"**SARAN**: A **S**ystem for **A**nd**r**oid **A**pplication I**n**terposition"

Aarno Labs LLC

Final Technical Report

Oct 20 2021

Sponsored By
Defense Advanced Research Projects Agency (DOD)
(Information Innovation Office (I20))

MIPR HR0011728488

Issued by U.S. Army Contracting Command - Redstone
Under
Contract No. W31P4Q-17-C-0149

Name of Contractor:	Aarno Labs LLC
Principal Investigator:	Michael Gordon
Business Address:	245 Main St, 2nd floor, Cambridge MA 02142
Phone Number:	681-222-7664
Contract Effective Date:	July 17 2018
Short Title:	SARAN
Contract Expiration Date:	October 20 2021
Reporting Period:	July 17 2018 to October 20 2021

TABLE OF CONTENTS

List of Figures

Figure **Page**

List of Tables

Table **Page**

1.0 SUMMARY

Mobile applications are an important computing platform for the Department of Defense (DoD); securing mobile devices, and their applications, is a primary concern. Unfortunately, the level of security controls exposed by mobile platforms do not typically meet DoD requirements. Effectively securing these applications requires the ability to transparently and efficiently identify functional code blocks and add instrumentation that satisfies DoD requirement and policies. Retrofitting security controls and policies to mobile devices and their applications is an extremely challenging problem. First, identifying instrumentation points (functional code blocks) is difficult when the application employs obfuscation. Second, retrofitting instrumentation transparently requires sophisticated techniques that hide the side-effects of injected instrumentation from the application, while maintaining functionality. Third, maintaining transparency and control of instrumentation can adversely impact application performance.

In this project we developed a new system: **S**ystem for **A**nd**r**oid **A**pplication I**n**terposition, Saran. Saran supports the transparent instrumentation of Android applications, enabling analysts to retrofit security policies to third-party applications. Our system is composed of the following components: Interface for Application Programming Interface (API) and Code Block Discovery; Framework for Instrumentation Insertion; Dynamic taint tracking; and an Introspection Transparency module.

The interface for API and Code Block discovery enables analysts to effectively find and instrument application-specific and Android API actions. To help instrumentation, analysts navigate the space of Android APIs. We developed a high-level action description language that summarizes common actions and exposes a common hook for instrumentation insertion.

The Framework for Instrumentation Insertion exposes an API that facilitates the insertion of analyst-written instrumentation at insertion hooks (e.g., pre and post API calls). An instrumentation API, in combination with a high-level action description language, can significantly reduce development time and improve the accuracy of instrumentation. To further enhance the capabilities of the Instrumentation API, we expose common instrumentation analyses, such as intra-procedural dynamic taint tracking, to the analyst. For example, the analyst can determine if location information flows to the network.

Dynamic taint tracking via instrumentation adds additional instructions to the original program to track the sensitive information sources that influence a program value. Calculations that operate on sensitive information will be recognized by Saran by inserting dynamic taint tracking instructions for sources of interest in sections of the application of interest. For example, if the introspection would like to report when location information is written to a network connection, Saran will add dynamic taint tracking instructions for all instructions that could be on the path from a location

read to a network write. This mechanism will provide unprecedented levels of precision for these types of predicated introspection directives. The current prototype supports intra-procedural taint tracking.

The Introspection Transparency module provides runtime instrumentation transparency for Dalvik Executable (DEX) instrumentation such that the instrumentation becomes invisible to the application. It also provides techniques that help evade application-specific integrity checks (e.g., it hide changes any changes to bytecode). In Phase I, we developed techniques that maintain transparency for reflective calls such as method and field introspection.

Finally, Saran was evaluated on both the Android Compatbility Test suite and approxiamtely 100 benchmark applications

2.0 INTRODUCTION

Saran is a novel system to transparently, and efficiently, instrument Android applications. Saran supports the transparent instrumentation of entire Android Android Package (APK)s Dalvik Executable (DEX) bytecode is instrumented using a static binary decompilation that lifts the bytecode into an intermediate representation that facilitates program analysis and transformation. Our DEX bytecode instrumentation maintains transparency by intercepting and sanitizing reflective, and other introspective, calls and provides completeness by supporting reflection.

Saran enables APK instrumentation by defining a set of common events to instrument. Saran will redirect such events to a static *wrapper* routine. The wrapper routine will take any desired actions and then call the original routine to complete the task transparently. All of the arguments and the return value will be available to the wrapper routine for inspection and instrumentation.

Dynamic taint tracking via Saran instrumentation adds additional instructions to the original program to track the sensitive information sources that influence a program value. Calculations that operate on sensitive information will be recognized by Saran by inserting dynamic taint tracking instructions for sources of interest in sections of the application of interest. For example, if the introspection would like to report when location information is written to a network connection, Saran will add dynamic taint tracking instructions for all instructions that could be on the path from a location read to a network write. This mechanism will provide unprecedented levels of precision for these types of predicated introspection directives. The current prototype supports intra-procedural taint tracking.

2.0.1 Related Work

2.0.1.1 ClearScope

Aarno Labs is a subcontractor for MIT's ClearScope project under Defense Advanced Research Projects Authority (DARPA)'s Transparent Computing (TC) program. ClearScope's goal is to augment Android with full-stack, precise, comprehensive and efficient provenance tracking for all program values. To date, Aarno Labs has led the effort to develop the DEX instrumentation component of the system. This component instruments the DEX component of Android APKs to add provenance injection, tracking, and reporting. Each primitive program value (and array) is augmented with a provenance tag rooting the sequence of sensitive sources and sinks that influence the value at the current program point. Indirect flows such as Remote Procedure Call (RPC), files and shared memory are supported with byte-level propagation and functional transparency. Aarno Labs has developed a robust DEX instrumentation framework built on top of the Soot Java Analysis

library [1]. We have also developed a suite of analyses relevant to reducing the overhead of DEX instrumentation.

The system includes a modified Dalvik runtime that is provenance-tag aware. We have modified the underlying runtime implementations of various foundational Java classes and system libraries to propagate provenance tags transparent to the application. These modifications help ClearScope to achieve unprecedented completeness, precision, and efficiency for the provenance instrumentation. However, modifying the underlying Android system is not strictly necessary, and Saran will provide taint tracking without modifications to the underlying Android system.

Furthermore, Aarno Labs has developed sophisticated optimization of the instrumentation to both elide tag propagation and to emit specialized versions of methods with reduced tag propagation code; both without sacrificing accuracy or precision. The combination of these optimizations enables the ClearScope system to achieve a 14% overhead for the CaffeineMark Java benchmark suite with full provenance tracking enabled. These optimizations will be leveraged for Saran to reduce the overhead of instrumentation such that it is not noticeable by the user.

Finally, ClearScope identifies and categorizes all sensitive sources and sinks that cross the application sandbox. These include sensor interactions and resource interactions. Saran's API-based functionality identification is built on this work.

Unlike ClearScope, Saran instruments stand-alone APKs without requiring modifications to the underlying Android operating system. Saran also supports custom user instrumentation.

2.0.1.2 VIBRANCE

Employees of Aarno Labs previously employed at MIT/CSAIL were responsible for the Java runtime instrumentation portion of Kestrel Institute's VIBRANCE project. VIBRANCE was part of IARPA's StoneSoup program the IARPA StoneSoup project (August 2010 to December 2014, Kestrel Institute PI Alessandro Coglio, coglio@kestrel.edu, 650-996-9634).

VIBRANCE uses run-time instrumentation of Java bytecodes to automatically protect Java programs from a variety of security vulnerabilities. These vulnerabilities include injection (SQL, LDAP, XQuery, etc), path traversal, number handling, error handling, concurrency and research drains.

VIBRANCE precisely tracks 32 bits of meta-data information for all primitive types including characters, bytes, integers, floats, array elements, and the effect of system libraries on propagation. The meta- data includes not only the traditional taint information, but also underflow/overflow information for integer values. This allows VIBRANCE to precisely detect dangerous operations on underflowed/overflowed values without false positives. VIBRANCE also detects SQL and command injection attacks and uses its novel taint-aware tokenizer to effectively repair many inputs allowing them to be correctly handled in the normal flow of execution.

VIBRANCE was evaluated by an independent evaluation team. The independent evaluation team identified a set of Java programs ranging in size from 9,000 to 540,000 lines of Java source code) injected errors into the programs with the goal of exposing limitations and/or errors in VIBRANCE,

and used these programs to evaluate the ability of VIBRANCE to identify and nullify security attacks without false positives. VIBRANCE successfully identified and nullified all injection, path traversal, and number handling vulnerabilities without false positives and without any change to program functionality. VIBRANCE also performed quite well on the other vulnerability areas.

VIBRANCE also included AutoRand [2] which automatically randomized SQL keywords to prevent injection attacks. The key technical innovation was *augmented strings*. Augmented strings allow extra information (such as random keys) to be embedded within a string. AutoRand transforms string operations so that the extra information is transparent to the program, but is always propagated with each string operation. AutoRand checks each keyword at SQL statements for the random key. When evaluated on the same programs as above, AutoRand nullified all SQL vulnerabilities without false positives.

Unlike VIBRANCE, Saran instruments stand-alone APKs without requiring changes to the underlying libraries. Saran also supports custom user instrumentation.

2.0.1.3 Java and DEX Instrumentation

Modifying program semantics via instrumentation is a well-known and popular technique for Java, and thus DEX. Java and DEX programs are shipped as bytecode which retain rich type and program information that enables straightforward instrumentation. However, care must be taken for Java idioms that can expose instrumentation (e.g., reflection) or that modify the semantics of bytecode constructs dynamically (e.g., Proxies). Many systems exist for bytecode instrumentation but they typically fall into two categories: instrumentation of application code only (with possible loss of transparency) [1, 3]; and instrumentation of system (library) code and application code (with best-effort transparency promises) [4, 5]. Saran provides provide strong transparency guarantees without modifying system code by adding runtime reasoning for idioms that could break transparency.

2.0.1.4 Dynamic Information Flow Tracking

TaintDroid [6] is an implementation of Dynamic Information Flow Tracking (DIFT) on Android versions up to and including 4.3. TaintDroid is implemented as modifications to the Dalvik VM runtime library that is used to execute DEX bytecode in Android versions up to 4.3. TaintDroid modifies the Dalvik runtime to track 32 bits of taint information. It tracks taint at the level of strings and arrays (conflating the taint on all elements). TaintDroid's mechanisms for capturing DIFT are inappropriate and outdated since Android has moved away from a virtual machine execution model, and they require modifications to the underlying Android system (which is out-of-scope for the requirements of this project). NDroid [7] extends TaintDroid with the ability to analyze native code by interposing software translation, using QEMU, at the JNI layer. Using software translation imposes significant overhead for the analysis of native code and provides no transparency support.

DroidScope [8] and CopperDroid [9] are VMI-based fine-grained dynamic binary instrumentation frameworks that supports the ARM architecture and provides a comprehensive interface for Android malware analysis. DroidScope and CopperDroid use full system emulation and thus in-

cur significant overhead and require system modifications that make the systems only useful for off-line malware analysis.

TaintART [10] modifies the on-device DEX compiler to insert DIFT instrumentation code during DEX to native compilation. TaintART achieves low overhead by packing taint bits into processor registers (by modifying the register allocator of the DEX-to-native compiler). TaintART supports imprecise tracking of indirect RPC/Interprocess Communication (IPC) flows. TaintART's mechanism for adding DIFT to Android apps requires modification of the Android system (i.e., a custom Android image).

Unlike these systems, Saran does not require any modifications to the underlying Android operating system or runtime libraries. Also, none of these systems support custom instrumentation of APKs

2.0.1.5 Android App Code Interposition for Security

Aurasium [11] is an Android application interposition tool that provides additional monitoring and policy enforcement of the Android application sandbox. Aurasium modifies and repackages an application APK to include and load a native library that interposes between the Android framework native code and the system shared c/c++ standard libraries. Aurasium modifies the load-time dynamic linking of the framework native code to the system standard libraries, and detours to their interposition library by modifying the target addresses. With this technique, Aurasium can intercept system calls such as low-level Binder communication and file system operations. Aurasium can then enforce more fine-grained security policies such as restricting network traffic based on source / destination and restricting command execution. Aurasium is able to maintain transparency for Java idioms, but it is not clear how they would intercept application integrity checks (such as SafetyNet) to deceive these checks. Furthermore, Aurasium does not support fine-grained information flow and sensitive calculation policies and interposition triggers. Finally, malicious native code could bypass Aurasium (e.g., by targeting the system calls directly, and bypassing the standard c/c++ libraries).

Boxify [12] provides application sandboxing on stock Android. Boxify runs applications as subprocesses using Android's *isolated process* feature. This allows it to capture binder and system call events from outside of the monitored process. The code of the targeted application is modified so that interactions between the application and the system are intercepted by Boxify.

Boxify allows users to explicitly use it to run other applications in the sandbox. Unlike Saran, it does not support delivering applications with included instrumentation.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

For all the research we performed in this program, we adopted an experimental approach driven by test cases. Our overarching goal was to produce systems that could successfully instrument real-world applications.

We focused on applications used as test cases for the Defense Advanced Research Projects Authority (DARPA) Transparent Computing program. These applications interact with the underlying system in ways that are intentionally opaque to analysis.

We evaluated our technology over different scenarios. We observed any deficiencies and developed techniques that addressed those deficiencies.

The underlying assumption behind this research is that these techniques will generalize to larger classes of programs. We see no way to test these assumptions other than testing our techniques out on a variety of different programs.

During the course of the research, we devoted a major effort to the packaging, testing, and usability of the system. Our releases included both unit and system tests (comprising our regression test). Regression tests were a key component to our development approach; allowing us to easily experiment with different approaches and confirm that the system was still meeting its goals.

All of the systems that we developed run on open source infrastructure (e.g., Android Open Source Project) and do not require proprietary software to build and run.

3.1 System

To evaluate our approach, we built Saran, a system to transparently, and efficiently, instrument Android applications. Saran supports the instrumentation of the Dalvik Executable (DEX) bytecode. DEX bytecode is instrumented using a static binary decompilation that lifts the bytecode into an intermediate representation that facilitates program analysis and transformation. Our implemented DEX bytecode instrumentation maintains transparency by intercepting and sanitizing reflective calls.

3.2 Interface for API and Code Block Discovery

We developed an interface that enables analysts to effectively find and instrument application-specific and Android Application Programming Interface (API) actions. For sensitive Android

API actions we used the thousands of sources and sinks already instrumented as part of the APAC and TC DARPA programs. The Android API typically exposes several interfaces for a single functionality. For example, there are tens of API interfaces for extracting location information. To help instrumentation analysts navigate the space of Android APIs, we developed a high-level action description language that summarizes common actions and exposes a common hook for instrumentation insertion. Using this description language, an analyst can select to add instrumentation to all location APIs succinctly. Beyond reducing the amount of actions required to add instrumentation, the use of a description language will also help avoid corner-cases or miss uncommonly used APIs.

Our system supports the following triggers: read/write files; read/write (to/from) network interfaces; read location; send/receive Intent data; receive/send SMS/MMS messages; read contact information, read/writes of the clipboard, voice calls and reads from the camera, microphone, accelerometer.

3.3 Framework for Instrumentation Insertion

We developed an API that facilitates the insertion of analyst-written instrumentation at insertion hooks (e.g., pre and post API calls). An instrumentation API, in combination with a high-level action description language, can significantly reduce development time and improve accuracy of instrumentation. To further enhance the capabilities of the Instrumentation API, we exposed intra-procedural dynamic taint tracking, to the instrumentation analyst. This, in turn, can significantly enhance the effectiveness of the analyst by helping identify sensitive flows to policy enforcement points. As an additional benefit, the exposed analyses, use of our internal data-flow analyses that enable low overhead. For example, using our system, an analyst can add detailed taint tracking of a given list of source and sinks, with little to no overhead.

There are three types of action routines corresponding to three types of triggers. Read triggers occur when the application makes a library call that performs a read (e.g, read a file, read a stream, etc). Write triggers occur when the application makes a library call that performs a write (e.g, read a file, send an SMS/MMS, etc). Callback triggers occur when the application receives data via a callback (e.g., read location, SMS/MMS, or contacts). Some data types may have multiple trigger types. For example, there is a method to directly read the location and it can be obtained via a callback.

Read and write triggers will wrap the original call with the analyst-defined action routine (if present). The unmodified arguments, along with any analysis results (e.g., information-flow analysis), will be passed to the action routine. The action routine can modify these arguments and make the original call or simply skip the call.

Analyst-defined callback actions interpose the existing application callbacks. The action routine can modify callback arguments and invoke the registered callback or simply skip the call (in which case the application will not see the event at all).

Action routines are specified by extending the class `SaranAPI`. `SaranAPI` defines action routines for each of the possible triggers. The arguments for each action routine vary based on the signatures

of the original library method or callback. See Section 3.5 for an example use of the SaranAPI.

3.4 Introspection Transparency

We extended Saran to be more fully transparent to the application in the following ways:

- **Reflection**: Inherited from Java, the Dalvik reflection API enables an application to introspect on classes and methods, and to perform actions on data representing methods and classes (as opposed to action on the methods and classes themselves). For example, an application could ask for data representing the fields in a class. If an instrumentation framework has altered the fields of the class, and the altered (added) fields are represented in the data, then the instrumentation framework is not transparent. Furthermore, since methods can be called indirectly via the reflection framework, in order to be complete, an instrumentation framework must reason about reflection. Saran interposes at reflection API calls such that method and class metadata are sanitized of all instrumentation artifacts. This achieves transparency for reflection. At invocations of functionality via reflection, Saran decodes the reflected metadata to reason about the runtime functionality to be executed by the reflection action.
- **Integrity Checks**: Applications can check themselves in a variety of ways to detect if they have been modified. We intercept these calls so that the results are the same as in an unmodified application. This includes the application's signing certificate, the installer ID, files in the Android Package (APK) (such as the manifest), etc.
- **Reporting**: A common use case for the system is logging sensitive actions. A log often needs to be sent off of the device. This can only be done directly if the application has network permissions. To enable reporting on applications that do not have network permissions, we provide a separate reporting application that can be communicated with. Communicating to the reporting application does not require a permission. The reporting application can be downloaded directly by a cooperative user, or it can be added to any other instrumented application that has network permissions.

3.5 Example

Next, we present an example of how the Saran Instrumentation API can be used to retrofit applications with the following policy: Sanitize any internal write streams that use location information (e.g., GPS data); and drop any writes to the network that contain location information.

```
1  Object writeStream (WriteInfo we, Object[] args, int[] taint) {
2    // zero out any arguments with location information
3    Object[] newargs = new Object[args.length];
4    for (int i = 0; i < args.length; i++) {
5      if ((taint[i] & LOCATION) != 0)
6        newargs[i] = new Double(0.0);
7      else
8        newargs[i] = args[i];
```

```
9       }
10      Object result = null;
11      if (!wi.dest.isNetwork())
12        result = wi.write (newargs);
13      return result;
14    }
```

Listing 3.1: writeStream trigger

In this example, the analyst selects the `readlocation` action and associates it with the `writestream` trigger. Listing 3.1 presents the code that represents the action for the policy. Line 1 shows the declaration of the `writeStream` action interface. The `WriteInfo` class contains information about the original call and its stream. `WriteInfo` additionally contains information about the stream (e.g., the filename or Internet Protocol (IP) address associated with the stream). `Object[]args` represents the argument to the stream as an array of Objects. `int[]taint` represents an array of 32-bit taint tags that is associated with each argument. Lines 3-9 show the implementation of the policy: sanitize any arguments passed to a write stream that contain location information. Each argument checks the taint array to determine if it is tagged with location information. If location information is associated with the argument, the argument is zero'ed (new 0.0 value). Lines 11-12 illustrates the later part of the policy implementation: only pass the sanitized data to a write stream destination that is not tagged as a Network destination.

The full manual for Saran is in Appendix A.

3.6 Instrumentation Approach by Trigger Type

Many of the triggers attempt to treat multiple application events in a consistent fashion to ease the implementation of action methods. For example, there are many different classes and methods that write values to a stream. These are supported by a single write stream action method.

Each action routine is passed an information object generated by Saran that provides additional information about the operation, the ability to skip callbacks (for callbacks) and a method to make the original call (for non-callbacks).

The default implementation for each action routine executes the original call without modification.

Each of the action routines and their corresponding information classes are defined in SaranAPI.

Details on each trigger type follow.

- Stream Writes (files and network)

  There are many different classes and methods that support stream writes. All of these will be directed to a single action routine which will accept an array of objects and their taint. The action routine will be called instead of the original write method giving the action routine the ability to modify the arguments as desired and choose whether or not to skip the write altogether. The write information object provides a method to make the original call and also provides information about the stream itself (stream type (file or socket), filename, IP address, etc)

The signature and default implementation for write is

```
public Object write (WriteInfo wi, Object[] args, int[] taint) {
  return wi.write (args);
}
```

Saran instrumentation replaces each write call with a call to a corresponding static function. The static funtion will take the arguments to write and a corresponding taint variable. Each write call also has a corresponding information class that extends the general WriteInfo class. That information class implements the write() entry point correctly for that function call. The static method and the information class can be automatically generated from the target signature.

For example, If the initial code is:

```
  PrintStream ps
  float f;
  ps.print (f);
```

the instrumented code would be:

```
SaranCalls.printstream_print (ps, new Float(f), f_taint);
```

The implementation of `printstream_print` would look like the following where the users implementation of SaranAPI is named `user_api`.

```
void printstream_print (PrintStream ps, Object val, int f_taint) {

  WriteInfo wi = new WriteInfo_printstream_print_float(ps);
  user_api.write (wi, new Object[] {val}, new int[] {f_taint});
  return;
}
```

The PrintStream.print() version of WriteInfo is:

```
class WriteInfo_printstream_print extends WriteInfo {

  PrintStream ps;

  public WriteInfo_printstream_print_float (PrintStream ps) {
    super (ps);
    this.ps = ps;
  }

  public Object write (Object[] vals); {
    ps.print (vals[0]);
    return null;
  }
}
```

- Stream Reads

As with writes, there are many different classes and methods that read from streams. All of these are directed to a single action routine that can make the original call (if desired) and optionally modify the return value.

The read information object provides an entry point that will make the original call and return the result as an object. There are read calls for integers, characters, strings, and character/byte arrays. The read call will return an object that can be examined to determine its type. The action routine is also passed an argument specifying the type of the read so that the action routine can make choices based on the type before making the read call. The read information object also provides information about the stream itself (stream type (file or socket), filename, IP address, etc)

Note that the actual read calls may modify a parameter rather than return a value, but the `ReadInfo.read()` method always returns the value.

The signature and default implementation for read is:

```
1   public Object read (ReadInfo ri, ReadType rt) {
2     return ri.read();
3   }
```

Saran instrumentation replaces each read call with a call to a corresponding static function. The static funtion's only argument is the read information object (`ReadInfo`). `ReadInfo` implements the `read()` entry point correctly for the function call. The static method and the information class can be automatically generated from the target signature. For example, if the inital code is:

```
1   BufferedReader br;
2   String line;
3   line = br.readLine();
```

The instrumented code would be:

```
1   line = SaranCalls.bufferedreader_readLine (br);
```

The implementation of `bufferedreader_readline()` would look like:

```
1   String bufferedreader_readline (BufferedReader br) {
2     ReadInfo ri = new ReadInfo_bufferedreader_readLine (br);
3     return (String) user_api.read (ri, ReadType.STRING);
4   }
```

The BufferedReader.readline() version of ReadInfo is:

```
1   class ReadInfo_bufferedreader_readline extends ReadInfo {
2
3     BufferedReader br;
4
5     public ReadInfo_bufferedreader_readline (BufferedReader br) {
6       super (br);
7       this.br = br;
8     }
```

```
9
10          public Object read () {
11            return br.readLine();
12          }
13        }
```

If the initial code takes arguments such as:

```
1         InputStream is;
2         Byte barr[100];
3         int cnt = is.read (barr, offset, len);
```

The instrumented code would be:

```
1         int cnt = SaranCalls.inputstream_read (is, barr, offset, len);
```

The implementation of `inputstream_read()` would look like:

```
1         /** copy the bytes read into barr at the correct position and
2          * return the number of bytes read.
3          **/
4         int inputstream_read (InputStream is, byte[] barr, int offset, int len) {
5           ReadInfo ri = new ReadInfo_inputstream_read2 (is, barr, offset, len);
6           byte[] user_arr = user_api.read (ri, ReadType.BYTE_ARRAY);
7           if (user_arr == null)
8             return -1;
9           System.arraycopy (user_arr, 0, barr, offset, user_arr.length);
10          return user_arr.length;
11        }
```

The InputStream.read(byte[], int,int) version of ReadInfo is:

```
1         class ReadInfo_inputstream_read2 extends ReadInfo {
2
3           InputStream is;
4           int offset;
5           int len;
6           int cnt = 0;
7
8           public ReadInfo_inputstream_read2 (InputStream is, int offset, int len){
9             super (is);
10            this.is = is; this.offset = offset this.len = len;
11          }
12
13          public Object read() {
14            byte[] user_arr = new byte[len];
15            int cnt = is.read (user_arr);
16            if (cnt == -1)
17              return null;
18            else if (cnt == len)
19              return user_arr;
20            else { // read fewer byte than requested
21              byte new_arr = new byte[cnt];
```

```
            System.arraycopy (user_arr, 0, new_arr, 0, cnt);
          }
          return user_arr;
        }
      }
```

- Location callbacks

  Location can be found via direct calls (`LocationManager. getLastKnownLocation()`) and callbacks. Callbacks are more common. Earlier versions of Android used the `LocationManager` which could call a users `LocationListener.onLocationChanged()` method. There is now a fused location provider that rather than returning a single location returns a list of locations in `LocationResult`.

  These can also be treated as pending intents, but that can be handled with `receiveIntent()`.

  Our goal is to provide a single location callback that can handle all three cases. Callbacks can be skipped (by returning out of the callback), but that is not possible for direct calls. The actual call to obtain the location could be skipped for direct calls, but not easily for callbacks.

  We chose a callback oriented approach because that is the most common mechanism for obtaining location information. Callback code can be skipped, but that option is not supported for direct calls. When `LocationResult` is provided (by the fused location manager), the Saran handler loops through each location and passes each one to the action routine in turn. If a single location is skipped, then it is removed from the SaranLocationResult passed to the application callback. If all locations are skipped, the callback is skipped (by returning in the prologoue).

  The location action method will be called at the beginning of each callback. The `LocationInfo` will be filled in to indicate that it is a callback. If it is a `LocationResult` callback, then the code will loop through each location calling the user action routine for each location. Locations that are skipped will be removed from `LocationResult` and if `LocationResult` ends up empty, the entire callback will be skipped. Since this is complex, there is a static call which will handle each case and return a boolean indicating whether or not to skip.

  If it is a direct call, the skip() setting is ignored and the only visible action that can be taken is to modify the Location itself. Direct location calls are implemented in the same way as the direct read calls.

  The signature and default implementation for readLocation is:

```
    public void readLocation (LocationInfo li, Location location) {
    }
```

  Callbacks are implemented by placing a static `SaranCall` call at the beginning of each location callback. There are two methods: one for single locations (`SaranCall.locationcb(Location)`) and one for a list of locations (`SaranCall.locationcb(LocationResult)`).

  Direct calls are handled by `readloc(Location)` with the result of the location call.

  For example, if the application callback code looked like:

```
class MyLocationListener implements LocationListener {

  public void onLocationChanged (Location location) {
    System.out.println (location);
  }
}
```

The instrumented code would be:

```
class MyLocationListener implements LocationListener {

  public void onLocationChanged (Location location) {
    LocationInfo li = SaranCall.locationcb (location);
    location = li.location;
    if (li.skip)
      return;
    System.out.println (location);
  }
}
```

The `locationcb` method for location would be:

```
public LocationInfo locationcb (Location location) {

  LocationInfo li = new LocationInfo (location, true);
  user_api.readLocation (li, location);
  return li;
}
```

- Read Intent

  Intents are received by an application in the `onReceive()` method of `BroadcastReceiver`. This callback is handled in a very similar fashion to the location callbacks. A call to `SaranCall.read_intent_cb()` is inserted in the prologue of the onReceive method.

  The signature and default implementation for receiveIntent() are:

```
public void receiveIntent (ReceiveInfo info, Context context,
                           Intent intent) {
}
```

  The user can modify the intent (or set a new one by calling `ReceiveInfo.set_intent()`). It can also skip the callback by calling `ReceiveInfo.skip_callback()`.

  For example, if the initial code is:

```
class MyBroadcastReceiver extends BroadcastReceiver {

  public onReceive (Context context, Intent intent) {
    System.out.println (intent);
  }
}
```

The instrumented code would be:

```
class MyBroadcastReceiver extends BroadcastReceiver {

  public void onReceive (Context context, Intent intent) {
    ReceiveInfo ri = SaranCall.recieve_cb (context, intent);
    intent = ri.intent;
    if (ri.skip)
      return;
    System.out.println (intent);
  }
}
```

The `recieve_cb` method would be:

```
public ReceiveInfo receive_cb (Context context, Intent intent) {
  ReceiveInfo ri = new Receive_info (intent);
  user_api.receiveIntent (ri, context, intent);
  return ri;
}
```

- Send Intent

  Sending an intent is a write trigger. Intents are sent with `Activity.startActivity` or `startActivityForResult`. There are several versions, but they can all be combined together into `startActivityForResult(intent,requestCode,options)`. The requestCode will be -1 if it is just `startActivity()`.

  There are some other variants (startActivityIfNeeded, startIntentSender, etc), that will be addressed in phase 2.

  Saran will insert the action routine into the application in the place of the original call (similar to stream writes). The user action routine can modify the parameters and/or skip the call.

  The signature and default implementation for `writeIntent()` are:

```
void writeIntent (WriteIntentInfo info, Intent intent, int requestCode,
                  Bundle options) {
  info.writeIntent (intent, requestCode, options);
}
```

  The instrumentation replaces the original call with a call to Saran's instrumentation routine. For example, if the original code were:

```
Intent intent = ...
startActivty (intent);
```

  The instrumented code would be:

```
Intent intent = ...;
SaranCall.writeIntent (intent, -1, null);
```

  The `SaranCall.writeIntent()` method would be:

```
public void writeIntent (Activity activity, Intent intent,
                         int requestCode, Bundle options) {
  WriteIntentInfo wii = new WriteIntentInfo (activity);
  user_api.writeIntent (wii, intent, requestCode, options);
}
```

- reading SMS/MMS

  Receiving an SMS or MMS is a callback trigger. These are received by `BroadcastReceiver.onReceive()` (see above). To determine that the received intent is an SMS, the options Bundle can be queried for the key "pdus".

- Sending SMS/MMS

  Sending an SMS or MMS is a write trigger. There are two basic mechanisms for sending messages. One is using `SmsManager.sendTextMessage()` and the other is sending an intent. Intents are already handled generically above, so Saran only supports `SmsManager` here.

  The calls for SMS and MMS are quite different, so there are separate write triggers for each. The signature and default implementation for each are:

```
void writeSMS (WriteSMSInfo info, String destAddress,
               String scAddress, String text, PendingIntent sentIntent,
               PendingIntent deliveryIntent) {
  info.writeSMS (destAddress, scAddress, text, sentIntent,deliveryIntent);
}

void writeMMS (WriteMMSInfo info, Uri contentUri, String locationUrl,
               Bundle configOverrides, PendingIntent sentIntent) {
  info.writeMMS (contentUri, locationUrl, configOverrides, sentIntent);
}
```

  The instrumentation replaces the original call with a call to Saran's instrumentation routine. For example, if the original code were

```
String destAddress = "781−555−1212";
SmsManager manager = ...
manager.sendTextMessage (destAddress, null, "hello␣there", null, null);
```

  The instrumented code would be:

```
String destAddress = "781−555−1212";
SmsManager manager = ...
SaranCall.writeSMS (destAddress, null, "hello␣there", null, null);
```

  The `SaranCall.writeSMS` method would be:

```
public void writeSMS (SmsManager manager, String destAddress,
                      String scAddress, String text,
                      PendingIntent sentIntent,
                      PendingIntent deliveryIntent) {
```

```
    WriteSMSInfo info = new WriteSMSInfo (manager);
    user_api.writeSMS (info, destAddress, scAddress, text,
                       sentIntent, deliveryIntent);
  }
```

- Reading contacts

  Contacts are normally read by making a query on a content provider. Saran instruments SaranContentResolver.query() so that any query can be instrumented. Queries are read triggers.

  Queries return a cursor with results. An action routine can modify and/or remove results by creating either a `MatrixCursor`, a `CursorWrapper` or a custom cursor.

  There are three `ContentResolver.query()` calls. Saran has one action routine that takes all of the possible arguments where missing arguments are null.

  The signature and default implementation for the query call is:

```
  Cursor readQuery (ReadQueryInfo ci, URI uri, String[] projection,
                    Bundle queryargs, CancellationSignal csig) {
    return ci.readQuery (uri, projection, queryargs, csig);
  }
```

  The instrumentation replaces the original call wtih a call to Saran's instrumentation routine. For example, if the original code were:

```
  ContentResolver cr = ...
  Uri uri = ...
  Cursor cursor = cr.query (uri, null, null, null)
```

  The instrumented code would be:

```
  ContentResolver cr = ...
  Uri uri = ...
  Cursor cursor = SaranCall.readQuery (cr, uri, null, null, null);
```

  The `SaranCall.readQuery()` method would be:

```
  Cursor readQuery (ContentResolver cr, URI uri, String[] projection,
                    Bundle queryargs, CancellationSignal csig) {
    ReadQueryInfo info = new ReadQueryInfo (cr);
    user_api.readQuery (info, uri, projection, queryargs, csig);
  }
```

3.7 Taint Tracking

The system supports up to 32 different types of taint per value. The types of taint that are of interest can be defined by the user. For example, the user can look at file pathnames and/or IP

addresses to determine how to mark values read from that stream. For example, /etc/passwd, /usr/include/ctype.h, and an application specific file might have different taint values.

Taint can be set in the appropriate action routine for each trigger type. The SaranAPI has calls to set the taint on an object. For example, SaranAPI.setTaint (obj, int)

String taint is tracked for the string as a whole (the union of the taints of every character). This is kept in a global weak identity hashmap indexed by the string. The intern operation will also merge the taint of the two strings.

For string operations, we implemented method summaries to transfer taint. Saran replaces string method calls with a static call that takes the string as an argument. The static call will merge the taints of the arguments as appropriate and then call the original method to perform the string operation. For example:

```
1  this_str.concat (arg_str) ==>
2
3  str_concat (this_str, arg_str) {
4    String result = this_str.concat (arg_str);
5    int taint = taint_map.get(this_str) | taint_map.get(arg\_str);
6    taint_map.put (result, taint);
7    return result;
8  }
```

Note that since the new call has the same stack signature as the original call, adding this instrumentation is easy even for a bytecode editor such as ASM. Obviously, also easy to do in Soot.

In phase 1 we implemented a straightforward intra-procedureal taint tracking mechanism. The Phase 2 system supports inter-procedural results, by passing taint between methods by using Thread Local Storage (TLS) variables rather than changing method signatures. Taint values are copied into local variables in the method prologue and the taint of the return value is copied to a TLS variable upon return. This approach uses the current stack but leaves the method signature unchanged. This supports exceptions and cleanly supports interactions between instrumented (application) and uninstrumented (system library) calls.

Using TLS to pass information between methods rather than adding arguments simplifies things by leaving signatures unchanged. That means we can instrument the original method rather than making a copy. The goal is to be able to inter-operate between instrumented and uinstrumented code. This is crucial because we are not instrumenting system libraries.

One difficulty is that one does not necessarily know at a call site whether or not the callee is instrumented or not. An interface or virtual call could be to an instrumented application method or to an uninstrumented system library method. Even within an application, the callee could be an instrumented method or an uninstrumented native call.

A safe approach to this is to add information (callee-id) that identifies the callee when setting argument taint information and return taint information. The callee-id can be checked in the callee to make sure that the taint information is intended for the callee. It can also be checked when obtaining taint for return values to ensure that the returned taint is from the callee. If there is a mismatch taint is simply set to unknown.

The callee-id is created by hashing the name of the method and the type of each of its primitive arguments. This can be done without any global information and with a 32 bit id is very unlikely to incorrectly match (a 64 bit ID could make this vanishingly unlikely, but seems like overkill).

The TLS information has the following information:

- int argument callee-id
- int[] array of argument taints
- int return value callee-id
- int return value taint

The argument callee-id is set at each call (along with the argument taints) and the return value callee-id is cleared. At each return, the argument callee-id is cleared and the return value callee-id is set. This ensures that there are not inadvertent matches.

Our current version checks to determine if a method is instrumented by examining its classloader. Instrumented code is presumed to have the java system classloader.

3.7.1 User Defined Taint

Users can add action routines that can execute arbitrary code when information is read. This code can set the taint of the corresponding data in an arbitrary fashion. The system provides 32 bits of taint for each program value and automatically propagates that taint. If taint from two values is merged (as when an add operation is performed on two values), the result has the taint of the union of the taint of the two operands.

One common use case would be to set different taint bits based on the the pathname or IP address of the source of the data. We will add calls to the ReadInfo object that return the pathname for files and the IP address for sockets. The user can determine what information should be tracked. For example, the contents of /etc/passwd and other system files might be marked as `system`. The contents of application specific files from specific application directories can be marked as `application`. If some application data should never be sent off of the device, it can be marked as `application-private`. User specified taint can be checked at any output methods (or other methods of interest) and appropriate action taken based on the taint.

We also support calls to set the tag of a value or to join a tag with any previous tag of a value.

3.8 Spatial/Temporal Locality

Saran allows a user to take action at events that are proximate to other events in space or time. The basic approach for proximity triggers is for the user to mark an event of interest and then query that mark at other events to see if the specified proximity exists. Marks are specified in the class `EventMarks`. The user can create up to 32 different EventMark types.

The purpose of proximity triggers is to look for relationships that may not directly transfer taint. This could occur because the application is purposefully hiding its actions (e.g., by implicitly

transferring taint or using side channels) or because the values are not explicitly transferred.

When the user creates a mark, the mark records the following information

- The current time (milliseconds)
- The current process cpu time (milliseconds)
- The current instruction count. This is approximated by a count of methods executed.
- The current method executing in the application including its class and package.
- The current method frame. The purpose of this is to be able to determine if the mark method is still active at a later point. The implementation of this uses the tag TLS infromation. Each mark specified in the method will be added to an active list. When the method exits, the mark will be removed from the active list. Since we already trap all method exits, this adds minimal additional overhead.

  Note that we can't just use the method name and the Java stack to implement this. The named method that called the mark may be on the stack, but it may not be the particular invocation of the method that called the mark.

The new mark is added to the front of a list of marks for the specified type (this can just be an array of lists indexed by the `EventMark` type). If there is a duplicate mark in the list (same method), it is removed. There is a global setting indicating how deep a list should be maintained. This can also be specified by time (eg, entries older than X can be removed). In practice, there are not that many marks that happen in many different methods, so the lists might not get very big.

Later at any other event (such as an event writing to the network), the user can query by any of the proximity types (time, instruction count, within, locality). There is a call for each proximity type that returns a list of matching EventMarks.

```
1  static List<EventMark> EventMark.recent (int type, int time_ms)
2  static List<EventMark> EventMark.recentCPU (int type, int time_ms)
3  static List<EventMark> EventMark.locality (int type, String name)
4  static List<EventMark> EventMark.callDistance (int type, int method_cnt)
5  static List<EventMark> EventMark.within (int type, int depth)
```

Each returns a list of events for the specified type that have occurred within the specified proximity. For 'recent' and 'recentCPU' this is specified in milliseconds. For 'callDistance' this is specified in the numbers of method calls executed. Locality is specified by a string that denotes a package, class, or method name. If the mark occurred in the specified component, it will match. The 'within' method specifies the depth on the stack between the current method and the marked method.

There is also a call that can search for events that match combinations of proximity types

```
1  static List<EventMark> EventMark.prox (int type, int time_ms, int cpu_time_ms,
2                                         int method_cnt, int depth, String name)
```

Each parameter has a setting that matches all events:

- **time_ms**: -1

- **cpu-time_ms**: -1
- **method_cnt**: -1
- **depth**: -1
- **name**: null

Thus to find all of the events that occurred in the last 50 millseconds that are within a depth of 5 on the call stack, execute:

```
EventMark.prox (type, 50, −1, −1, 5, null)
```

The EventMark class looks like:

```
class EventMark {
  int time; // current time in msecs
  int cpu_time; // current cpu utilization time in msecs
  int inst_cnt; // current instruction or method count
  String method; // fully qualified name of current method
  int type; // user specified type.
  Object info; // Event specific information
  String descr; // Optional descriptor of information
}
```

3.9 Method Proxies

Proxies are wrappers that pass function invocation through to a different class, adding/modifying functionality as desired. Proxies can be created for arbitrary classes and can delegate back to the original call as needed. They are thus easily used for logging or timing or other operations that still execute the original call. They can, however, choose a different implementation of the call some or all of the time.

To support proxies our instrumentation must continue to catch all calls to methods of interest and correctly instrument them.

Similarly to how reflection is supported, the taint for each primitive argument passed to proxy is first stored in a weak identity hash map for each argument (because arguments are boxed for proxies) and then extracted when the actual call is made. A similar process was implemented for return values.

3.10 Dynamic Loading

We added support for dynamic loading of DEX code. This allows the program to correctly operate but does not instrument the dynamically loaded code. Events within the dynamically loaded code will not be recognized.

Supporting events in dynamically loaded code is not feasible because it would require our entire tool (including Soot) to be included with the instrumented application.

3.11 Optimizations

Taint tracking can be expensive (see Section 4.4 for details). We have developed a number of optimizations that significantly (more than 50X) reduce overhead.

We addressed a variety of optimizations as uncovered by the CaffeineMark Java benchmark. The major problems result from array manipulations which require us to obtain and update taint from a global identity hash map (as we cannot add fields to arrays to store taint). One possible optimization angle is to take advantage of the fact that the taint for all of the elements of an array are conflated.

- **Identify Optimization Opportunities**. We analyzed the code looking for opportunities to hoist taint operations out of loops. We also identified situations under which we could create specialized versions of functions based on the input taint of the function arguments. We also found situations where the return value (and other side effects) of the fucntion are not affected by the argument taint. For example, a function that traverses lists/arrays and returns count information (such as the total count of the list or the number of items that meets a particular condition) does not need to process/propagate taint information.
- **Loop Optimizations**. Implemented a preliminary version of optimizations for loops over arrays. Preliminary testing shows good results in some situations.
- **Improved Loop Optimizations and Data Flow Analysis**. Our data flow analysis strives to identify which taint operations can be hoisted out of loops or other wise eliminated. We were able to add precision to our data flow analysis allowing us to hoist more operations out of the loops. In the CaffeineMark Java Benchmark, we are able to achieve good results (less than 50% overhead on several of the benchmarks).

  We also fixed some bugs in the trivial flow propagator to correctly merge the trivial flows up the hierarchy till a fixed point is reached, and added optimizations to TagPropAnalysis for eliding field read/write expressions that are not influenced by any other outside tag.
- **Identify Loop Invariant Tag Operations**. In some cases, the tag operations are identical in each iteration of the loop. This is common because the tag for each element of the array is the same in each iteration (because arrays conflate the taint across their elements). This allows the opportunity to calculate the taint once (on the first iteration) and remove the taint operation from subsequent iterations. We have experimented with this approach and it seems promising. We will continue to investigate this.
- **Identify and Optimize Trivial Flows**. Added optimizations for determining whether a method has a trivial flow (i.e. the tag is only influenced by the tags on the parameters, and contains no other side-effects with respect to tags). We then use this analysis to skip the instrumentation of these calls, and essentially assign the tag for the return after the call with the result of the trivial flow.
- **Raised Loop Invariant Tag Operations**. In some cases, the tag operations are identical in each iteration of the loop. This is common because the tag for each element of the array is the same in each iteration (because arrays conflate the taint across their elements). This

allows the opportunity to calculate the taint once (on the first iteration) and remove the taint operation from subsequent iterations. We have implemented this approach and have seen some substantial speedups (3X to 4X over unoptimized code). These loops however are still slower than we would like and we will need to investigate this problem further.

- **Resolved Verification Errors**. The loop optimizations introduced some new verification errors (due to interactions of our optimizations with existing but not yet uncovered Soot problems). We were able to resolve these errors.
- **Improved Trivial Flow Propagator**. Fixed bugs in the trivial flow propagator to correctly merge the trivial flows up the hierarchy till a fixed point is reached.
- **Improved Tag Propogation Analysis**. Added optimizations to TagPropAnalysis for eliding field read/write expressions that are not influenced by any other outside tag.
- **Improved Tag Map**. Replaced the tag map with a (potentially more performant) map found in guava that specifically handles weak references over the identity and is also concurrent. It seems to at least be on par with the previous implementation, and significantly more performant when the map is sufficiently big (> 1 million entries or so).
- **StringBuffer vs StringBuilder** The string benchmark uses `StringBuffer` to build strings. This class is threadsafe. The taint tracking code needs to add the `StringBuffer` into the tag map and then look it up whenever it is modified. The tag map is an `IdentityHashMap` and is indexed using `System.identityHashCode()`. Because of the synchronization required by a thread safe class, the first call to obtain the identity hashcode is much slower than subsequent calls and our optimizations (which remove subsequent calls) are not as effective as expected. However, in this, and many other cases, the use of a `StringBuffer` is not needed (the object is created within the function and never escapes it). Our instrumentation automatically changes such `StringBuffer` objects to `StringBuilder` objects which has the same interface but it is not thread safe. The resulting code is much more efficient in all cases. Our overhead is reasonable (even when compared to a version of the benchmark that uses `StringBuilder` instead of `StringBuffer`).

3.12 Implementation Details

Implementing instrumentation that is transparent to the application is challenging. Below are some of the issues that we resolved in this project:

- **Large Try/Finally:** We add a try/finally block around the entire function to make sure that we can catch any items before the function exits. There is a limit on the size of such blocks that is exceeded by some functions. We resolved the problem by creating a second function that calls the original functions (and placing the try/finally in the second function). This is only done for large functions (which are rare).
- **Constructor/Clone:** Changed our approach to instrumentation to no longer create a static call that we used to replace constructor/clone calls. This caused problems in some special cases. Instead we insert a summary (for taint transfer) before calls to clone/constructors.

This ensures that the call is made in the same context as originally.

- **Support External DEX files:** By default, Soot only instruments DEX files in the root directory of the APK. Some applications put additional DEX files in other locations (such as in assets). We now ensure that all DEX files in the APK are instrumented.
- **Signature Transparency:** We fixed a problem where signatures were not transparent to the application.
- **Reflection:** We fixed a problem so that reflection across the instrumented/uninstrumented boundary works correctly.
- **ClassLoaders:** We fixed a problem where class loader code gets called between pushing tag information into TLS and reading the tag (normally we presume that no code can execute between calling a function and entering that function. If, however, a new class is loaded as part of the call, the classes static constructor and application specific class loaders could get called. We added appropriate checks for this.
- **Infinite Recursions:** The system would infinitely recurse on calls to super if classes we instrument are extended by the application. We resolved this by not instrumenting calls to super (which is not required for our instrumentation)
- **Verify Errors with Monitor Instructions**. Java needs to create try/finally blocks around any critical sections to ensure that the monitor is released. Our instrumentation sometimes replaces code that could never throw an exception (e.g., return) with code that could (e.g., a method call). However, our code will never actually return an exception. We resolve this by always surrounding our code with empty try/catch blocks so that we don't engage the faulty soot code that attempts to deal with these changes.
- **Interning:**. Added support for interning strings and other primitive values. This is necessary because, in some circumstances, the system will intern objects with identical contents into a single object. Since those contents may have different tags, we would lose taint precision when this occurs. We handle this by leaving the objects with identical content as separate objects and overriding == such that the equals operation will yield the correct result (by comparing the interned values).
- **Compressed Resources**. We were incorrectly compressing some resources that shouldn't be compressed (like MP3s). Compressing them led to problem where they couldn't be mapped into memory and directly used. We fixed this to only compress when appropriate.
- **Package Signatures**. Fixed a problem where the package's signature was being applied to all packages queried and not just the current package.
- **Handle Large Methods**. Since we wrap the entire instrumented method body with an exception, it's possible that the method could become too large, since traps can only pack an unsigned short-sized number of instructions into it. For these, we push the instrumented method body into a separate method that is called from the original. The trap can therefore simply be wrapped around this method call. However, we needed to make these calls private, otherwise they could turn into a polymorphic call which could lead to indirect infinite recursion on overridden calls that contain calls to the super. We eliminate this problem by

making the calls private, since they would necessarily turn into a direct call.

- **Extendable Summaries**. Added "extendable" summaries for Map and IdentityHashMap which potentially replaces object references with their interned value, so that data structures using the '==' operator (as opposed to .equals()) will work as expected. Also fixed the analysis for applying "extendable" summaries, since it was only considering calls that explicitly override. Whereas it should consider any descendant call on that method.

We fixed a number of corner cases in Soot where it wasn't creating correct DEX output. These are not related to our instrumentation but show up in a Soot identity transformation (one without any instrumentation) These include:

- **Removed verify errors from unnecessary casts**. Soot's type assigner will occasionally insert casts if, for example, a register holding a primitive can also at some point hold an Object. Dead assignment elimination will normally find constants that are loaded into these registers that are simply dangling due to copy propagation. However, if there is a cast, then it will fail to do so since casts have the potential side-effect of throwing an exception (unless it's an Object typecast of the null constant). To eliminate this problematic jimple instruction, we mark casts that were inserted by soot, and in the dead assignment eliminator we consider these casts to be non-essential and the dangling assignments will be removed (thus preventing verify errors created by soot).

- **Constant Class Eliminator**. Modified the constant class eliminator jimple transformation to consider other types of primitive casts of the form:

  ```
  $x = (int) null
  ```

  This is because const_0 can refer to either a primitive or null constant in Dalvik (unlike Java bytecode), and in some cases where the type assigner is able to figure this out, it may insert these casts (if a register can hold a primitive or object reference, for example). These will be converted to:

  ```
  $x = 0
  ```

  Then we re-run the copy propagator transformation to copy propagate these assignments.

- **Fixed problem with long jumps**. When translating jimple back to DEX, the long jumps have to either be replaced with jumps where the offset has greater precision than the instruction will allow (in the event that the offsets dramatically increase) or be corrected with a series of conditionals and gotos (in the case where there is no simple replacement). There is an optimized path that simply replaces the instruction with a GOTO of higher precision, however this was incorrectly being utilized for conditional jumps as well.

- **Trap Minimizer**. The trap minimizer assumed that class constants loaded into a register cannot throw an exception, which is not true at all. This was leading to semantically incorrect behavior, since an essential trap was being removed in the Dalvik of some APK that at some point was loading a class constant of a class that was neither a system class, nor was defined in the DEX of that APK.

- **Verify Errors:** There are a number of verify errors in the application set. In general, these

don't seem to cause problems with running programs (because the code in question is not executed), but we are working on fixing these. These errors are a result of using the Soot tool (a Soot identity transformation has the same errors). Newer versions of Soot fix some of these problems and introduce new ones. Almost all of the problems seem to result from handling exception blocks correctly. We have discovered a set of commits that seems to resolve the problems without adding new errors that allows about 95% of the applications to verify correctly.

4.0 RESULTS AND DISCUSSION

We verified the correct operation of the system for both compatibility and functionality. We also measured overhead using the standard CaffeineMark [13] tests.

4.1 Functional Tests

We wrote 353 unit tests for the system that tested each piece of functionality for correctness. We test for compatibility with proxies, and other transparency tests such as reflection and Android Package (APK) introspection, as well as concurrency and making sure we haven't broken interning. We test taint tracking through various classes such as java.lang.Math, StringBuilder, StringBuffer, String, regular expressions, primitive boxing/unboxing, serialization, various interprocedural tagging configurations (between tracked/untracked calls), and arrays. We have tests for a few failing CTS tests, as well as tests for the optimizations. We have tests for dynamically loading instrumented as well as loading uninstrumented classes. There's also tests for the tag reporter as well as the event marking mechanism. The remaining tests check the trigger points (i.e. network, microphone, clipboard, calendar, location, read/write info classes, sensor, and activity).

4.2 Android Compatibility

We ran the Android Compatibility Test Suite [14] against instrumented Java code. There are 53,668 tests and we pass all but 58 of those tests. Most of the failures are corner cases due to small discrepancies introduced by the instrumentation that are not typically relevant to a running application. For example, signing differences, size differences, and in some cases different identify comparisons.

4.3 Application Compatibility

We ran the system against 90 common applications and tested to ensure that the applications ran correctly. 73 of the applications worked without issues. The details of the results by applications are in Tables 4.1 and 4.2. In the 'RUNS' column an 'X' indicates success. The other codes are:

*	Broken on vanilla	2 apps
***	Broken on soot identity	2 apps
!	Factory image is needed to run	5 apps
?	Broken version of `android.content.pm.PackageManager.getPackageInfo()` is required to run	2 apps
L	Loads, but requires an app update to run	5 apps
P	Requires Phone number (our test phones didn't have a cellular connection) try again on a pixel with a SIM)	2 apps
S	Resigning causes authentication issues preventing the app from logging in	6 apps
N	Safety net might be causing the 'L' or 'S' problems	n/a

Table 4.1: Tested Applications (A-P)

APK	Instruments	Verify Errors	Runs
8_Ball_Pool_v5.0.1	Yes	0	Yes
APKPure-Temple_Run_2_v1.58.0	Yes	0	Yes
Amazon_Prime_Video_v3.0.253.188041	Yes	0	X
Amazon_Shopping_v18.13.0.100	Yes	0	X
Android_Auto_Google_Maps_Media_Messaging_v5.4.502264-release	Yes	0	SN
Antivirus_Free_2019_Scan_Remove_Virus_Cleaner_v1.3.5	Yes	0	X
Aquapark_io_v0.7	Yes	0	X
Archero_v2.3.1	Yes	0	X
BitLife_Life_Simulator_v1.10.1	Yes	0	X
Bling_Launcher_Live_Wallpapers_Themes_v1.3.0	Yes	0	X!
Block_Hexa_Puzzle_tm_v1.5.44	Yes	0	X
Bottle_Flip_3D_v1.25	Yes	0	X
Clean_Road_v1.5.5	Yes	0	X
Coin_Master_v3.5.163	Yes	0	X
Color_Bump_3D_v1.2.2	Yes	0	X
Color_Flash_Launcher_Call_Screen_Themes_v1.2.0	Yes	0	X!
Color_Hole_3D_v1.1.2	Yes	0	X
Crowd_City_v1.3.0	Yes	0	X
Dancing_Road_Color_Ball_Run_v1.6.0	Yes	0	X
Discord_Chat_for_Gamers_v9.0.9	Yes	0	X
Drink_Water_Tracker_Water_Reminder_Alarm_v1.1.0	Yes	0	X
Drum_Pad_Machine_Beat_Maker_v2.1.0	Yes	0	X
Facebook_Lite_v154.0.0.7.120	Yes	0	X
Flip_Dunk_v1.41	Yes	0	X
Fun_Race_3D_v1.1.5	Yes	0	X
Gardenscapes_v3.5.0	Yes	0	LN
Google_Play_Games_v2019.06.11077-257270202.257270202-000406-apkpure.com	Yes	0	-*
Google_Translate_v6.0.0.RC07.257066911	Yes	0	X?
Granny_v1.7.3	Yes	0	X
Hole_io_v1.6.2	Yes	0	X
Homescapes_v2.7.2.900	Yes	0	LN
House_Paint_v1.3.4	Yes	0	X
Hulu_Stream_TV_shows_hit_movies_series_more_v3.68.0.308290	Yes	0	X
Idle_Theme_Park_Tycoon_Recreation_Game_v2.4.2	Yes	0	X
Instagram_v101.0.0.15.120	Yes	0	X
Jelly_Shift_v1.6.0	Yes	0	X
Jetpack_Jump_v1.2.3	Yes	0	X
Kick_the_Buddy_Forever_v1.2	Yes	0	X
Life360_Family_Locator_GPS_Tracker_v19.0.0	Yes	0	X
Lucky_Day_Win_Real_Money_v6.1.1	Yes	0	SN
Messenger-Text_and_Video_Chat_for_Free_v223.0.0.11.119	Yes	1	X
Messenger_Lite_Free_Calls_Messages_v62.2.1.14.283	Yes	0	X
Microsoft_Outlook_v4.2032.2	Yes	0	X
Netflix_v6.26.1_build_15_31696	Yes	0	-***
News_Break_Local_Breaking_v5.2.1	Yes	0	X
OfferUp_Buy_Sell_Offer_Up_v3.20.1	Yes	0	X
Paint_By_Number_Free_Coloring_Book_Puzzle_Game_v2.22.1	Yes	0	X
Party_io_v1.6	Yes	0	X
PayPal_Mobile_Cash_Send_and_Request_Money_Fast_v7.10.0	Yes	0	P
Pinatamasters_v1.2.5	Yes	0	X!
Pinterest_v8.31.0	Yes	0	X
Pluto_TV_It_s_Free_TV_v3.7.0	Yes	0	X
Pokemon_GO_v0.147.1	Yes	0	LN
Polysphere_v1.4.2	Yes	0	X!
Poshmark_Buy_Sell_Fashion_v3.06.01	No	-	-
Pottery_v1.4.5	Yes	0	X

Table 4.2: Tested Applications (R-Z)

APK	Instruments	Verify Errors	Runs
Roblox_v2.450.411874	Yes	0	X
Roku_vv6.0.8.248598	Yes	0	X
Roller_Splat_v1.7	Yes	0	X
Run_Race_3D_v1.1.8	Yes	0	X
SmartNews_Breaking_News_Headlines_v5.5.4	Yes	0	X
Snapchat_v10.61.0.0	Yes	0	SN
Snowball_io_v1.2.14	Yes	0	X
SoundCloud_Music_Audio_v2019.06.24-release	Yes	0	X
Stack_Ball_Blast_through_platforms_v1.0.46	Yes	0	X
Subway_Surfers_v1.105.0	Yes	0	X
Tank_Stars_v1.3.1	Yes	0	X
TextNow_Free_Texting_Calling_App_v6.33.1.0	Yes	0	SN
The_Mighty_Quest_for_Epic_Loot_v1.0.5	Yes	0	L
Tiles_Hop_EDM_Rush_v3.0.4	Yes	0	X
Tomb_of_the_Mask_v1.3	Yes	0	X!
Touch_The_Wall_v1.1	Yes	0	X
Township_v6.7.0	Yes	0	LN
Traffic_Run_v1.6.6	Yes	0	X
Train_Taxi_v1.2.5	Yes	0	X
Tubi_Free_Movies_TV_Shows_v2.21.2	Yes	0	X
Twist_Hit_v1.8.8	Yes	0	X
Twitter_v8.58.0-release.00	Yes	0	X
Walmart_v19.26	Yes	0	X
Waze_GPS_Maps_Traffic_Alerts_Live_Navigation_v4.58.0.1	Yes	0	X
Weather_radar_and_live_maps_The_Weather_Channel_v9.10.0	Yes	0	X
WhatsApp_Messenger_v2.20.194.16	Yes	0	P
Wish_Shopping_Made_Fun_v4.31.5	Yes	0	SN
Word_Stacks_v1.6.0	Yes	0	X
Words_Story_Addictive_Word_Game_v1.7.0	Yes	0	X
Wordscapes_v1.1.3	Yes	0	X
Wrecking_Ball_v0.3.0	Yes	0	X
YOLO_Anonymous_Q_A_v1.0.1	Yes	0	S
YouTube_Music_Stream_Songs_Music_Videos_v3.23.52	Yes	0	X?
ZEDGE_tm_Wallpapers_Ringtones_v6.8.3	Yes	0	-***

4.4 Overhead Results

We measured the overhead of our system using CaffeineMark [13]. Tracking taint and the other modifications can be expensive. Table 4.3 shows the raw overhead numbers (bigger is better) and Table 4.4 shows the percentage overhead. Each table shows how the performance improves by applying the optimizations we developed for the project for each of the benchmark tests.

Note that the *method* test runs faster than the original. This is due to some built-in optimizations in the dex -> jimple -> dex translation (not our optimizations).

The *double* test is our worst performing benchmark. This test contains triply nested loops over tainted values that our optimizations are unable to fully handle. But the overall overhead is still quite reasonable.

The settings for optimizations are:

- **-opt**: Replaces local StringBuffer allocations with their respective StringBuilder calls. It will also hoist loop invariants and attempt to find trivial flows through method calls.
- **-opt2**: All of the optimization from -opt. Unrolls the first iteration of the loop so that only the first iteration will propagate tags.
- **-opt3**: All of the optimizations from -opt and -opt2. Attempt to propagate field accesses in non-synchronized methods to the tops of the method (note this is optimization might lose taint in some corner cases)

More information on Saran's optimizations can be found in Section 3.11

Table 4.3: Raw Overhead

benchmark	orig	instr	instr -opt	instr -opt2	instr -opt3
sieve	110632	1102	109320	108946	110560
loop	208815	671	1028	1021	135395
logic	771054	289813	764206	760409	760596
string	221776	4106	23521	164798	165046
double	116016	566	675	15304	19736
method	58434	368	132479	132218	131983
OVERALL	172970	2383	23775	55172	130303

Table 4.4: Relative Overhead

benchmark	instr	instr -opt	instr -opt2	instr -opt3
sieve	100x	1.01x	1.02x	1x
loop	311x	203x	205x	1.54x
logic	2.66x	1.01x	1.01x	1.01x
string	54x	9.43x	1.35x	1.34x
double	205x	172x	7.58x	5.88x
method	159x	.44x	.44x	.44x
OVERALL:	72.6x	7.28x	3.14x	1.33x

4.5 Instrumenting Obfuscated Code

We tested the tool to make sure that obfuscation techniques (such as renaming classes, fields, and methods) and other modifications to the dex code (such as removing unused/duplicated code, peephole optimizations, variable allocations, inline constants, etc) do not degrade the capabilities of the tool.

All of our regression tests pass both with and without such obfuscations as implemented by the obfuscation tools Proguard and Allatori.

4.6 Deliverables and Artifacts

We delivered documented source code and build tools for Saran. We also delivered a manual, an example program, and unit tests.

5.0 CONCLUSION

Our evaluation of Saran prototype demonstrated the usefulness of our approach. Various triggers can easily be added without access to source code. These triggers are transparent to the program itself. These can be used for debugging, auditing, and understanding.

In detail, we evaluated Saran using ten distinct triggers (e.g. read/write files, send/receive Intent messages, read location information, etc.,) and three distinct actions (reads, writes, callbacks). We used our Instrumentation Application Programming Interface (API) to implement several security policies that we evaluated on benchmark applications (e.g., Gather App) derived from the Defense Advanced Research Projects Authority (DARPA) Transparent Computing program.

Inside the Department of Defense (DoD), we envision that Saran will be used to transparently and efficiently retrofit Android applications with security controls that meet the requisite DoD policies. This, in turn, will enable the DoD to use off-the-shelf applications thus significantly reducing the acquisition costs of secure mobile software. Specifically, we expect that the system will see use by the services within the laboratory environment (e.g., Space and Naval Warfare Systems Center (SSC) Pacifics Combined Test Bed) or a simulated operational environment.

In the commercial sector, the ability to transparently retrofit Android applications with custom security controls will enable companies meet both security and compliance needs. For example, Saran may enable companies to use existing applications, with additional instrumentation, such that they meet HIPAA compliance needs.

6.0 REFERENCES

[1] Vallée-Rai, Raja, Co, Phong, Gagnon, Etienne, Hendren, Laurie, Lam, Patrick, and Sundaresan, Vijay, "Soot-a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, IBM Press, p. 13

[2] Perkins, Jeff, Eikenberry, Jordan, Coglio, Alessandro, Willenson, Daniel, Sidiroglou-Douskos, Stelios, and Rinard, Martin, "AutoRand: Automatic Keyword Randomization to Prevent Injection Attacks," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, Springer-Verlag New York, Inc., New York, NY, USA, DIMVA 2016, pp. 37–57, URL http://dx.doi.org/10.1007/978-3-319-40667-1_3

[3] ASM, "OW2 Consortium," http://asm.ow2.org/, URL http://asm.ow2.org/

[4] Binder, Walter, Hulaas, Jarle, and Moret, Philippe, "Advanced Java Bytecode Instrumentation," in *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, ACM, New York, NY, USA, PPPJ '07, pp. 135–144, URL http://doi.acm.org/10.1145/1294325.1294344

[5] Tanter, Éric, Ségura-Devillechaise, Marc, Noyé, Jacques, and Piquer, José M., "Altering Java Semantics via Bytecode Manipulation," in *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, Springer-Verlag, London, UK, UK, GPCE '02, pp. 283–298, URL http://dl.acm.org/citation.cfm?id=645435.652652

[6] Enck, William, Gilbert, Peter, Chun, Byung-Gon, Cox, Landon P., Jung, Jaeyeon, McDaniel, Patrick, and Sheth, Anmol N., "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of OSDI 2010*, URL http://appanalysis.org/tdroid10.pdf

[7] Qian, Chenxiong, Luo, Xiapu, Shao, Yuru, and Chan, Alvin TS, "On tracking information flows through jni in android applications," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, IEEE, pp. 180–191

[8] Yan, Lok-Kwong and Yin, Heng, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis." in *USENIX security symposium*, pp. 569–584

[9] Tam, Kimberly, Khan, Salahuddin J, Fattori, Aristide, and Cavallaro, Lorenzo, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors." in *NDSS*

[10] Sun, Mingshen, Wei, Tao, and Lui, John C.S., "TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, New York, NY, USA, CCS '16, pp. 331–342, URL http://doi.acm.org/10.1145/2976749.2978343

[11] Xu, Rubin, Saïdi, Hassen, and Anderson, Ross, "Aurasium: Practical Policy Enforcement for Android Applications," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, USENIX, Bellevue, WA, pp. 539–552, URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/xu_rubin

[12] Backes, Michael, Bugiel, Sven, Hammer, Christian, Schranz, Oliver, and von Styp-Rekowsky, Philipp, "Boxify: Full-fledged App Sandboxing for Stock Android," in *24th USENIX Security Symposium (USENIX Security 15)*, USENIX Association, Washington, D.C., pp. 691–706, URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/backes

[13] Pendragon Software Corp, "CaffeineMark 3.0," http://www.benchmarkhq.ru/cm30/, URL http://www.benchmarkhq.ru/cm30/

[14] Google, "Android Compatibility Test Suite," https://source.android.com/docs/compatibility/cts, URL https://source.android.com/docs/compatibility/cts

7.0 LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

API Application Programming Interface. 1, 7, 8, 9, 34

APK Android Package. 3, 4, 5, 6, 9, 25, 26, 28

DARPA Defense Advanced Research Projects Authority. 3, 7, 34

DEX Dalvik Executable. 2, 3, 5, 6, 7, 22, 25, 26

DIFT Dynamic Information Flow Tracking. 5, 6

DoD Department of Defense. 1, 34

IP Internet Protocol. 10, 12, 18, 20

IPC Interprocess Communication. 6

RPC Remote Procedure Call. 3, 6

TLS Thread Local Storage. 19, 20, 21, 25