figure 1, the Intent is altered so that the extra data with key "frag.name" includes the value "MainFragment".

6 EXPLOIT ORACLE INSTRUMENTATION

To detect if a generated Intent successfully exploits a vulnerability, EOI produces an oracle that is instrumented into the app or the Android framework. For each of the three aforementioned vulnerability types, we describe the manner in which we specify, generate, or instrument the oracle. Although each vulnerability type requires a customized oracle, we only need to specify or construct the instrumentation, or instrumentation algorithm, once. After that one-time specification or construction, we can continually reuse the resulting oracle or instrumentation to detect successful exploitation.

To instrument apps vulnerable to an IDOS attack, EOI adds, for each vulnerable statement, an instrumented statement that logs whether the vulnerable statement has been executed. Additionally, EOI post-processes the logged information to identify exceptions indicating crashes of the app (e.g., an exception printout with a stack trace) after a vulnerable statement is executed.

XAS instrumentation requires a special algorithm that instruments each statement vulnerable to XAS. To identify if injection of the malicious URL is successful, instrumentation must determine the URL loaded once the WebView's page has finished loading. To that end, EOI instruments a vulnerable WebView object by providing it a WebViewClient designed to log the current URL of a WebView once its page has finished loading. A WebViewClient allows overriding of callbacks of WebView in order to perform custom functionalities on a variety of a WebView's events. An example of how such instrumentation can be achieved is shown below, where webView is an object of class WebView.

```java
webView.setWebViewClient(new WebViewClient() {
    @Override
    public void onPageFinished(WebView view, String url) {
        Log.i("Instrument","loaded url:" + url);
        super.onPageFinished(view, url);}});
```
Table 1: Apps used for reaching Intent accuracy experiments

<table>
<thead>
<tr>
<th>App Package Name</th>
<th>App Description</th>
<th>SLOC</th>
<th>IPaths</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.samsung.appl</td>
<td>List a device’s apps by categories of permissions</td>
<td>4,649</td>
<td>47</td>
</tr>
<tr>
<td>com.naholyx.android.horairesmf</td>
<td>Search and track regional trains in France</td>
<td>4,054</td>
<td>90</td>
</tr>
<tr>
<td>crit_sanity</td>
<td>Phone call, SMS, audio recording, and bluetooth management</td>
<td>9,604</td>
<td>458</td>
</tr>
<tr>
<td>com.ghostsq.commander</td>
<td>Multi-protocol local and remote file manager</td>
<td>24,883</td>
<td>973</td>
</tr>
<tr>
<td>org.thiallihat.android.app</td>
<td>Android port of OpenPGP for data encryption and decryption</td>
<td>461,338</td>
<td>2,650</td>
</tr>
</tbody>
</table>

application domain (e.g., security, file management, regional train tracking, etc.). They vary in their sizes in terms of SLOC—from 4KSLOC to over 460KSLOC. Most importantly, these apps exhibit sophisticated Intent usage by performing different operations along different program paths based on the Intents they receive, and the contents of those Intents. They include apps with over 2,600 program paths that involve Intent usage or Intent control.

For each app, we manually checked every program path—over 4,200 program paths—to determine if the expressions generated correctly describe the Intents and path conditions, particularly Intent-controlling path conditions. We checked the correctness of Intent information generated along a program path in the following conservative manner: If our analysis generated any extra information not valid for the path, we considered all of its information incorrect. For example, if any extra datum was missing along a program path, we considered the entire path incorrect. Consequently, we deem any partially correct expressions describing Intents or path conditions as completely incorrect. Furthermore, if extra information about an Intent was generated by Reaching Intents, we also consider all the Intent information generated for a program path as incorrect. For instance, a spurious extra datum that is described as belonging to an Intent is considered extra information and, for evaluation purposes, renders all information along the path as incorrect. To that end, we use the following correctness metric to assess the accuracy of Reaching Intents per app:

\[
\text{Correctness Rate} = \frac{P_{cor}}{P_{tot}} \times 100
\]

\(P_{cor}\) is the number of correct Intent control-based paths; \(P_{tot}\) is the total number of Intent control-based paths.

The accuracy results that answer RQ1 are shown in Table 2. For each app, the table lists the number of paths with correct Intent information \(P_{cor}\), the number of paths with incorrect Intent information \(P_{inc}\), the total number of Intent-controlling paths \(P_{tot}\), and the correctness rate \(\% \text{Correct}\).

Reaching Intents’s correctness rate is very high with no app having a rate lower than 96%. Overall, this indicates that for the overwhelming majority of cases, Reaching Intents generates correct Intent information.

Table 2: Accuracy of Reaching Intents’s analysis

<table>
<thead>
<tr>
<th>App Package Name</th>
<th>(P_{cor})</th>
<th>(P_{inc})</th>
<th>(P_{tot})</th>
<th>% Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.samsung.appl</td>
<td>47</td>
<td>0</td>
<td>47</td>
<td>100.00%</td>
</tr>
<tr>
<td>com.naholyx.android.horairesmf</td>
<td>90</td>
<td>0</td>
<td>90</td>
<td>100.00%</td>
</tr>
<tr>
<td>crit_sanity</td>
<td>454</td>
<td>4</td>
<td>458</td>
<td>99.13%</td>
</tr>
<tr>
<td>com.ghostsq.commander</td>
<td>906</td>
<td>17</td>
<td>943</td>
<td>96.66%</td>
</tr>
<tr>
<td>org.thiallihat.android.app</td>
<td>2,565</td>
<td>85</td>
<td>2,650</td>
<td>97.79%</td>
</tr>
</tbody>
</table>

7.2 RQ2: Exploitability

To assess RQ2, we applied LetterBomb to a random set of 10,000 apps from Google Play. Table 3 depicts the results of that study: including information about the three different Vulnerability Types of this paper—IDOS, XAS, and FI; the number of Apps for which LetterBomb staticaly detected a vulnerability; the number of apps for which LetterBomb successfully generated an exploit (Exploited Apps); the number of Vulnerabilities detected by LetterBomb’s VI; the number of Exploits detected for each vulnerability type; and the number of Unique Exploits, where an exploit is unique if it either reaches a unique vulnerable statement or, in the case of FI, successfully injects a unique Fragment.

Table 3: Detected vulnerabilities and generated exploits

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Apps</th>
<th>Exploited Apps</th>
<th>Vulnerabilities</th>
<th>Exploits</th>
<th>Unique Exploits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOS</td>
<td>750</td>
<td>54</td>
<td>1,866</td>
<td>104</td>
<td>71</td>
</tr>
<tr>
<td>XAS</td>
<td>33</td>
<td>25</td>
<td>33</td>
<td>25</td>
<td>22</td>
</tr>
<tr>
<td>FI</td>
<td>52</td>
<td>3</td>
<td>193</td>
<td>52</td>
<td>52</td>
</tr>
</tbody>
</table>

LetterBomb successfully exploited 54 apps containing IDOS vulnerabilities, 25 apps containing XAS vulnerabilities, and 3 apps containing FI vulnerabilities. LetterBomb obtained 71 unique exploits and 104 exploits in total for IDOS, 25 unique and total exploits for XAS, and 52 unique and total exploits for FI. Note that a vulnerable statement may be exploited from more than one program path, resulting in non-unique exploits with respect to a vulnerable statement. Furthermore, we have informed the app authors of these exploits. These results indicate that LetterBomb is capable of producing a sizeable number of exploits.

7.3 RQ3: Spurious Vulnerability Reduction

Although a static analysis may be effective for identifying vulnerabilities, exploitability of a vulnerability can aid in assessing whether a statically determined vulnerability is spurious, since it is not exploitable. By producing attack Intents for all possible paths to a vulnerable statement, we can determine if a vulnerable statement is spurious. To assess the spuriousness of the VI’s vulnerability detection, we compute the false discovery rate (FDR) as follows:

\[
\text{FDR} = \frac{\text{nev} \times 100}{\text{nev} + \text{uev}}
\]

where nev is the number of non-exploitable vulnerabilities and uev is the number of exploits that are unique with respect to a vulnerable statement. nev counts the number of false positives of a static vulnerability analysis; uev counts the number of true positives in such an analysis. Given that no security analysis can say with absolute certainty that they have not missed a potential vulnerability (e.g., due to an indeterminable number of special cases), FDR is a much more sensible metric that takes false positives and negatives into account, compared to more traditional ones such as precision or the false positive rate, which is infeasible due to the need to account for such misses or false negatives. Furthermore, FDR allows us to answer RQ3 by indicating the extent to which the static vulnerabilities are false positives or spurious.

Table 4: Spurious vulnerability reduction results

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Non-Expl. Vuln.</th>
<th>Uniq. Expl. Vuln.</th>
<th>False Discovery Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDOS</td>
<td>1,762</td>
<td>71</td>
<td>96.13</td>
</tr>
<tr>
<td>XAS</td>
<td>8</td>
<td>25</td>
<td>24.24</td>
</tr>
<tr>
<td>FI</td>
<td>141</td>
<td>52</td>
<td>73.06</td>
</tr>
</tbody>
</table>

Table 4 shows the results of our study for RQ3: including the vulnerability type, non-exploitable vulnerabilities (Non-Expl. Vuln.), unique exploitable vulnerabilities (Uniq. Expl. Vuln.), and the false discovery rate results as a percentage. 100% means all the detected vulnerabilities could not be exploited; 0% means no spurious vulnerability reduction was achieved. Our results indicate that LetterBomb produces very high reductions for IDOS (96%) and FI (73%)
vulnerabilities, and significant reductions for XAS vulnerabilities (24%). Note that our manual analysis of over 4,200 program paths involving Intents from RQ1 gives us high confidence that nearly no potential attack Intents were missed. Recall that more than 96% of those program paths had correctly produced Intents, and that any incorrect Intent information makes the entire program path incorrect for that experiment.

7.4 RQ4: Runtime Efficiency

To answer RQ4, we ran VI and AIG, i.e., the static analysis components of LetterBomb, on 1,000 apps randomly selected from our original set of 10,000 apps. For this experiment, we used a machine with two AMD Opteron 6376 2.3GHz 16MB Cache Sixteen-Core Processors and 256GB RAM.

Table 5 presents the average, minimum, and maximum execution time, in seconds, of the static analysis portion of LetterBomb for each vulnerability type. On average, static analysis of LetterBomb took between 99 and 182 seconds. The minimum execution time for LetterBomb’s static analysis took between 1 and 2 seconds; the maximum execution time was 39 minutes to 3.5 hours, depending on the vulnerability type.

The overwhelming majority of apps take about 3 minutes to analyze statically, despite our use of symbolic execution to make our analysis path sensitive. LetterBomb analyzed multiple apps at once, dedicating 4 cores to potentially parallelize analysis of each app. More cores would allow further parallelization and reduce runtime. Furthermore, 3.5 hours in the worst case is a reasonable analysis time for generating highly precise attack Intents.

For the dynamic-analysis portion of LetterBomb, we executed the generated Intents on the instrumented apps or platform using the Andy [2] Android device emulator on a MacBook Pro with a 2.8 GHz Intel Core i7 and 16GB RAM. For each attack Intent sent to an app, LetterBomb waited 3 seconds before stopping the app under attack, in order to reset its state. LetterBomb would then wait another 2 seconds before sending the next attack Intent.

Table 6 depicts the execution time results for the dynamic portion of LetterBomb. For each vulnerability type (Vuln. Type), the table shows the number of apps analyzed, the number of attack Intents sent to all apps, the average execution time per app in minutes, and the total execution time for all apps in minutes.

As our results indicate, LetterBomb’s dynamic analysis of Android apps to assess vulnerabilities is fast, allowing us to analyze hundreds of apps marked as vulnerable, according to LetterBomb’s static analysis, in a matter of hours. On average, analysis time per app is between 11 seconds and 1.25 minutes.

On average, LetterBomb’s combined static and dynamic analysis time is between 3.12 to 4.30 minutes, for a single app and vulnerability type. For an analysis that combines path-sensitive static symbolic execution with a dynamic analysis, this execution time is reasonably fast, especially for generating an exploit.

7.5 RQ5: Vulnerability Detection Comparison

Although no approach automatically generates exploits for Android apps, one approach, IntentDroid [27], focuses on vulnerabilities from the Intent interface and analyzes the three types of vulnerabilities that LetterBomb targets. Unfortunately, this approach’s implementation is proprietary, owned by IBM, and costs between $204 and $417 USD per app scanned, which is a prohibitive cost for conducting a scientific study involving large numbers of apps. IBM does offer a 30-day trial of this scanner, which is called IBM Application Security on the Cloud (ASC), that allows a maximum of 10 apps to be scanned. To address these usage limitations of IBM ASC, a set of graduate students created trial accounts enabling us to analyze and automatically generate exploits, a feature IBM ASC does not support.

Table 8 shows the results of LetterBomb’s vulnerability comparison with IBM ASC—including the three targeted vulnerability types (Vuln. Types), and the number of apps detected as vulnerable according to each approach and the ground truth. For all 40 apps, LetterBomb is capable of identifying all their vulnerabilities. For IDOS, IBM ASC misses 9 vulnerable apps, which constitutes a 33% improvement for LetterBomb. In the case of XAS apps, IBM ASC misses 8 apps, which constitutes a 60% improvement for LetterBomb. For FI, IBM ASC misses a single app; thus, LetterBomb performs 33% better than IBM ASC.

8 THREATS TO VALIDITY

In terms of accuracy, the main threat to external validity is the selection of the five apps we utilized for answering RQ1. To mitigate this threat, we selected apps varying across several dimensions, allowing us to draw more general conclusions about Reaching Intents’s analysis results, be more confident in the accuracy of that
analysis, and reduces bias due to the limited number of apps. These apps come from different application domains, and they vary in size to as much as 460KLOC and collectively contain over 4,200 program paths involving sophisticated usage of Intents that control execution along different program paths. To identify these program paths, we built automated analyses that determine Intent usage and computed the program paths that contain them. All of these apps are from Google Play, the official Android market maintained by Google. The number of apps is limited to five to enable us to evaluate manually the accuracy of LetterBomb’s reaching Intent analysis on a large number of program paths—a painstaking task that took about a year to complete.

For each vulnerability type, an instrumentation algorithm at the app level is constructed, or the Android framework itself is instrumented. However, each instrumentation algorithm or framework modification involves only a few dozen lines of code. Once constructed, instrumentation is fully automated, generally applicable, and reusable across all apps. A user of LetterBomb need not understand it or its implementation details.

LetterBomb’s implementation is currently focused on three vulnerability types. IDOS vulnerabilities, in particular, focus on null checks. Although other forms of IDOS vulnerabilities may exist, null dereference errors are highly common in Android apps [19], making them a highly important target for automatic exploit generation. Furthermore, we selected diverse vulnerability types that can result in severe security or privacy issues (e.g., spoofing or injection of malicious input for XAS and bypassing authentication in the case of FI). Supporting exploit generation for further vulnerability types remains interesting future work.

9 RELATED WORK
A large number of approaches have focused on identifying vulnerability types in Android apps [37]. A number of prominent approaches have relied primarily on static analysis to identify ICC-based vulnerabilities, including ComDroid [16], one of the first major works to characterize ICC-based vulnerabilities in detail; Epicc [34] and its follow-up work IC3 [33], extract information about Intents in a flow-sensitive but not path-sensitive manner; IccTA [29] and COVERT [11], identify vulnerabilities involving combinations of apps rather than only individual apps. Woodpecker [25] analyzes capability leaks, where permissions may be used by an app that does not request it—a form of privilege escalation. FlowDroid [7] conducts a static taint analysis to identify flows or privacy leakages from sensitive Android API sources and sinks. None of these approaches can determine program paths and the Intents needed to execute them, especially since only leverage path-sensitive static analysis. This prevents accurate satisfaction of $\pi_{\text{exploit}}$ in the AEG equation. Furthermore, none are automated exploitation techniques, and as such they have no mechanisms to satisfy $\pi_{\text{exploit}}$.

Another set of approaches rely exclusively on dynamic analysis to identify vulnerabilities. Buzzer [13] fuzzes Android system services to identify vulnerabilities. Mutcher et al. [31] study Android web apps for vulnerabilities. Stowaway [21] dynamically detects permission overprivilege. IntentDroid [27] dynamically explores an app’s Intent interface to identify vulnerabilities. None of these techniques take steps to verify exploitability (i.e., $\pi_{\text{exploit}}$). These approaches inability to analyze non-executed code prevents them from finding a large number of potential ICC-based program paths that may exploit a vulnerability.

A variety of approaches rely upon a combination of static and dynamic analysis to identify vulnerabilities. ContentScope [43] analyzes Content Providers of Android apps to determine cases where those components may have their data leaked or polluted, which occurs when an app manipulates another app’s Content Provider without appropriate permissions or authorization. IPC Inspection [22] is an OS-based defense mechanism that examines an app’s privileges as it receives requests from other apps to prevent privilege escalation attacks. AppAudit [42] focuses on detecting privacy leakage vulnerabilities, but performs limited Intent analysis (e.g., fails to account for a variety of Intent attributes). AppCaulk [38] detects and prevents data leaks through static analysis, dynamic analysis, and the ability to specify policies regarding data leaks. None of these techniques aim to generate exploits.

Certain approaches focus heavily on authentication or authorization issues of Android apps. AuthDroid [41] is a framework that analyzes vulnerable OAuth [26] implementations in an Android app. A few approaches focus on SSL-based vulnerabilities in Android apps. SMV-HUNTER [39] identifies SSL vulnerabilities that may be used for man-in-the-middle (MITM) attacks. Onwuzurike et al. [35] conduct experiments to measure SSL and privacy-leakage vulnerabilities in apps and attempt to attack them using MITM attacks. None of these approaches focus on automatically generating exploits, particularly for the Intent interface of apps.

Two approaches have applied the theory of AEG to Linux binaries, i.e., the original work on AEG [9] and a tool called MAYHEM [15]. The original AEG work targets buffer overflow vulnerabilities and relies on symbolic execution with pre-conditioned inputs, whose goal is to direct execution toward vulnerable program paths. Satisfying $\pi_{\text{exploit}}$ is achieved through formal verification, as opposed to instrumentation and test oracles for LetterBomb. MAYHEM improves upon the original AEG by reducing execution time and memory utilization of AEG, and further applies AEG to format-string vulnerabilities. Neither of these approaches have been applied to Android—whose managed code and different means of memory management make it less susceptible to the types of control-flow attacks that the original AEG and MAYHEM target.

10 CONCLUSION
This paper introduces LetterBomb, an approach for automatic exploit generation for vulnerabilities exposed from an Android app’s Intent-based interface. LetterBomb leverages a highly accurate path-sensitive Intent analysis and Intent generation, app-level and platform-level instrumentation, and software test oracles to generate exploits. LetterBomb can reduce spurious vulnerabilities by 24% to 96% and find vulnerabilities in an app, on average, within 3.12 to 4.30 minutes. Compared to a state-of-the-art detection approach for three ICC-based vulnerabilities, LetterBomb obtains 33%-60% more vulnerabilities at a 6.66 to 7 times faster speed. In the future, we aim to build upon LetterBomb to (1) generate exploits for other vulnerability types and (2) use the generated exploits to aid in automatically fixing Android app vulnerabilities by using automatic program repair [24, 28].

11 ACKNOWLEDGEMENTS
This work was supported in part by awards CCF-1252644, CNS-1629771 and CCF-1618132 from the National Science Foundation, HSHQDC-14-C-B0040 from the Department of Homeland Security, and FA95501610030 from the Air Force Office of Scientific Research.
REFERENCES


