# VulHunter: Toward Discovering Vulnerabilities in Android Applications

A new static-analysis framework helps security analysts detect vulnerable applications. The authors designed an app property graph (APG), conducted graph traversals over APGs, and reduced the manual-verification workload. They implemented the framework in VulHunter and modeled five types of vulnerabilities. Results showed that of 557 randomly collected apps with at least 1 million installations, 375 apps (67.3 percent) had at least one vulnerability.

Chenxiong Qian
Xiapu Luo
Yu Le
Hong Kong Polytechnic University

Guofei Gu
Texas A&M University

With the mobile Internet's prosperity, recent years have witnessed an unprecedented number of Android applications ("apps") published and sold in app markets. However, short development cycles and insufficient security development guidelines have led to many vulnerable apps. After analyzing 2,107 apps from companies on the Forbes Global 2000, HP research recently found that 90 percent of apps are vulnerable (http://zd.net/1FK7I5b).

Motivated by recent research,[1] we propose a new static-analysis framework to facilitate vulnerability discovery for apps by extracting detailed and precise information from apps, easing the identification process, and reducing the manual-verification workload. More precisely, we design a novel data structure called the *app property graph* (APG), which smoothly integrates abstract syntax trees (ASTs), an interprocedure control-flow graph (ICFG), a method-call graph (MCG), and a system dependency graph (SDG) to represent each app. Although the APG is motivated by the code property graph (CPG),[1] the APG differs from the CPG due to the significant difference between apps and C source codes (see the "Related Work in Vulnerability Discovery" sidebar for details). For example, the APG employs the ICFG, MCG, and SDG to characterize the frequent interprocedure and intercomponent communications in apps. The APG also incorporates permissions and other unique features in apps as properties. To ease the identification process, we model common vulnerabilities of apps reported in the Common Vulnerabilities and Exposures (CVE) system as graph traversals and detect vulnerable apps by conducting graph traversals over APGs. Note that each app needs to be processed just once for extracting APG and then we can conduct

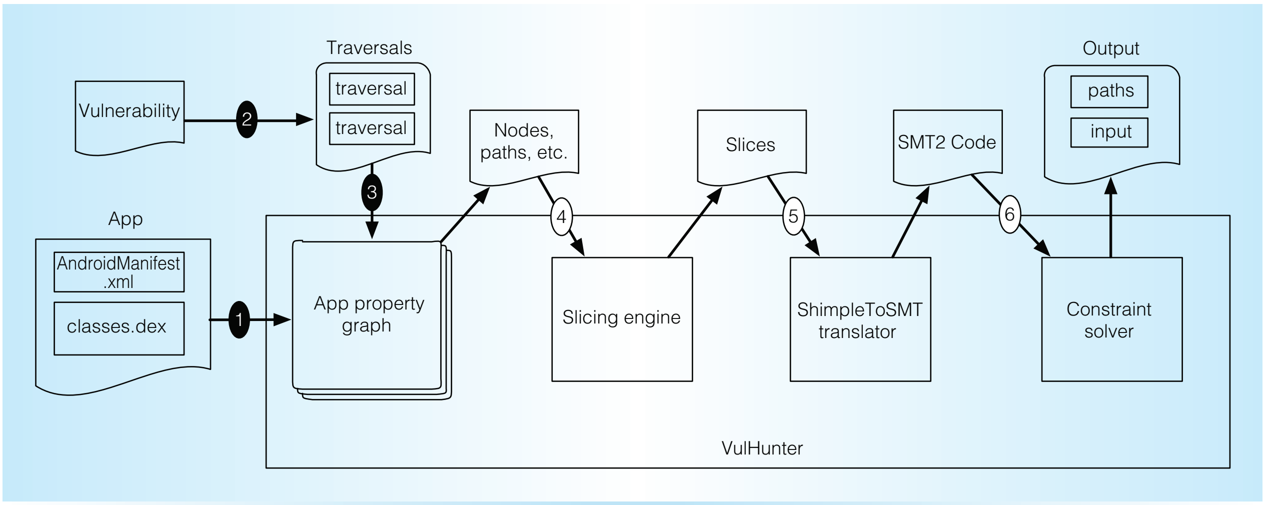


Figure 1. The static-analysis framework. Steps 1 through 3 are necessary, and steps 4 through 6 are optional.

various graph traversals, including those extracted from new vulnerability patterns. Moreover, to reduce the manual-verification workload, we employ symbolic execution to filter out infeasible paths and suggest attack inputs whenever possible.

In creating the APG, we tackled many challenges, including dealing with object references and inheritance in Shimple IR codes and handling Android's event-driven mechanism. Also, we propose an approach to convert Shimple IR codes to SMT-Lib2 codes so that the existing SMT solver can be used. Finally, we implemented the framework in VulHunter with 9,145 lines of Java codes. We modeled five common vulnerabilities as graph traversals and checked the security of 577 popular apps, each of which has more than 1 million installations. The result shows that 375 apps have at least one vulnerability.

## Overview of VulHunter

Figure 1 depicts the major steps in our framework, which has three necessary steps and three optional steps, depending on the type of vulnerability. VulHunter has implemented all these steps. We use a real vulnerable app, GoSMS Pro (com.jb.gosms, v3.72), to illustrate how VulHunter works. This app has an exported service, CellValidateService, which sends a short message service (SMS) according to incoming intents. Because this service does not sanitize incoming intents, an adversary can send a crafted intent for triggering GoSMS to send an SMS to an arbitrary destination address. Figure 2 illustrates the vulnerable code snippet, the corresponding Shimple IR code, and its APG.

### Constructing APG

VulHunter first constructs an app's APG according to its AndroidManifest.xml and classes.dex and then stores it in a graph database, which uses graph structures (including nodes, edges, and properties) to represent and store data. AndroidManifest.xml provides essential information about an app, such as required permissions and intent filters.

We use Soot (http://bit.ly/1veFjB8) to disassemble classes.dex into Shimple IR code (http://bit.ly/1zbB3BM) and then construct the AST, MCG, ICFG, and SDG. These data structures compose an app's APG, denoted as $\zeta = \{N, R, P\}$. $N$ is the set of nodes that denote classes, methods, statements, class fields, instance fields, operands, and operators. Different kinds of nodes have different labels. $R$ is the set of edges (or relationships) that represent connections among classes, methods, and statements, such as the data dependency and control dependency in the SDG, syntax relations in the AST, and control flow in the CFG. $P$ is the set of properties that represent the attributions of nodes and relationships. For instance, the properties of a class node include method signature, name, modifier, argument count, and whether it is an entry of certain component. The

## Related Work in Vulnerability Discovery

Existing research on automatic vulnerability discovery for applications ("apps") usually focuses on several specific types of vulnerabilities because of the undecidability of the generic problem of spotting program vulnerabilities.[1] For example, ComDroid aims at Intent-related issues (that is, unauthorized Intent receipt and Intent spoofing).[2] SMV-Hunter detects SSL and Transport Layer Security (TLS) man-in-the-middle vulnerabilities.[3] ContentScope examines the vulnerabilities of an unprotected content provider.[4] AndroidLeaks uncovers potential private information leakage.[5] Woodpecker targets capability leak vulnerabilities.[6] CHEX discovers component hijacking vulnerabilities.[7] However, these systems' effectiveness and efficiency are usually restricted in practice due to the exponential growth of paths to examine, simplified assumptions, and the limited number of vulnerability patterns.[1,8] Moreover, it is not easy to extend these systems to capture new vulnerabilities, although they share some common components (such as constructing control-flow graphs and dataflow graphs).

Enck et al. used Fortify SCA to conduct a systematic study of 1,100 apps in order to uncover several kinds of vulnerabilities.[9] However, they did not discover vulnerable apps, and it is not clear how SCA processes those apps. We propose a new static-analysis framework to facilitate vulnerability discovery for apps by extracting detailed and precise information from apps and easing the identification process. Moreover, the framework can reduce the manual-verification workload by performing slicing and filtering out infeasible paths. To our knowledge, existing approaches cannot achieve these goals simultaneously. Moreover, defining app property graphs (APGs) and employing graph databases can scale up the vulnerability discovery process.

Researchers are exploring an alternative vulnerability-discovery approach of facilitating security analysts by providing detailed and precise information and expert knowledge. The work closest to our approach is the code property graph (CPG),[1] which combines an abstract syntax tree (AST), control-flow graph (CFG), and program dependency graph (PDG) to represent C source codes and model common vulnerabilities as graph traversals. Therefore, finding potential vulnerabilities is turned into performing graph traversals over CPGs with much better performance in terms of accuracy and flexibility.[1]

Although we also model vulnerabilities as graph traversals and conduct graph traversals to find vulnerable apps, significant differences exist between the two approaches. First, we design APGs specifically for apps, because they have many unique features. Moreover, the APG is based on disassembled Shimple IR code, whereas the CPG is built from C source codes. Second, because apps have frequent interprocedure and intercomponent communications, we employ an interprocedure CFG and a system dependency graph, which consider the dependencies among procedures; the CPG includes only an intraprocedure CFG and a PDG that captures dependencies within a procedure. Third, we use a method call graph and address challenges caused by Android's event-driven nature and its object references and inheritances. Fourth, besides properties of codes, the APG also

properties of dependency relationships record the condition ("true" or "false") of `If-Stmt`, or the lookup value of `TableSwitch-Stmt` and `PackedSwitch-Stmt`.

### Modeling vulnerabilities as graph traversals

Traversals denote the ways we query the graph database according to nodes, edges, and properties. By modeling vulnerabilities as graph traversals, we perform them over APGs to identify vulnerable apps. In our example, finding paths from `onStart` to `sendTextMessage` is modeled with the following graph traversal:

$$(\mathrm{N}_{Method}\{\text{name}:\text{“onStart”}\}) - [\mathrm{R}_{ICFG}]^{+} \rightarrow (\mathrm{N}_{Stmt}\{\text{type}:\text{“invoke”}, \text{callee\_name}: \text{“sendTextMessage”}\}) \quad (1)$$

where $(\mathrm{N}_{Method}\{\text{name}:\text{“onStart”}\})$ denotes method nodes with the name "onStart", $(N_{Stmt}\{\text{type}:\text{“invoke”}, \text{callee\_name}: \text{“sendTextMessage”}\})$ represents statement nodes that call the method "sendTextMessage," and $[\mathrm{R}_{ICFG}]^{+}$ denotes the relationship between nodes and its length is not less than one. Note that we use $(N_{type}\{key : value\})$ to represent nodes with label *type* and property *key* : *value,* and $[R_{type}]^{+}$ to denote the relationships of type *type* with a length not less than one. This traversal will return paths that start at method node `onStart`, end at statement node `sendTextMessage`, and have nodes connected with ICFG relationships. Similarly, to find out whether `sendTextMessage`'s arguments depend on the input from `onStart`, we model it with another graph traversal:

$$(\mathrm{N}_{Stmt}\{\text{type}:\text{“invoke”}, \text{callee\_name}: \text{“sendTextMessage”}\}) - [\mathrm{R}_{SDG_{Data}}]^{+} \rightarrow \left(\mathrm{N}_{Stmt_{Identity}}\{\text{tainted}:\text{true}\}\right) \quad (2)$$

records essential information from the manifest file, because an app cannot run normally without such information. Finally, VulHunter reduces the manual-verification workload by performing slicing, translating Shimple IR codes to SMT-Lib2 codes, and then using Z3-str to filter out infeasible paths.

Because it is a static-analysis system, VulHunter complements dynamic analysis systems, such as TaintDroid[10] and others.[11]

**References**

1. F. Yamaguchi et al., "Modeling and Discovering Vulnerabilities with Code Property Graphs," *Proc. IEEE Symp. Security and Privacy*, 2014, pp. 590–604.
2. E. Chin et al., "Analyzing Inter-application Communication in Android," *Proc. 9th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys 11), 2011, pp. 239–252.
3. D. Sounthiraraj et al., "SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps," *Proc. Network and Distributed System Security Symp.* (NDSS 14), 2014; www.internetsociety.org/doc/smv-hunter-large-scale-automated-detection-ssltls-man-middle-vulnerabilities-android-apps.
4. Y. Zhou and X. Jiang, "Detecting Passive Content Leaks and Pollution in Android Applications," *Proc. Network and Distributed System Security Symp.* (NDSS 13), 2013; http://internetsociety.org/doc/detecting-passive-content-leaks-and-pollution-android-applications.
5. C. Gibler et al., "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," *Proc. 5th Int'l Conf. Trust and Trustworthy Computing* (TRUST 12), 2012, pp. 291–307.
6. M. Grace et al., "Systematic Detection of Capability Leaks in Stock Android Smartphones," *Proc. Network and Distributed System Security Symp.* (NDSS 12), 2012; www.internetsociety.org/systematic-detection-capability-leaks-stock-android-smartphones.
7. L. Lu et al., "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," *Proc. ACM Conf. Computer and Communications Security* (CCS 12), 2012, pp. 229–240.
8. S. Arzt et al., "Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps," *Proc. 35th ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 14), 2014, pp. 259–269.
9. W. Enck et al., "A Study of Android Application Security," *Proc. 20th USENIX Conf. Security* (SEC 11), 2011, article 21.
10. W. Enck et al., "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *Proc. 9th USENIX Conf. Operating Systems Design and Implementation* (OSDI 10), 2010, article 1–6.
11. C. Qian et al., "On Tracking Information Flows through JNI in Android Applications," *Proc. 44th Ann. IEEE/IFIP Int'l Conf. Dependable Systems and Networks* (DSN 14), 2014, pp. 180–191.

where $N_{Stmt_{Identity}}\{tainted : true\}$ denotes identity statement nodes with taint. Note that we use $N_{type_{subtype}}$ to represent nodes that have label *type* and belong to the subtype *subtype*. This traversal ends at identity statement nodes with the property *tainted:true,* meaning that the left value of the identity statement is tainted. In our example, it is the parameter `paramIntent` of `onStart` (see Figure 2).

### Performing traversals over APGs

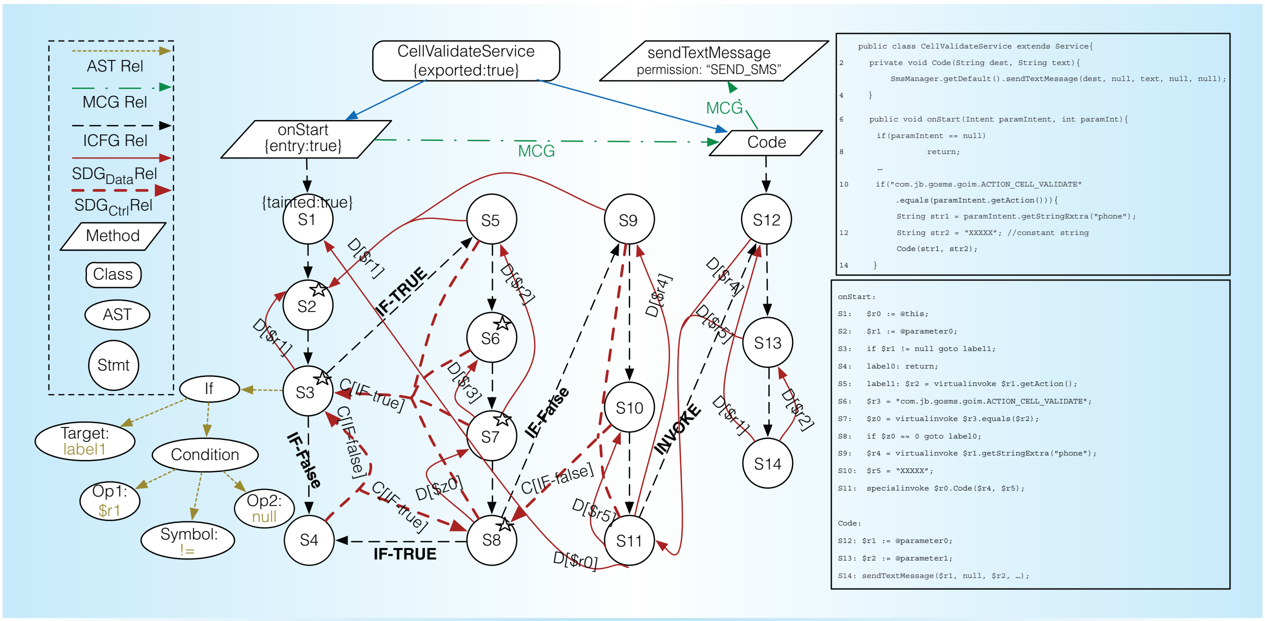
To detect vulnerable apps, we perform traversals over APGs. The output includes nodes and paths with properties. In our example, traversal 1 returns paths connected by ICFG relationships, and traversal 2 returns paths connected by $SDG_{Data}$ relationships, meaning that the input from `onStart` will affect `sendTextMessage`'s arguments.

### Slicing the program

Although some traversals may return many paths, not all of them are feasible. Therefore, we will filter out infeasible paths by checking whether the conditions on these paths can be fulfilled or not. More precisely, we turn those conditions into constraints using symbolic execution,[2] and then decide whether they are satisfiable through satisfiability modulo theories (SMT) solvers. Before doing this, we conduct program slicing (http://bit.ly/1ycjK5f) to extract statements that affect the values in constraints (that is, `If-Stmt`, `PackedSwitch-Stmt`, and `Table Switch-Stmt`). For example, by performing slicing on paths returned by traversal 1, we get the statements with a star in the top left corner in Figure 2.

### Translating Shimple IR codes to the SMT-Lib2 language

The SMT-Lib2 language is used to describe SMT problems. After translating Shimple IR codes in each slice to SMT-Lib2 language, we convert the constraints into a



```
public class CellValidateService extends Service{
    private void Code(String dest, String text){
        SmsManager.getDefault().sendTextMessage(dest, null, text, null, null);
    }

    public void onStart(Intent paramIntent, int paramInt){
      if(paramIntent == null)
                return;
      …
      if("com.jb.gosms.goim.ACTION_CELL_VALIDATE"
          .equals(paramIntent.getAction())){
          String str1 = paramIntent.getStringExtra("phone");
          String str2 = "XXXXX"; //constant string
          Code(str1, str2);
      }
```

```
onStart:
S1:   $r0 := @this;
S2:   $r1 := @parameter0;
S3:   if $r1 != null goto label1;
S4:   label0: return;
S5:   label1: $r2 = virtualinvoke $r1.getAction();
S6:   $r3 = "com.jb.gosms.goim.ACTION_CELL_VALIDATE";
S7:   $z0 = virtualinvoke $r3.equals($r2);
S8:   if $z0 == 0 goto label0;
S9:   $r4 = virtualinvoke $r1.getStringExtra("phone");
S10:  $r5 = "XXXXX";
S11:  specialinvoke $r0.Code($r4, $r5);

Code:
S12: $r1 := @parameter0;
S13: $r2 := @parameter1;
S14: sendTextMessage($r1, null, $r2, …);
```

Figure 2. The vulnerable code and its Shimple IR code and part of the corresponding app property graph (APG) in GoSMS Pro v3.72. This app exports a service named CellValidateService, which will send a short message service (SMS) according to an incoming intent. Because this service does not sanitize incoming intents, an adversary can send a crafted intent to force this app to send an SMS to an arbitrary destination.

combination of binary-valued functions and then solve them using SMT solvers.

### Solving constraints

We feed the SMT-Lib2 codes to Z3-str,[3] an extension of Z3 SMT, for computing the constraints and generating inputs for feasible paths whenever possible, because Z3-str supports string types and can solve constraints with string operations.

## The APG

We use an APG, comprising the AST, ICFG, MCG, and SDG, to characterize each app.

### AST

As a basic data structure in program analysis, an AST eases the traversals over the APG and the translation from Shimple IR codes to SMT-Lib2, because it records each statement's operands and operators and keeps the structures of Shimple IR codes.

### MCG

The MCG records the invocation relationships among methods. If a node represents a method that requires permissions, we set the permission property of that node. With MCG, we can quickly decide the feasibility for a method to trigger certain operations. For example, according to the MCG in Figure 2, we know that `onStart` may trigger `sendTextMessage`.

We tackle two challenges when constructing MCGs. The first one results from the commonly used object references and inheritance in Shimple. For example, it is difficult to determine the concrete class to which an object reference points, especially when handling `InterfaceInvoke-Stmt`. To address this issue, we conduct points-to analysis using Spark[4] and maintain the class inheritance hierarchy. If an object reference cannot be resolved by Spark, we consider all possible subclasses.

The second challenge is due to Android's event-driven nature. There are two major types of event handling in Android: listeners to process events, and callbacks. For the former, we redirect invocations (for example, `setOnClickListener`) to the relevant objects' callbacks (`onClick`). For the latter, we follow Grace et al. to connect methods according to well-defined semantics (for example, in java.lang.Thread, methods `start` and `run` are correlated).[5] However, it is nontrivial to handle callbacks involved in the intercomponent communication (that is, those through `Intent`). To approach this issue, we use Epicc (http://siis.cse.psu.edu/epicc) to map invocations (such as `startActivity`) to callbacks (`Activity.onCreate`).

### CFG

The CFG depicts the execution flow of statements. The APG adopts ICFGs because of the frequent communications between subroutines and components in apps. There are five types of branch statements in Shimple IR. For the conditional statements `If-Stmt` and `Switch-Stmt`, we add a property about the condition on those relationships. For example, in Figure 2, if the statement s3's condition is satisfied (that is, true), statement s5 will be executed. Otherwise, statement s4 will be executed. Therefore, we set *true* ($s3 \rightarrow s5$) or *false* ($s3 \rightarrow s4$) to the corresponding relationships between `If-Stmt` and its target statements. We connect the direct jump statement `Goto-Stmt` to the target statement. When handling `Invoke-Stmt`, if the invocation is not a method in the Android SDK, we link it to the first statement of target method ($s11 \rightarrow s12$) and also add relationships between the target method's exit statements and the `Invoke-Stmt`; otherwise, we do nothing. For exception-handling statements (such as `Throw-Stmt` and `Catch-Stmt`), we connect each statement in the try block to the first statement in the exception handler.

### SDG

The SDG is an interprocedural extension of the PDGs for modeling statements' dependencies that can be either control or data dependent. A data dependency between statements $s1$ and $s2$ indicates that one variable defined at $s1$ is used at $s2$. A control dependency shows the influence of a statement on other statements. In an SDG, nodes represent statements, and edges correspond to dependencies.

We decide data dependencies on the basis of def-use chains, and we implement the algorithm[6] to determine control dependencies by constructing dominator trees. As Figure 2 shows, data dependency has property `D[var]` , meaning *var* is defined at one statement and used at another. Control dependency has property `C[condition]` , indicating that the dependency exists when the predicate statement satisfies the condition (true or false of `If-Stmt`, or lookup value of `Switch-Stmt`). In particular, to handle the interprocedural data dependency, we link `Identity-Stmt` of a callee method to `Invoke-Stmt` in a relevant caller, and we set the property value *var* to the argument name at `Invoke-Stmt`.

## Modeling common vulnerabilities as graph traversals

We consider three types of graph traversals for modeling common vulnerabilities: those being detected through syntactical information, those requiring control-flow information, and those needing additional dataflow information.

### Definitions

We use the following symbols in our traversals:

- $MATCH^{p}_{label}$ represents matching nodes with label `label` and properties *p*,
- $ARG(N)_i$ indicates traversing from `Invoke-Stmt` node *N* to get its *i*th argument, and
- $N1 - [R^{p}_{type}]^{len} \rightarrow N2$ denotes a path from node *N*1 to node *N*2. The path is connected by relationship type with length `len`, which can be omitted if it equals 1.

### Syntax-level vulnerability

We use a real example (http://bit.ly/1rRoozD) to explain such vulnerability. Skype v1.0.0.831 creates private files using the method `openFileOutput` with parameters `MODE_WORLD_READABLE` and `MODE_WORLD_WRITABLE`. Therefore, any other apps can access Skype's private files, causing unauthorized information leakage and manipulation. There are two necessary conditions for such vulnerability: method `openFileOutput` is called, and its second argument is `MODE_WORLD_READABLE`, `MODE_WORLD_WRITABLE`, or both. To model these two conditions, we design the following traversal:

$$MATCH^{p2}_{Ast} \circ ARG(N)_2 \circ MATCH^{p1}_{Stmt} \quad \text{(A)}$$

where $p1$ contains properties for selecting `Invoke-Stmt` or `Assign-Stmt` whose right value is an invocation expression. $p2$ requires that the augment's value is `MODE_WORLD_READABLE`, `MODE_WORLD_WRITABLE`, or both. Traversal A first matches statements invoking method `openFileOutput`, then visits the nodes of their second argument, and finally selects those whose value is `MODE_WORLD_READABLE`, `MODE_WORLD_WRITABLE`, or both.

Note that traversal A cannot capture the cases when the second augment of `openFileOutput` is a variable rather than a constant. Therefore, extra information, such as dataflow information, is needed.

### Control-flow-level vulnerability

Because some vulnerabilities will be triggered through several statements, control-flow information is needed to identify them. We define the following traversal for capability vulnerabilities:

$$MATCH^{p1}_{Method} - [R_{MCG}]^{+} \rightarrow MATCH^{p2}_{Method} \quad \text{(B1)}$$

where $p1$ selects source methods, such as the entry methods of exported components, and $p2$ chooses sink methods, such as sensitive methods that require permissions. Also, $-[R_{MCG}]^{+} \rightarrow$ means there is one or more MCG relationships. Hence, traversal B1 returns paths from entries to important calls, connected with MCG relationships. Because traversal B1 describes only whether the vulnerability exists at the method level, we define the following traversal to retrieve control-flow paths with statement nodes if traversal B1 does not return nil:

$$MATCH^{p1}_{Method} - [R_{ICFG}]^{+} \rightarrow MATCH^{p2}_{Stmt} \quad \text{(B2)}$$

where $p1$ selects entry methods of exported components, and $p2$ chooses `Invoke-Stmt`, which invokes permission-protected methods. As Figure 2 shows, traversal B2 returns the path ($s1$, $s2$, $s3$, $s5$, $s6$, $s7$, $s8$, $s9$, $s10$, $s11$, $s12$, $s13$, $s14$) from the method node `onStart` to the `Invoke-Stmt` node that calls `sendTextMessage`.

### Dataflow-level vulnerability

Some vulnerabilities can be triggered only if an attacker can inject proper data. Because dataflow information is necessary for discovering such vulnerabilities, we construct the following traversal:

$$MATCH^{p_{sink}}_{Stmt} - [R^{V}_{SDG_{Data}}] \rightarrow N - [R_{SDG_{Data}}]^{*} \rightarrow MATCH^{p_{source}}_{Stmt} \quad \text{(C)}$$

where nodes with properties $p_{source}$ are sources, such as calling methods to get tainted sensitive information. Nodes with properties $p_{sink}$ are sinks, such as statements that could leak information (for example, `Log.d` or `FileOutputStream.write`). $N1 - \left[R^{V}_{SDG_{Data}}\right] \rightarrow N2$ means that node $N1$ is linked to node $N2$ with an SDG data dependency relationship, and set $V$ contains variables used at $N1$ that are of interest. For example, for the statement `sendTextMessage` ($s14$) in Figure 2, $V$ includes $\{r1, r2\}$.

Although traversal C returns paths showing how sensitive data produced at sources is propagated to sinks, not all paths are feasible.

## Solving constraints

To remove infeasible paths, we collect constraints on the path and decide whether they can be satisfied. For control-flow paths, we extract predicate statement nodes on the paths. For dataflow paths, we consider more statement nodes that could affect the path.

To use an SMT solver to decide whether a path is feasible and generate appropriate input if possible, we first convert Shimple IR code to SMT-LIB2 code. However, existing SMT solvers lack types. Hence, we map the types defined in Shimple IR to the types supported by Z3-str—more precisely, we map the Shimple primitive types to the Z3-str primitive types. We map the reference type to Z3-str's String type. Although this type mapping is coarse, it leads to a low failure rate in Z3-str's execution, and in our

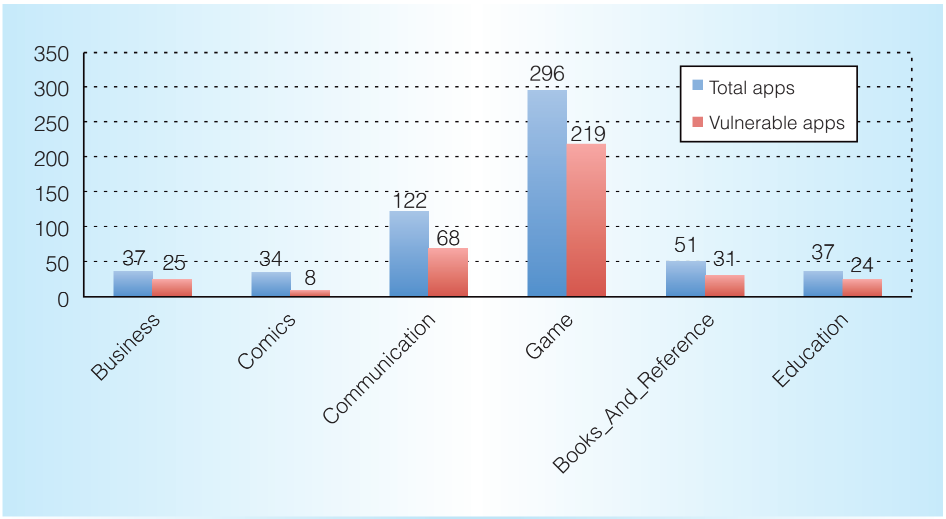


Figure 3. We randomly selected 577 popular apps, each of which has at least 1 million installations, and checked whether they have one or more common vulnerabilities. The figure shows their category distribution.

experiments all SMT-Lib2 code generated by our system can be successfully executed by Z3-str. We convert each `Invoke-Stmt` to an SMT-Lib2 function declaration statement if the function has not been declared before, and it is followed by an assertion statement to make sure the return value is equal to that invocation. We pass *this* object as the first argument if it is an instance object method call.

## Evaluation

We implemented VulHunter with 9,145 lines of Java codes, excluding third-party libraries, and modeled five common vulnerabilities in apps according to CVE reports from January 2011 to April 2014. Then, we applied VulHunter to scan 577 popular apps downloaded from Google Play in April 2014. All these apps have more than 1 million installations; Figure 3 shows the category distribution. We found that 375 apps had at least one vulnerability.

### Modeling common vulnerabilities

We analyzed all CVE reports related to Android apps from January 2011 to April 2014 and modeled as graph traversals five common vulnerabilities (see Table 1) that account for 72.5 percent of all cases. The five common vulnerabilities include the following:

**Table 1. Five common vulnerabilities. For each vulnerability, we give both the number of Common Vulnerabilities and Exposures reports in each year and the number of vulnerable apps within the 577 apps under examination.**

| Vulnerability type | 2011 | 2012 | 2013 | 2014 | Vulnerable apps |
|---|---|---|---|---|---|
| Capability leak | 1 | 63 | 3 | 6 | 4 |
| Content provider directory traversal | 0 | 0 | 0 | 9 | 3 |
| X509TrustManager implemented improperly | 1 | 7 | 2 | 6 | 337 |
| Public file access permission | 3 | 6 | 6 | 3 | 133 |
| Log sensitive information | 0 | 2 | 1 | 2 | 20 |

- *Capability leak.*[5] An app exports its component(s) without sanitizing the intent from other apps, and therefore it could leak its capabilities to other apps, which gain extra capabilities without requesting the corresponding permissions. To fix this vulnerability, an app can check the source of intent or not export its component(s).
- *Content provider directory traversal* (http://bit.ly/1FILcrB). An app exports its content providers without properly canonicalizing the URI to methods like `openFile` or `open`

AssetFile, so that other apps can access arbitrary files. To solve this vulnerability, an app should check the path.

- *X509TrustManager implemented improperly.* An app uses or inherits class X509TrustManager but the methods `checkClientTrusted()` and `checkServerTrusted()` are overridden with a blank implementation (http://bit.ly/1FILcrB) or the default implementation (http://bit.ly/1yzoIrM), so that it does not verify the certificate. To avoid this problem, an app should follow best practices to implement those methods.[7]
- *Public file access permission* (http://bit.ly/12klYme). An app creates files with permission `MODE_WORLD_READABLE`, `MODE_WORLD_WRITABLE`, or both, so that other apps can read or modify the files.
- *Log sensitive information* (http://bit.ly/1yzp0Ph). An app logs sensitive information (such as the device ID or location), so that other apps can retrieve them by declaring `READ_LOGS` permission.

We will model the remaining vulnerabilities in future work.

## Discovering vulnerable apps

We performed the five graph traversals over the APGs from the 577 apps, and Table 1 gives the results.

*Capability leak.* We selected only those functions that send SMS as sink functions, and we detected four apps that let other apps send SMS through themselves. More sinks will be included in future work. After manual analysis, we confirmed one app and determined why the other three apps could not be triggered.

The verified app is an email client called TouchDown (com.nitrodesk.droid20.nitroid, version 8.4.00086). It lets users wipe data remotely through an SMS. However, it checks the SMS's content without verifying the source of SMS, and sends a confirmation SMS after wiping out data. An adversary can send a malicious SMS to wipe out data on target devices. VulHunter cannot generate input to trigger such an attack, because constraints on paths depend on data stored in the app's SQLite databases and preference files, which are not accessible during static analysis. However, VulHunter provides enough information for us to conduct manual analysis.

The other three apps use the same payment framework that sends an SMS upon receipt of a specific SMS. Similar to TouchDown, they do not verify the source of the incoming SMS. After manually analyzing these apps, we found that two apps invoked the vulnerable codes but did not activate that function. In future work, we will enhance VulHunter with dead-code detection to eliminate such false positives. Another app supports only telecommunications service providers in Taiwan, so we could not check it.

*Content provider directory traversal.* Three apps export their content providers, such that other apps can access arbitrary private files. We manually confirmed these vulnerabilities.

*X509TrustManager implemented improperly.* This was a common vulnerability, as 337 apps used X509TrustManager without properly implementing those methods, and therefore were vulnerable to the SSL and TLS man-in-the-middle (MITM) attack.[7] We randomly selected five apps, manually verified them, and located the vulnerable codes (that is, the default TrustManager was overrode with a blank implementation.). When we used tapioca (http://bit.ly/1r8gqAL) to launch the MITM attacks, we found that two apps could be attacked. For the remaining three apps, one could not be triggered by GUI operations, and the other two contained dead codes. In future work, we will add dead-code detection to reduce false positives, and we will employ dynamic analysis tools such as SMV-Hunter[7] to automatically verify them.

*Public file access permission.* This vulnerability occurred in 133 apps. We randomly selected 10 apps from them, manually verified these apps, and confirmed that all created publicly readable or writable files.

*Log sensitive information.* We regard the device ID, latitude, and longitude as sensitive information, and found 20 vulnerable apps, all of which have been manually verified without false positives.

We implemented our static-analysis framework in VulHunter and modeled five types of vulnerabilities as graph traversals. Checking 557 popular apps, we found that 375 (67.3 percent) of apps had at least one vulnerability. Our future work is twofold. One aim is to further improve the framework, such as adding dead-code detection to reduce false positives, proposing formal approaches to model new vulnerabilities using APGs, and optimizing queries according to vulnerability patterns. The other aim is to integrate it with dynamic analysis for verifying suspicious apps automatically. MICRO

## Acknowledgments

This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the PolyU Research Grant (G-UA3X), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province (2013GDDSIPL-04).

## References

1. F. Yamaguchi et al., "Modeling and Discovering Vulnerabilities with Code Property Graphs," *Proc. IEEE Symp. Security and Privacy*, 2014, pp. 590–604.
2. J. King, "Symbolic Execution and Program Testing," *Comm. ACM*, vol. 19, no. 7, 1976, pp. 385–394.
3. Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-Based String Solver for Web Application Analysis," *Proc. 9th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 13), 2013, pp. 114–124.
4. O. Lhotak and L.J. Hendren, "Scaling Java Points-to Analysis Using Spark," *Proc. 12th Int'l Conf. Compiler Construction* (CC 03), 2003, pp. 153–169.
5. M. Grace et al., "Systematic Detection of Capability Leaks in Stock Android Smartphones," *Proc. Network and Distributed System Security Symp.* (NDSS 12), 2012; www.internetsociety.org/systematic-detection-capability-leaks-stock-android-smartphones.
6. T. Lengauer and R.E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Trans. Programming Languages and Systems*, vol. 1, no. 1, 1979, pp. 121–141.
7. D. Sounthiraraj et al., "SMV-Hunter: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps," *Proc. Network and Distributed System Security Symp.* (NDSS 14), 2014; www.internetsociety.org/doc/smv-hunter-large-scale-automated-detection-ssltls-man-middle-vulnerabilities-android-apps.

**Chenxiong Qian** is a research assistant in the Department of Computing at the Hong Kong Polytechnic University. His research focuses on security and privacy with an emphasis on mobile security. Qian has a BEng in software engineering from Nanjing University. Contact him at cscqian@comp.polyu.edu.hk.

**Xiapu Luo** is a research assistant professor in the Department of Computing and an associate researcher at the Shenzhen Research Institute at the Hong Kong Polytechnic University. His research focuses on smartphone security, network security and privacy, and Internet measurement. Luo has a PhD in computer science from the Hong Kong Polytechnic University. He is the corresponding author; contact him at csxluo@comp.polyu.edu.hk.

**Yu Le** is a research assistant in the Department of Computing at the Hong Kong Polytechnic University. His research focuses on security and privacy with an emphasis on mobile security. Le has a BEng in information security from Nanjing University of Posts and Telecommunications. Contact him at cslyu@comp.polyu.edu.hk.

**Guofei Gu** is an associate professor in the Department of Computer Science and Engineering at Texas A&M University. His research interests include network and system security, social Web security, and cloud and software-defined networking (SDN/OpenFlow) security. Gu has a PhD in computer science from the Georgia Institute of Technology. Contact him at guofei@cse.tamu.edu.