SurgeScan: Enforcing Security Policies on Untrusted Third-Party Android Libraries

Jonathan Vronsky*, Ryan Stevens†, Hao Chen‡
Department of Computer Science, University of California, Davis
*Email: *jvronsky@ucdavis.edu. †rcstevens@ucdavis.edu, ‡chen@ucdavis.edu,*

***Abstract*—Many Android apps include third-party libraries for advertising, payment, social media, etc. However, since the library code runs with the same privilege as the app code, the app developer has to either trust the library, a potential security risk, or refrain from using untrusted libraries. We designed and implemented SURGESCAN, a framework for specifying and enforcing security policies on untrusted third-party code. We call this third-party code *plugins*, as SURGESCAN supports both statically and dynamically loaded code. SURGESCAN consists of a static analysis component and code rewriting component. To use SURGESCAN, the app developer selects a security policy that declares security-sensitive methods in the Android API. Then, using static analysis, SURGESCAN finds all the Android API calls in the plugin binary that may reach those security-sensitive methods, and generates AspectJ code for enforcing the security policy on those API calls. Next, SURGESCAN runs AspectJ to weave the policy into the plugin. After that, the app can safely load the plugin. SURGESCAN requires no modification to the OS and incurs negligible runtime overhead. We describe our algorithms for achieving high accuracy in our static analysis. To evaluate SURGESCAN, we designed policies on network and sensor access and applied them to open source apps. We demonstrated various use scenarios for SURGESCAN, including securing distributed network measurement, securing ad libraries, controlling UI and screen estate, and patching applications.**

1. Introduction

Android developers rely on third-party libraries for ads, analytics, payment, and social media. However, not all these libraries are trustworthy. For example, some ad libraries were found to exfiltrate user data silently when included in an application [23, 11]. This is possible because library code runs with the same privileges as its embedding application. Researchers proposed approaches to contain untrusted Android code. Layercake ran the library code in a separate process, but it required platform modifications [21]. AndroidLeaks checked permissions required by an app but was unable to prevent dangerous calls [10]. Retroskeleton provided a framework to automatically rewrite security-sensitive method calls [6]. However, Retroskeleton did not provide as in depth analysis to find all the security-senstive methods. We present SURGESCAN, a framework for enforcing security policies on untrusted third-party code. SURGESCAN consists of two components: a static analysis tool for implementing the developer's security policies by analyzing the Android API implementation, and an instrumentation tool for weaving the policy into third-party Java bytecode. We refer to the third-party code as a *plugin*, since an app can either statically link it or dynamically load it. SURGESCAN allows the app developer to place his trust in his security policy instead in the authors of the untrusted plugin.

SURGESCAN makes the policy writing as trouble free as possible by finding dangerous code paths automatically from APIs that are discovered to be risky. The policy writer provides a *policy declaration*, which specifies security-sensitive Android API methods (*sinks*). Any method visible internally to the Android API may be used as a sink, notably low-level calls to the Dalvik virtual machine and *libcore*, which implement most of API's functionality. SURGESCAN identifies code paths through both the Android API and the plugin to find all relevant callsites. Then, SURGESCAN's static analysis tool uses a bottom-up data flow analysis to find all the methods (*super-sinks*) and the subset of their parameters (*data sources*) that will eventually flow into the sinks. For each super-sink, SURGESCAN automatically generates an AspectJ file (*policy implementation*) to implement a default policy, which denies the call, and to annotate the data sources. The policy writer may customize the AspectJ file, e.g., to control access based on the values in the data sources. Finally, SURGESCAN identifies where in each code path it can inject a security policy, respecting the fact that the Android API cannot be modified. Developers may share a security policy if their apps have similar *plugin* usage and security concerns.

Since SURGESCAN requires no modification to the Android framework, it enables new paradigms of code sharing and distribution in the Android ecosystem. For example, it allows apps to securely run ad code that is either statically linked in the app or dynamically loaded at runtime. Market owners can use SURGESCAN to patch a vulnerable library in all of its apps automatically without the help from the app developers (subsection 6.1).

We evaluated SURGESCAN by specifying a *policy declaration* for restricting network and sensor access. Then, SURGESCAN automatically generated the default policy implementation, which we then manually revised for finer access control. Finally, SURGESCAN weaved the policy into open-source applications. We verified that the policy enforcement worked as expected, and observed that the memory and run time overhead of policy enforcement was negligible.

SURGESCAN has the following advantages:

- *Deployability*. It modifies neither the Android platform nor the app code. Instead, it instruments only the plugin, which the app either links statically or loads dynamically. This improves its deployability.
- *Precision and Customization*. It automatically generates AspectJ code to implement the security policy, and weaves the policy into the plugin. Moreover, it uses an accurate bottom-up data flow analysis to identify the data sources, which the policy writer can use to customize the policy in AspectJ to implement fine-grained access control.
- *Expressiveness*. It provides a more expressive security model than Android's permission model. For example, the policy could grant the *INTERNET* permission while blocking certain Internet domains (See Listing 4). We demonstrated that SURGESCAN could be used in many scenarios, such as securely running distributed network measurement and ad libraries, controlling the UI and screen estate of libraries, and patching a vulnerable library in many apps without the help from the app developers.

2. Background

Android is a smartphone platform based on the Linux kernel. Android sandboxes apps by installing each app under a separate Linux user. Android requires each app to specify *permissions*, which determine which security-sensitive information or action the app may access.

In order to enforce the policies we used AspectJ [14]. AspectJ is an aspect oriented programming extension to Java. The language allows for specifying rules to be executed at a particular point in the code.

The control flow analysis of our code is done by Soot [12] and platformed developed on top of soot.

3. Goals

SURGESCAN is a framework for Android app developers to enforce access control policies on untrusted *plugins*, which are third-party libraries to be included in the app. At a high level, SURGESCAN operates in three steps: The policy writer declares an access control policy, SURGESCAN automatically generates code implementing the policy and weaves the code into the plugin, the app either links the instrumented plugin statically or loads it at runtime.

3.1. Motivation

Android apps use many external libraries. SURGESCAN provides a customizable, portable, accurate approach to modify untrusted libraries to make them safer without modifying the Android framework. Use scenarios include: filtering Internet communications, controlling access to sensitive data sources (e.g., sensors or GPS) and others described in more details in Section 6.1

3.2. Threat model

SURGESCAN involves three parties: app developer, policy writer, and plugin developer. We assume that the app developer and the policy writer are trusted. Since the developer wants to protect her app and the policy writer wants to satisfy the developer requirement. By contrast, we assume that the plugin developer is untrusted. The goal of SURGESCAN is to prevent the plugin from violating the security policy.

3.3. Non-goals

SURGESCAN provides a framework for specifying and automatically enforcing security policies. However, it does not provide a comprehensive set of security policies for all plugins, because those depend on the specific requirements by individual apps. We will showcase a few policies, notably for securing network and sensor access. One can easily extend these policies to restrict the plugin's use of permissions.

SURGESCAN does not handle code invoked via Java reflection or loaded dynamically by the plugins, because AspectJ can only intercept call sites that are visible in the code at weaving time. However, it is easy to intercept and block Java reflection and dynamic code loading in the policy because they rely on a few specific API calls.

SURGESCAN does not provide its own call-graph construction algorithm and instead is built on top of *soot*. As a result SURGESCAN inherits *soot*'s limitation.

4. Design

SURGESCAN takes the following inputs:

- Plugin, an untrusted library (as Java bytecode) to run in the app, and all the libraries used by the plugin.
- Compiled full *android.jar* binary for the target device and Android API specification (as Java byte code, also named *android.jar*). We can extract the former binary from the emulator and the latter is provided by Android.
- Policy. The policy writer may specify his security policy in two stages. First, he specifies a *policy declaration* containing (1) entry points into the plugin from the app, and (2) sinks, which are Android API methods that are security sensitive and that the developer wishes to interpose on. SURGESCAN analyzes the policy declaration, the plugin, and *android.jar*, and then outputs a default *policy implementation*, which denies all the Android API function calls that eventually flow into the sinks. If the developer wishes for finer-grained access control than simply denial, he may add arbitrary code to the policy implementation.

SURGESCAN outputs the plugin with the policy implementation weaved into it. SURGESCAN does not take into account the operating system files and hence cannot block native code calls. Moreover, because reflection code might not be available in static time SURGESCAN cannot block code calls through reflection.

4.1. Stages

Figure 1 shows the stages when using SURGESCAN.

4.1.1. Find Super-Sinks by Bottom Up Flow Analysis

As mentioned above, a sink is an Android API method that consumes data and therefore may cause security risks. A naive approach is to look for all the sinks in the plugin's *jar* file. However, sometimes a plugin invokes a method in the Android SDK and this method invokes a sink, but the SDK bytecode isn't in the plugin's *jar* file because it is dynamically linked. As a result, SURGESCAN cannot

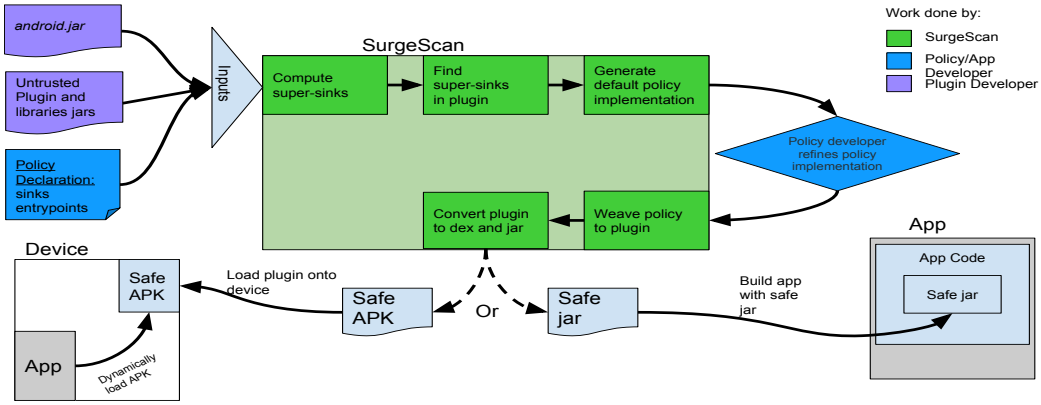


Figure 1: Oveview of the SURGESCAN workflow.

apply the security policy, as the bytecode is unavailable at compile time. One solution would be to modify the Android platform using SURGESCAN. However, we want to avoid such changes as they hurt the portability and flexibility.

To overcome this problem, we developed a *bottom up flow analysis* algorithm to find *super-sinks*. The analysis is built on top of *soot*. SURGESCAN currently uses *soot* SPARK [15, 8] to compute call graphs. The *bottom up flow analysis* we developed for the data flow analysis portion uses super-sinks, which are method calls that may eventually reach a sink. Each super-sink contains a set of parameters and fields that it might leak. To find super-sinks, the analysis begins at each sink specified by the developer and traverses upwards towards all the methods that might eventually call a sink. The call-graph required for such analysis is initiated in the entrypoint defined by the developer. We assume it should be easier for the policy writer to find entrypoints as they are the locations in the code where the app would access the plugin. The algorithm finds pairs of caller and callee methods and computes the data flow from the parameters of the callers into the parameters of the callee. When SURGESCAN finds a flow from caller values to a super-sink parameter that may leak data, it marks those values as possible leaks too. We refer to these parameters as *marked parameters*. The algorithm also track fuzzy flows, which are flows that are not yet a leak but could become one if an already computed parameter becomes marked. subsection 5.1 will explain this algorithm in detail, and Listing 2 provides an example. At the end of the analysis, the algorithm adds all the fuzzy flows that became leaks to the set of super-sinks. Algorithm 1 shows the pseudo code. Since it computes each caller-callee pair separately, we can trivially modify it to compute all the pairs in parallel.

Algorithm 1 Bottom Up Data Flow

```
1: procedure BOTTOMUPDATAFLOW(S, E, P)                ▷ S-sinks, E-entrypoints, P-binaries
2:     callGraph ← constructCallGraph(E, P)
3:     callerCalleePairs ← findPairsWithBFS(S, callGraph)
4:     for each caller, callee ∈ callerCalleePairs do
5:         flowResults ← intraProceduralFlow(caller, callee)
6:         for each result ∈ flowResults do
7:             if result flows into marked parameter then
8:                 callee.mark(result.flowingVariable)
9:             else
10:                callee.addFuzzyFlow(result.flowingVariable, callee)
11:            end if
12:        end for
13:    end for
14:    Set superSinks ← ∅
15:    for each caller, callee ∈ callerCalleePairs do
16:        superSinks.add(caller)
17:        superSinks.add(callee)
18:    end for
19:    foundNewFlow ← true
20:    while foundNewFlow do
21:        foundNewFlow ← false
22:        for each superSink1 ∈ superSinks do
23:            for each superSink2 ∈ superSinks do
24:                if superSink1 has fuzzy flow to superSink2 then
25:                    superSink1.mark(fuzzyFlowParameter)
26:                    remove fuzzy flow from superSink1
27:                    foundNewFlow ← true
28:                end if
29:            end for
30:        end for
31:    end while
32: end procedure
```

4.1.2. Apply the policy

We weave the security policy on each super-sink in the plugin. First, for each super-sink, SURGESCAN generates access control code as an AspectJ file. The file consists of the name of the super-sink, pseudo-dynamic analysis code (subsubsection 4.1.3), potentially risky variables (in comments), and a default action of denying the call by throwing a security exception. While this default action may cause unexpected behavior, it is easy to detect during runtime. Moreover, the developer can change this default action by modifying the AspectJ file. After the policy modification, SURGESCAN weaves the AspectJ code into the plugin. Finally, SURGESCAN converts the weaved plugin into dex bytecode and then a library jar. The developer may link the library jar statically or load it dynamically.

4.1.3. Pseudo Dynamic Analysis

Defining good policies for abstract methods is challenging. Abstract methods are unresolved at compile time and so cannot be statically analyzed. To handle this issue, we use AspectJ's abstract method handling, which allows us to specify one policy for an abstract method and all of its implementers. In the policy implementation, we use dynamic cast to delegate to the best specific policy for the specific class. This provides more functionality and flexibility to the developer. For example, a developer can specify different policies for different concrete implementations of the abstract class `URLConnection`.

4.1.4. Find Super-Sinks in Plugin

After the bottom-up flow analysis outputs all the super-sinks, SURGESCAN looks for the super-sinks in the plugin

to weave the security policy into. We scan only Android API available to the app.

5. Implementation

5.1. Computing Super-Sinks

Super-sinks are API methods that can potentially leak data into one of the sinks defined by the policy writer in the policy declaration. The use of super-sinks allows us to propagate sinks up to Android API used by the app. SURGESCAN computes super-sinks on a per plugin and policy declaration basis. The reason behind re-computing super-sinks for each policy is that different policies may have a different set of sinks, which may result in a different set of super-sinks. One could make a general policy for a set of sinks using all the *public* methods in the Android API as entry points, but this implementation would reduce the analysis's accuracy, because it may include extra methods for objects that are irrelevant to the plugin. Our analysis uses SPARK, a points-to analysis framework provided by soot [15, 8]. The advantage of using a points-to analysis is that only objects that can be created in the code will be included in the call graph. Reducing entry points minimizes the call graph, which prevents the analysis from expanding to irrelevant objects and makes the analysis more granular. Listing 1 shows an example where our analysis will give a more accurate result.

```
// This class is in the plugin.
public class PointsToExample {
  public void A(int a) {
    BaseClass c = new FirstClass(); C(c, a);
  }
  public void B(int b) {
    BaseClass c = new SecondClass(); c.foo(b);
  }
  public void C(BaseClass c, int c) {
    c.foo(c)
  }
}
// The following two classes are in the Android SDK.
public class FirstClass extends BaseClass {
  @Override public foo(int f) { /*nothing.*/ }
}
public class SecondClass extends BaseClass {
  @Override public foo(int f) { sink(f); }
}
```

Listing 1: Advantages of Points-To analysis

In the example, if we assume that method A is the only entry point, then method C will never be tainted, as the only $BaseClass$ instance is of type $FirstClass$ but none of its methods ever reaches a sink. However, if we also include B as an entry point, then once we get to method C we will have to taint it as it calls the abstract method `BaseClass.foo()`. At this point our analysis will include `foo` as a super-sink as type $SecondClass$ consists a sink within method `foo`. Therefor we could not verify the origin of the $BaseClass$ instance. Re-computing the super-sinks for each plugin prevents this issue as we construct a call graph only to the relevant methods. Another issue in this example is that $BaseClass$ has two implementations. Using SPARK as our call graph algorithm allows us to pick only the child classes that are actually initiated. In this case this is $FirstClass$ because A is the only entry point. Hierarchy-based call graph will have to consider both implementations because it does not know the context. SPARK allows us to have more accurate results by ignoring the part of the code that has no effect on our analysis.

We use breadth-first search (BFS) to find the caller-callee pairs. We could not use topological sort due to possible cycles.

The lack of order can lead to missing some flows, so we use a technique called *fuzzy matching* to finalize our analysis. Listing 2 shows an example of such a case. Method A and B could be analyzed in either order. Assuming we analyze method A before B, we will miss the flow of variable a into B because B was never marked as a super-sink. However, in a deeper look we can see that B's parameters were not marked as flowing into the sink because B was not analyzed yet. We can conclude that even if B was a super-sink with no marked variables this flow would still be missed. We solve this problem by re-visiting A at the end of our analysis once we've determined there is a data flow path from B to $sink$ in the analysis of B. Then we can look at the fuzzy flows from A to B and determine that they are now leaks. We repeat this re-visiting step until we reach a fixed point, when we uncover no new paths. Once the bottom up data flow analysis completes, we get the set of super-sinks and their marked dangerous fields and parameters.

```
public class FuzzyMatchExample {
  public void A(int a) {
    B(a); sink(a);
  }
  public void B(int b) {
    sink(b);
  }
}
```

Listing 2: Fuzzy match code example

5.2. Policy Definition and Generation

Now that we have a set of super-sinks, we search for where they are called from within the plugin code, as described in subsubsection 4.1.4. The important difference here is the granularity of the analysis. While in the previous step we have used the entire SDK, here we only scan the available code in the plugin's binary. This step is important so we will apply the AspectJ policy files only to methods that actually exist in the plugin's binary. Once we have all the super-sinks within the binary, we can organize them to generate the default policy implementations as follows.

First, for each method we determine its highest super method and add the super method to its declaring class policy file. We add each child method of the super method as a concrete implementation of the method. For example, the super method of `HttpURLConnection.connect` is `URLConnection.connect`, so the former is a concrete implementation of the latter. Later, we use that relation towards the pseudo dynamic analysis in subsubsection 4.1.3. Each class declaration now has all its methods, and each method has all of its dangerous concrete implementations. Finally, we iterate through all the methods to generate the default policy implementation as a *privileged aspect* in AspectJ for each class. Within each *privileged aspect* we define the AspectJ policy for each method of this class. The default policy denies all calls by throwing *SecurityException* as a fail-safe default, which allows the user to detect unexpected behavior more easily than logging.

5.2.1. Risky Parameters and Fields

Finally, for each super-sink, we computed, we now compute a set of fields and parameters that flow into the sinks declared by policy writer. These variables could potentially be dangerous, so they are very useful to policy writer for refining the policy implementation. We record these variables as comments in the AspectJ files.

6. Evaluation

We tested SURGESCAN on both real world apps and apps that we wrote to demonstrate specific properties.

6.1. Application Scenarios and Example Policies

6.1.1. Secure Distributed Network Measurement

This test shows a use scenario for SURGESCAN: secure distributed network measurement. A scalable, open platform for network measurement from mobile devices is very valuable, e.g., Mobilyzer [19]. It provides a library to be embedded into existing apps, where the library controls measurements and gathers results locally. Although Mobilyzer provides a great service to network researchers, app developers may be wary because of security concerns. SURGESCAN can help overcome this deployment obstacle by allowing the developer to enforce his desirable security policy on the library. For example, the policy writer may specify the following policies: the library may not read from any sensor except the accelrometer, and the library may not connect to a certain IP addresses.

Listing 3 shows a simple implementation of the first policy, where `getDefaultSensor` was the sink that the policy writer specified. First, SURGESCAN automatically generated this file and suggested that the variable `arg0` contains the sensor type. Then, the policy writer inserted the three highlighted lines, which throw a security exception when the sensor type isn't accelerometer.

Listing 4 shows a simple implementation of the second policy, where we used the same sink as before. Again, SURGESCAN automatically generated the file except for the three highlighted lines (added by the policy writer), which throw a security exception when the policy is violated. We omitted less important methods in Listing 4. We note that the approach discussed in this section could be easily extended to other sensor-based distributed measurements, such as distributed barometer measurement.

```
privileged aspect androidHardwareSensorManagerPolicy {
  android.hardware.Sensor around(
  android.hardware.SensorManager obj, int arg0):
       target(obj) && args(arg0)
    && !within( androidHardwareSensorManagerPolicy)
    && call(public android.hardware.Sensor
    android.hardware.SensorManager
    +.getDefaultSensor(int)) {
      /* VAIRABLE arg0 MIGHT BE DANGEROUS */
      if(obj instanceof android.hardware.SensorManager) {
        if(arg0!=
           android.hardware.Sensor.TYPE_ACCELEROMETER)
           {
             throw new SecurityException(
               "Unauthorized");
        }
      }
      return proceed(obj, arg0);
    }
  }
```

Listing 3: A policy blocking all sensors except the accelerometer. Manually modified lines are highlighted

```
privileged aspect javaNetURLPolicy {
  java.net.URL around(java.lang.String arg0): args(arg0)
       && !within(javaNetURLPolicy)
    && call(public java.net.URL.new(java.lang.String)) {
    /* VARIABLE arg0 MIGHT BE DANGEROUS */
    if(arg0.equals("http://169.237.4.76")) {
      throw new SecurityException("Unauthorized");
    }
    return proceed(arg0);
  }
}
```

Listing 4: A policy blocking network calls to illegal domains. Manually modified lines are highlighted

6.1.2. Secure Ad Libraries

This test shows the ability of SURGESCAN to secure private information while using ad libraries. Ad libraries are pervasive in Android apps, but Android's sandbox model allows all the libraries to run with the same privilege as their embedding app. Since many ad libraries abuse this privilege to leak sensitive information [23, 11], the policy writer wishes to grant certain permissions (e.g., to access the GPS sensor) to the app but not to its ad library. SURGESCAN can modify the ad library according to the security policy that the developer specifies. We demonstrate a policy in Listing 5, which limits access to GPS.

```
privileged aspect androidContentContextPolicy{
  java.lang.Object around(android.content.Context obj,
       java.lang.String arg0): target(obj) &&
  args(arg0) && !within(androidContentContextPolicy) &&
      call(public java.lang.Object android.content.
        Context.getSystemService(java.lang.String)) {
    if(arg0.equals(
        android.content.Context.LOCATION_SERVICE)) {
        return new
          CustomizedLimitedLocationService(obj);
    }
    return proceed(arg0);
  }
}
```

Listing 5: A policy blocking GPS access.

We applied the policy to Mobfox [18], a popular library that supports native ads. We wrote a simple application implementing native ads. We verified that Mobfox functioned correctly while enforcing our policy implementation.

6.1.3. Control UI and Screen Estate

This test shows how SURGESCAN can address UI concerns. Screen estate (the amount of screen space used) is highly valuable for Android apps, especially on small devices. Unscrupulous libraries may grab more screen space than desired by the app developer. For example, ad libraries may get more clicks and raise the value of impressions by increasing its window size. Since many libraries do not allow the app developer to customize its windows size, the developer can use SURGESCAN to resize ads and other graphics.

Modifying the UI is challenging, because the UI and plugin run in different threads, but SURGESCAN does not modify the UI code.[1] Instead, our policy hooks on the call in the plugin to set up the UI.

To demonstrate the feasibility, we wrote a plugin that loads an image from the Internet and displays it on the screen in full size. Our policy implementation hooks on the method that sets up the image on the screen (`LayoutParameters`), and then changes the size and position of the image (Listing 6).

1. To increase deployability, SURGESCAN modifies only the plugin code but not either the Android platform or app code.

```
privileged aspect androidViewViewGroupPolicy {
    void around(android.view.ViewGroup obj,
         android.view.View arg0): target(obj) &&
         args(arg0) &&
    !within(androidViewViewGroupPolicy)  && call(public
         void
    android.view.ViewGroup+.addView(android.view.View)) {
       arg0.setLayoutParams(new
         android.view.ViewGroup.LayoutParams(300,
           300));
       proceed(obj, arg0);
  }
}
```

Listing 6: A policy resizing and repositioning images displayed by the plugin. Manually modified lines are highlighted.

6.1.4. Patch Applications

This test shows our ability to update apps using the same technology that SURGESCAN is built on. App markets are the primary channel for distributing Android apps. Good markets endeavor to keep its apps benign and secure by diligently removing malicious ones. However, when a vulnerability is found in a popular library, the market is at the mercy of all the developers to update and upload their apps.

SURGESCAN provides a better alternative. The market owner specifies a policy that declares the method containing the vulnerability. Then, he runs SURGESCAN to generate a default policy implementation, refines the implementation to incorporate his patch, and weaves the patch into the library. The owner can automate this process on all the apps on her market.

Listing 7 demonstrates a simple refined policy implementation that changes the behavior of the URL.new(java.lang.String) method only within the com.seccess.marketer.testad.LoadImpl package. The line highlighted in red restricts the policy to only a specific package in the plugin. Our test confirmed that calls to *URL.new* were modified only when they came from the package com.seccess.marketer.testad.LoadImpl.

Deploying a patched app faces a practical problem, because the new app is not signed by its original developer. We could overcome this problem by introducing a trusted-third party, e.g., the market, who signs the app.

```
privileged aspect javaNetURLPolicy {
  java.net.URL around(java.lang.String arg0): args(arg0)
       && !within(javaNetURLPolicy) &&
  within(com.seccess.marketer.testad.LoadImpl)
     && call(public java.net.URL.new(java.lang.String)) {
    /* VARIABLE arg0 MIGHT BE DANGEROUS */
    if(arg0.equals("http://169.237.4.76")) {
      throw new SecurityException("Unauthorized");
    }
    return proceed(arg0);
  }
}
```

Listing 7: Policy blocking network calls to illegal domains only within a certain package. Manually modified lines are highlighted in yellow and red, where the yellow lines describe the new behavior and red line restricts the policy to a certain package.

6.2. Accuracy

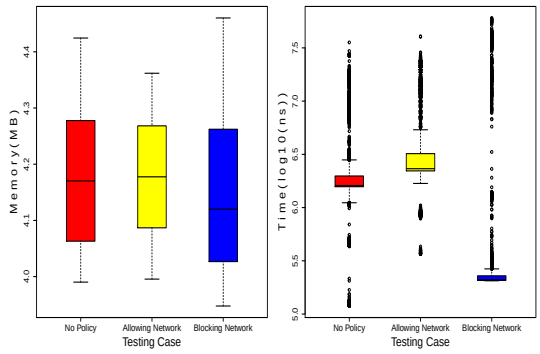
The purpose of this test was to ensure that the policy was applied only to the third-party plugin but not the app code. During this test, we focused on policies that restrict the Internet use for the plugin. We wrote a simple test app that loads a plugin and makes one method call to invoke the plugin. We examined logs to see whether Internet connections were established. We chose the following open-source applications from F-droid [9] as plugins: *URLEvaluator, External IP, CineCat*, and *Flashlight*. Only the first three may reach the Internet, but we also analyzed the last one (*Flashlight*) to evaluate SURGESCAN when the sink isn't expected to be present. We used the following sink for all the plugins: `libcore.io.IoBridge: boolean connect( java.io.FileDescriptor, java.net.InetAddress,int,int)`

Our tests used Android level 21. We extracted *android.jar* from the emulator. We ran *netstat* [16] to see which application listened on a specific port. By checking the active ports, we learned if the test app established network connections.

- *URLEvaluator*: This app evaluates short URLs. SURGESCAN created 8 output policy implementations containing all the network methods. However, it also tainted some more generic methods, all of which came from either `java.lang` or `java.util`. After ignoring the generic methods, we easily modified the URL and HTTP connection methods, and then verified that we successfully blocked Internet connections.
- *CineCat*: This app provides information about Catalan movies. It uses the Internet and SQL database as well as makes calls to resources. SURGESCAN generated policy implementations for all the expected network classes, SQL classes and Android resource getters. We saw generic methods similar to what we found in `URLEvaluator`. We were able to modify the network methods and successfully block network connections.
- *ExternalIP*: This app displays the local and external IP addresses of the network interfaces. It uses `org.apache` Internet methods provided by Android. SURGESCAN found all these methods as we expected. However, we did not expect SURGESCAN to also taint a constructor to java.util.Calender. We found that it was tainted because when it accesses resources on the device, it uses an API method similar to that of network access. SURGESCAN also found some policies around classes that access IO resources. This is because the sink we used is also a method used for IO communication, so it is unsurprising that SURGESCAN found methods dealing with IO.
- *Flashlight*: This app uses the camera flash as a flashlight. This app does not use the Internet, but SURGESCAN generated policy implementations for setting and getting parameters from the camera. This is because most communications inside the android system are done using sockets.

6.2.1. Automated Testing

SURGESCAN can be used to secure not only plugins but also entire apps. This automatic test evaluated the ability of SURGESCAN to find all the potential risky points in real-world apps. We crawled 274 random apps from Baidu market, a top Chinese market [5]. We chose Baidu instead of Google Play because apps on Baidu were reportedly less well regulated and analyzed, so they were more likely to derail automated tools, such as SURGESCAN. Of these apps, 9 either had corrupt APK files or included dex code that *soot*



(a) Memory Use (b) Run time

Figure 2: Memory and run time overhead

could not parse, 6 could not be unpacked and repacked, and 34 could not be installed or run even without modification. We applied SURGESCAN to the remaining 225 apps successfully. We determined the launching and call-back methods using soot-android-infoflow[1] and used them as the entry points. We used the same sink as before. For each app, we first ran the original app and then ran the app with the policy weaved in, while monitoring network activities during each run. We observed that 101 apps established internet connections when we ran the original version. After applying SURGESCAN, we observed no internet connection from these apps. This demonstrates SURGESCAN's high accuracy.

6.3. Performance

We evaluated the impact of SURGESCAN on the performance of apps. We wrote a plugin that downloaded a 20-byte webpage from a local server. The app that loaded the plugin recorded memory usage and other statistics. We tested this plugin in the following three scenarios: Run the plugin without any policy, apply a policy that checks the destination of the connection and then allows the connection through and apply a policy that denies access to our local server by raising a security exception. We ran each scenario several thousands times to take the average measurement.

Figure 2a and 2b show that executing the policy has a negligible performance penalty on the app. As expected the third scenario reduces both the memory and run time.

7. Discussion

7.1. Data Flow Analysis

As discussed in subsubsection 4.1.1, SURGESCAN's current bottom-up data flow analysis can be easily parallelized to increase its speed significantly. Since data flow analysis is critical for SURGESCAN, a more accurate data flow analysis could improve the accuracy of SURGESCAN.

Currently, the developer has to find all the sinks manually, which demands high familiarity with the Android API. Pscout is a tool for mapping Android API methods to permissions that guard those methods via static analysis [2]. When the developer wishes to restrict certain permissions used by the plugin, he could provide the corresponding API methods as sinks to SURGESCAN. Another approach is to help the developers detect sinks. SuSi is making progress in this direction, which attempts to detect sources and sinks automatically using machine learning [20].

7.2. Policies

To write good policies for SURGESCAN, the developer may need to be familiar with the code of the plugin. SURGESCAN currently prints the name of each method for which it has applied a policy, but it could use a more complex trace analysis using AspectJ to provide a great starting point to better understand the plugin's code interaction [24].

Policies regarding the UI-thread are difficult to implement, because UI modifications have to occur on the UI-thread, but SURGESCAN runs the dynamically loaded plugin on its own thread. subsubsection 6.1.3 uses the trick of interposing on UI setup to overcome this difficult, but more general modifications of UI behavior would be more challenging.

8. Related Work

AndroidLeaks [10] found privacy leaks by finding flow of data between sources and sinks by mapping specific permissions to data getters in the Android SDK. However, permission checks could be implemented in native code which would be overlooked by AndroidLeaks. SURGESCAN can detect calls to permission checked by the Android API. SURGESCAN is able to detect all permission checks. Even though some permission checks are done in the native code, SURGESCAN can find the calls that link the Android API into the native code. Therefore given the correct sinks SURGESCAN can easily detect permission checks.

As mentioned in subsection 7.1, Pscout [2] could help simplify writing the policy declaration.

I-arm-droid [7], RetroSkeleton [6], and AppGuard [4] explored applying policies to Android apps. They focused on rewriting Dalvik bytecode in released Android apps. These systems could be used as a replacement for AspectJ in SURGESCAN, requiring we change our policy language.

Aframe [25] and AdSplit [22] isolated a process from the running app. Both used ad library as an example of a process worth isolating. Aframe achieved isolation via modifications to the Android OS while AdSplit did not. SURGESCAN has the same advantage as AdSplit as it does not require OS modification. However, SURGESCAN provide more fine-grained control over access control.

Dr. Android and Mr. Hide [13] and PEDAL [17] are tools also developed to allow for finer permission access control. However, Dr. Android and Mr. Hide stil provided a premade set of permissions and PEDAL removed permission from certain portions of the app. SURGESCAN allows the user to define her own permissions based on the wanted parameters, thus allowing more control.

Boxify [3] is a platform that restricts app resources through sandboxing the app. While Boxify can address similar security concerns by blocking the app calls for resources or system calls, the main difference lies in SURGESCAN's

extra capabilities to also interpose on the app code itself, such as UI calls.

9. Conclusion

Many Android apps use third-party libraries (plugins), but untrusted plugins can compromise app security. We proposed SURGESCAN to instrument the plugins with the developer's security policy, therefore relieving the trust that the developer must place in the plugin.

SURGESCAN is easily deployable because it requires modification to neither the Android platform nor the app code. SURGESCAN allows highly customizable policies,

We developed a bottom-up data flow analysis to find the data sources that are security-sensitive in the plugin. Based on the analysis, SURGESCAN automatically enforces the policies using AspectJ and allow for the developer to impleent fine-grained access control with negligble runtime overhead.

References

[1] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 259–269.

[2] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 217–228.

[3] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. "Boxify: Full-fledged app sandboxing for stock Android". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 691–706.

[4] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. "AppGuard–enforcing user requirements on android apps". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013.

[5] *Baidu App Market*. URL: http://shouji.baidu.com/.

[6] Benjamin Davis and Hao Chen. "RetroSkeleton: retrofitting android apps". In: *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM. 2013, pp. 181–192.

[7] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. "I-arm-droid: A rewriting framework for in-app reference monitors for android applications". In: *Mobile Security Technologies* 2012 (2012).

[8] Arni Einarsson and Janus Dam Nielsen. "A survivors guide to Java program analysis with soot". In: *BRICS, Department of Computer Science, University of Aarhus, Denmark* (2008).

[9] *F-Droid*. URL: https://f-droid.org/.

[10] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale". In: *Trust and Trustworthy Computing*. Springer, 2012, pp. 291–307.

[11] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. "Unsafe exposure analysis of mobile in-app advertisements". In: *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM. 2012.

[12] L. Hendren. "Soot A framework for analyzing and transforming Java and Android Applications". In: *Cetus Users and Compiler Infrastructure Workshop*. Oct. 2011.

[13] Jinseong Jeon et al. "Dr. Android and Mr. Hide: fine-grained permissions in android applications". In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2012, pp. 3–14.

[14] Gregor Kiczales et al. "An overview of AspectJ". In: *ECOOP 2001Object-Oriented Programming*. Springer, 2001, pp. 327–354.

[15] Ondej Lhotk and Laurie Hendren. "Scaling Java points-to analysis using Spark". In: *Compiler Construction*. Springer. 2003, pp. 153–169.

[16] LipiLee. *netstat*. Nov. 2015. URL: https://github.com/LipiLee/netstat.

[17] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. "Efficient privilege de-escalation for ad libraries in mobile apps". In: *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM. 2015, pp. 89–103.

[18] *Mobfox*. URL: http://www.mobfox.com/.

[19] Ashkan Nikravesh, Hongyi Yao, Shichang Xu, David R. Choffnes, and Z. Morley Mao. "Mobilyzer: An Open Platform for Controllable Mobile Network Measurements". In: *13th international conference on Mobile systems, applications, and services (MobiSys),* 2015.

[20] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *NDSS*. 2014.

[21] Franziska Roesner and Tadayoshi Kohno. "Securing Embedded User Interfaces: Android and Beyond." In: *USENIX Security*. 2013, pp. 97–112.

[22] Shashi Shekhar, Michael Dietz, and Dan S Wallach. "Adsplit: Separating smartphone advertising from applications". In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 553–567.

[23] Ryan Stevens, Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "Investigating User Privacy in Android Ad Libraries". In: *Workshop on Mobile Security Technologies*. 2012.

[24] Maximilian Strzer, Jens Krinke, and Silvia Breu. "Trace analysis for aspect application". In: *Workshop on Analysis of Aspect-Oriented Software (AAOS)*. 2003.

[25] Xiao Zhang, Amit Ahlawat, and Wenliang Du. "AFrame: isolating advertisements from mobile applications in Android". In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM. 2013, pp. 9–18.