A5: Automated Analysis of Adversarial Android Applications

Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, Patrick Tague
ECE/CyLab
Carnegie Mellon University

Working paper

Technical Report CMU-CyLab-13-009
First version: February 21, 2013
This version: June 3, 2014

Abstract

Mobile malware is growing – both in overall volume and in number of existing variants – at a pace rapid enough that systematic manual, human analysis is becoming increasingly difficult. As a result, there is a pressing need for techniques and tools that provide automated analysis of mobile malware samples. We present A5, an automated system to process Android malware. A5 is a hybrid system combining static and dynamic malware analysis techniques. Android's architecture permits many different paths for malware to react to system events, any of which may result in malicious behavior. Key innovations in A5 consist in novel methods of interacting with mobile malware to better coerce malicious behavior, and in combining both virtual and physical pools of Android platforms to capture behavior that could otherwise be missed. The primary output of A5 is a set of network threat indicators and intrusion detection system signatures that can be used to detect and prevent malicious network activity. We detail A5's distributed design and demonstrate applicability of our interaction techniques using examples from real malware. Additionally, we compare A5 with other automated systems and provide performance measurements of an implementation, using a published dataset of 1,260 unique malware samples, showing that A5 can quickly process large amounts of malware. We provide a public web interface to our implementation of A5 that allows third parties to use A5 as a web service.

1 Introduction

The number of applications available for mobile phones and tablets has surged dramatically over the past couple of years; this trend has been particularly pronounced for Android devices, that now represent 73% of all mobile devices [17]. Concomitant with this rise in the number of applications available, malware targeting mobile platforms, and specifically Android, has also started to appear [16, 32, 39]. Even though industry reports of "exponential growth" in mobile malware [13, 23] must be taken with a grain of salt [22] there is little doubt that the overall volume of mobile malware is increasing at a pace that makes it difficult to sustain systematic manual analysis. It is, therefore, important to develop automated analysis capabilities for mobile malware.

Detecting, analyzing and combating mobile malware presents a number of unique challenges. First, different from the situation with personal computers, users generally do not have full administrative access to their mobile device, which makes it much more challenging to develop effective anti-virus tools. Second, carriers and network operators, who can fairly tightly control the network, may have only limited capabilities to control individual devices. Third, techniques useful to "sandbox" potentially harmful applications, such as virtualization, are much less mature on mobile devices than they are on PCs.

These unique challenges suggest that traditional malware analysis and detection methods need to be rethought in the context of mobile devices. For mobile devices, network-based identifiers (e.g., network traffic patterns) are considerably more actionable than host-based identifiers (e.g., writing a specific file). Indeed, a carrier or operator could easily disconnect, and potentially reset, a mobile device that produces suspicious network traffic. On the other hand, detecting, on the device itself, that an application is malicious is much more complex without elevated privileges. In other words, given the current administrative models, network-based intrusion detection systems appear considerably more useful to mobile devices than their host-based counterparts.

We use these insights to propose "A5," short for Automated Analysis of Adversarial Android Applications. A5 is a system that draws conceptual design from existing dynamic analysis (or "sandbox") systems. At a high level, A5 executes malware in a sandbox environment that consists of a combination of physical devices and virtual Android systems hosted on a PC. A5 allows malware to connect to the Internet, in order to record network threat indicators and create network intrusion detection system (IDS) signatures. These signatures can in turn be used by an enterprise to protect mobile devices that connect to the Internet through a corporate network or to protect all corporate devices by forcing mobile device traffic through a network proxy. Similarly, cellular providers could use these signatures to protect devices connected to carrier networks. We provide a web-based interface to our current implementation of A5 at `http://dogo.ece.cmu.edu/a5`.

The key novelty in A5 is to use a combination of static and dynamic analysis to *coerce* the application into triggering its malicious behavior. Indeed, in mobile applications, activity can be triggered by a wide assortment of system events – for instance, receiving a phone call, or having the screen go into lock mode. A5 attempts to exhaustively determine all possible paths that can trigger malicious behavior, before separately evaluating them. Doing so, A5 can capture activity that would be missed by naïvely executing the malware (i.e., simply "clicking on the icon"). Furthermore, by combining physical devices with virtual Android images, A5 can capture a wider range of malicious behavior than a sandbox solely based on emulation would and can correctly process malware that employs certain type of sandbox evasion techniques. Likewise, A5 can accommodate a wide range of different hardware and software (e.g., SDK) configurations.

In the reminder of this paper, we first introduce background on static and dynamic analysis in section 2, where we also differentiate A5 from the relatively large body of related work on Android security. We then describe the design and architecture of A5 in section 3. We present a performance evaluation of our current implementation of A5 in section 4, notably showing that, using parallelism, A5 is able to analyze 1,260 unique malware samples in just over 10 hours. We discuss A5's limitations in section 5, and draw conclusions in section 6.

2 Background and Related Work

Without access to source code for analysis, inspection and understanding, one must resort to other techniques when analyzing compiled software. In the context of malware analysis, *dynamic analysis* involves executing the malware samples to observe their behavior [25]. Conversely, *static analysis* refers to techniques that inspect or process a sample, but never execute the malware [14]. Manual, static analysis, colloquially known as as "reverse engineering," can be very effective, but often requires highly trained individuals and is time consuming. Thus, it is difficult to scale manual analysis at the pace that mobile malware is growing – both in terms of volume and in number of existing variants [13, 16, 23, 32, 39].

A dynamic analysis technique often used in vulnerability discovery can be automated to process input to samples automatically. *Fuzzing* is the process of sending data as input to a program, possibly intentionally invalid data, in order to coerce a desired condition or behavior. The input can be created programmatically to cover a range of inputs, and in this way can be thought of as a brute-force attack against the software. This technique may be considered inelegant, but fuzzing implementations are often straight-forward, and effective. Fuzzing is used in automated vulnerability discovery to find software vulnerabilities that are not feasible to audit in any other way [29].

Malware sandboxes. Malware sandboxes automate dynamic analysis techniques to inspect large volumes of malware automatically. The general operation of a sandbox system is to execute each input sample much like a user would, but in a controlled environment instrumented to monitor host and network activity. The sheer volume of unique malware samples on traditional computers makes the use of automated sandboxes appealing. Numerous commercial products, such as CWSandbox [37], and academic projects, such as ANUBIS [6], have appeared over the past several years. Automated sandboxes often scale linearly with computational power. A sandbox addressing computer malware may boot a virtual machine, copy the samples to the virtual machine, then execute the sample. The sandbox can monitor and report on changes to the host (i.e., registry keys, files) and network communications. For instance, Rossow el al. present a dynamic analysis system called Sandnet [25], which is used to collect network traffic from PC malware samples. Sandnet is used to process 100,000 samples and the authors find that DNS and HTTP have novel trends in malware use.

Malware analysis systems for Android. The work most related to A5 is a dynamic analysis system called Andrubis [2]. Andrubis is an extension to the automated PC malware analysis project ANUBIS, but is designed for processing Android packages. The inner-workings of Andrubis are not publicly known, but the creators allow anyone to interact with a public interface via website. Blasing et al. [7] describe another dynamic analysis system for Android. Their system focuses on classifying input applications as malicious (or not). The system instruments Linux features and scans applications for the use of potentially dangerous criteria. Like Andrubis, this system interacts with the malware by starting the application's primary Activity.

A5 differs from these two systems primarily in the way A5 interacts with the malware—using multiple techniques to coerce the execution of the malicious code. However, there are several other differences such as the parallel implementation of A5, support for every Android API version, and the ability to use virtual instances, physical devices or both.

DroidBox [21] is a generic app monitoring tool for Android apps. It monitors an Android app for various activities at runtime, such as incoming and outgoing network data, file read and write operations, services started, etc. It then provides a timeline view of the monitored activity from the app. DroidBox is useful for manually identifying malware by viewing its observed behavior. Compared to A5, DroidBox does not automatically coerce the app into undertaking particular behaviors, and A5 specifically captures network traffic for finding malicious network indicators. In addition, A5 uses static-analysis in addition to dynamic monitoring of the app to find coercion points automatically.

Similar to A5's bytecode static-analysis, ComDroid [8] performs static-analysis of decompiled bytecode of Android applications. ComDroid performs flow-sensitive, intra-procedural analysis to find Android "Intents" sent with weak or no permissions—but contrary to A5, ComDroid does not perform any dynamic analysis.

A5 currently only captures network traffic to aid in finding malicious network indicators. It may make sense to pair A5 with taint tracking systems such as TaintDroid [10] in order to track host-based malware indicators. For instance, Andrubis employs TaintDroid. However, it may take significant effort to extend TaintDroid to support all SDK target versions and to work with a range of physical devices, as A5 does right now.

Automated signature creation. Automating the tedious and error-prone process of creating network IDS signatures is a well researched topic but remains an open problem. As a representative example, Kim and Karp create an automated system call Autograph that generates signatures for TCP-based Internet worms [19]. Like many efforts at automatic signature creation, Autograph's detection mechanisms are particularly designed to address one type of malware, in this case worms. As such, Autograph's pre-filtering step that discerns unsuccessful TCP connections, is not particularly useful for identifying malicious Android application traffic. A different system called Honeycomb presents similarities to A5's desired goal of automatically creating IDS signatures. Kreibich and Crowcroft describe the system which collects traffic from a honeypot and subsequently creates network signatures [20]. Since the network traffic is captured from a honeypot, the traffic is assumed to be malicious (or at least suspicious). A5 similarly assumes that all input is malware, but due to the repackaging common in Android malware, malicious network traffic is likely to be mixed with benign traffic.

3 A5 architecture

The immediate need for a system like A5 is driven by increasing volumes of mobile malware. However, the design of A5 is also directed by several criteria borrowing from the more mature field of PC-based dynamic analysis and the unique nature of today's mobile device ecosystem. Here we enumerate a list of desired features for such a system, and describe an implementation designed to meet these goals.

3.1 Objectives and Design

Autonomy and scalability. The system must be able to handle volumes of malware without user interaction. As with PC malware, mobile malware is now growing at a rate that makes manual, human analysis unfeasible.

Evasion resistance. The system must be able to adapt to evasion advances in malware. As seen in the PC, mobile malware is increasing in sophistication. With the advent of automated malware processing, malware authors have already begun to include minor attempts to evade sandbox systems.

Mobile-specific interaction. The system must interact with malware in Android-specific ways. It is indeed more difficult to solicit malicious behavior from current mobile malware, than traditional malware. Simply executing the malware (i.e., "clicking on the icon") may not exhibit any malicious behavior. Indeed, current mobile operating systems permit applications to register a software handler for a wide range of system events; for instance, receiving an SMS, screen going in lock mode, and so forth. Any such event may trigger some application code. Traditional computer programs may receive input along with execution; mobile applications may receive input along with a myriad of system events. In either case, the behavior may depend upon the input to the application.

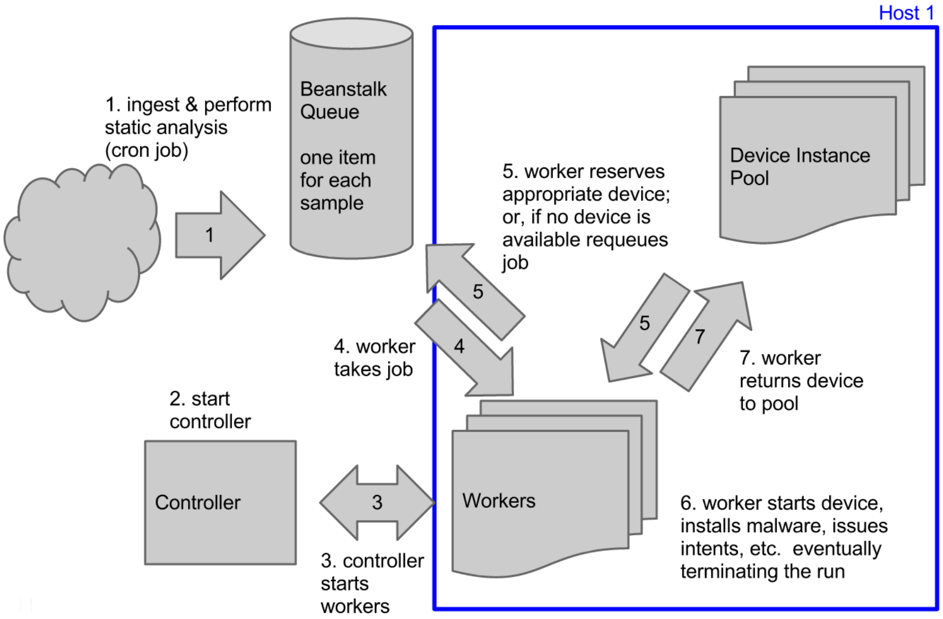
Host 1
1. ingest & perform static analysis (cron job)
1
Beanstalk Queue
one item for each sample
5. worker reserves appropriate device; or, if no device is available requeues job
5
Device Instance Pool
4. worker takes job
4
5
7
7. worker returns device to pool
2. start controller
Controller
3
Workers
6. worker starts device, installs malware, issues intents, etc. eventually terminating the run
3. controller starts workers

Figure 1: A5 architecture. Malware is first ingested (1) into a shared job queue. Independently, an overall controller is started (2) which starts one or more Worker processes (3). Each Worker retrieves jobs from the queue (4) and either postpones work or reserves a device instance (5) for dynamic analysis (6). Once analysis is complete, the device is returned to the ready pool (7). There may be many Workers and Device Pools on a single host. The controller and job queue may service many hosts (each with many Workers).

Network-level indicator collection. The system should primarily collect network threat indicators. Host-based indicators, such as the modification of a file found on the device, are of limited value on Android. Indeed, since Android's architecture does not permit file system hooks, and, more generally does not even permit privileged access to most components of the system, it is not possible to implement controls similar to anti-virus products found on the PC. Even if Android's architecture were adapted to permit such products (e.g., by systematically "rooting" devices), network indicators are particularly useful to cellular carriers and/or wireless network operators to protect the device even without the ability to install controls on the device itself.

Modularity. The system should have a modular, expandable design. Mature analysis systems need to have interfaces allowing for the system to interact with other software systems such as intrusion-detection systems (IDS) or firewall management tools. This requirement is generally driven by entities that have larger research and analysis environments of which A5 may become a component. Additionally, the system must be generally able to adapt to unforeseen circumstances, such as malware that exhibits some new behavior or technology that was not yet imagined when the system was designed.

Based on these objectives, we made the following design choices for the A5 architecture. To process as much malware as possible, A5 is highly parallel and distributed. The basic steps involved are shown in Figure 1 and detailed in the following sections. A5 consists of a queue, a main controller, and a set of workers which interact with a pool of device instances – these device instances are a combination of hardware resources (e.g., a specific phone model), and Android images running as virtual machines on a traditional PC.

First, malware is moved into the system and two stages of static analysis are performed to determine methods of interacting with the malware. Once static analysis is complete, an entry is created in a job queue for subsequent dynamic analysis. Later, one of many worker processes retrieves the job from the shared queue and executes the malware using an available device from the device pool. The dynamic analysis is informed from the static analysis; this combination of static and dynamic analysis allows our system to better coerce malware to execute nefarious behavior.

The remainder of this section details the implementation of A5. In particular, each of stage 1 static analysis, stage 2 static analysis, and dynamic analysis are detailed. Then, the concept of device pools consisting of virtual and physical devices is described, followed by a discussion of some Android-specific interaction techniques.

3.2 Malware Ingestion

A5 assumes all input samples are malware. The primary functions of the ingestion process are to create a shared job queue entry, (we use beanstalkd[1]), to calculate several pieces of meta-data (such as cryptographic hash values), and to initiate the static analysis.

A5's ingest process is designed to run on each individual sample. This allows for on-demand submission, such as what one may expect of a web service, and as a batched process running periodically, consuming samples that are placed in a particular system path. This allows all of A5 to run perpetually with no interaction from a user. Many security companies receive thousands of samples daily from sources such as VirusTotal [35] or MWCollect [36]. These incoming samples can easily be sorted, for example, to collect all Android samples in one location for input into A5.

3.3 Static Analysis

A5 first resorts to static analysis to try to detect potentially malicious actions. In Android, applications are usually written in Java (less than 5% have "native" C components [40]), and are distributed as APK (Android package) files. These APK files are in fact Zip archives, which contain compiled Java classes (in Dalvik DEX format), application resources, and an `AndroidManifest.xml` binary XML file containing application meta-data. The structure of Android applications and the Android security mechanisms have been well-documented [12, 27] and many tools exist for creating and manipulating APKs [3, 5].

Typically, Android applications that have a user interface specify at least one Android *Activity* and those that do not have a user interface specify at least one *Service*. These are classes that typically contain the core functionality of the mobile application, and are the primary method for executing application code. Much of the interaction with an Activity will be through the Graphical User Interface (GUI). However, a Service may exhibit no GUI components at all, requiring different interaction during later dynamic analysis.

Android Inter-Process Communication (IPC) typically occurs in the form of an Android event known as an *Intent*. For instance, Intents are used to transfer information between applications and to notify applications when a particular system event, such as the receipt of a text message, has occurred. Since Android Services have no GUI, it is precisely these types of events that initiate a Service.

The chief output of static analysis is an enumeration of "interaction points" (e.g. Activities) and a set of "receivable intents" (e.g. BOOT_COMPLETED). Any of these may cause the application to take actions that would not normally occur if the application was simply launched using the graphical interface. As such, A5 will use these sets in order to coerce behavior from the malicious application. Many of these meet the need for better mobile-specific interaction.

[1] beanstalkd can be found at `http://kr.github.com/beanstalkd/` and is described as "a simple, fast work queue" originally designed to reduce latency on high-volume websites.

```
<receiver
   android:name=”.message.SmsReceiver”
   android:enabled=”true”
   android:exported=”true” >
  <intent-filter
     android:priority=”214783648” >
    <action
       android:name=”android.provider.Telephony.SMS_RECEIVED” >
    </action>
  </intent-filter>
</receiver>
```

Figure 2: Receiver from ANDROID-DOS malware. A5 notes the action for this receiver. Receipt of a text message may be the only way this method is executed. In other words, the `.message.SmsReceiver` code may never be invoked if the instance never receives a text message.

3.3.1 Stage 1 Static Analysis: AndroidManifest

Much of the stage 1 analysis in A5 revolves around the `AndroidManifest.xml` file. This file dictates much of how an application may interact with the device. Each application must advertise the desire to receive particular Intents by declaring permissions in the AndroidManifest. Similarly, through documentation [34] and source code analysis [15], use of certain API functions implies the ability to receive certain Intents.

Even though the manifest is stored in binary XML form, tools are readily available for parsing key components such as requested permissions, broadcast receivers, background services, and activities. Each of these components define key interaction points for the application, and are cataloged for later use in dynamic analysis.

For instance, an Android `BroadcastReceiver` or “receiver” is a way for an application to register the desire to receive an Intent from the system or another application . A receiver from recent Android malware is shown in Figure 2. A5 parses and saves the action from this portion of the manifest, in this case receipt of an SMS message. During dynamic analysis, the receipt of a text message may be the only action that invokes the `.message.SmsReceiver` method.

Instead of creating yet-another-tool to extract pertinent information, we elected to leverage an existing open source tool known as Androguard [1]. If Androguard did not support a particular function that A5 required, we implemented the feature and submitted patches back to the Androguard developers.

3.3.2 Stage 2 Static Analysis: Bytecode

In addition to the relatively naïve stage 1 analysis of the Android application manifest, we also analyze the Java bytecode of the application binaries. The goal of this stage 2 static analysis is to identify additional interaction points which enable users or the system to interact with the application. While many interaction points are declared in the application manifest, some may be created dynamically by the application, thus being missed by naïve analysis.

An example of an interaction point that may be missed during stage 1 is shown in Figure 3. The application in Figure 3 performs a `registerReceiver` call registering the desire to be notified when either the user begins interacting with the device or the device screen turns off. Neither of these Intents are found in the `AndroidManifest.xml`.

```
1 public static void h(Context paramContext)
2 {
3   IntentFilter localIntentFilter = new IntentFilter();
4   localIntentFilter.addAction(”android.intent.action.USER_PRESENT”);
5   localIntentFilter.addAction(”android.intent.action.SCREEN_OFF”);
6   paramContext.registerReceiver(new UserActivityReceiver(),
       localIntentFilter);
7   return;
8 }
```

Figure 3: Code section reverse engineered from a GoldDream malware sample. Here, the desire to receive USER_PRESENT and SCREEN_OFF Intents are registered dynamically - these Intents do not appear in the AndroidManifest.xml and would be missed by analysis techniques that do not incorporate bytecode level analysis.

```
1 Calendar c = Calendar.getInstance();
2 String bootc = ”android.intent.action.BOOT_COMPLETED”;
3 int seconds = c.get(Calendar.SECOND);
4 intentFilter bootcif = new intentFilter(bootc);
5 registerReceiver(bootcif);
```

Figure 4: Code section demonstrating the need to resolve variables. In this case the CFG is recursively traversed in order to find the value of bootcif at the time `registerReceiver` is called in line 5. In this case, A5 concludes that the program dynamically registered the desire to be notified when the system has finished booting.

The stage 2 static analysis algorithm is fairly intuitive. First, A5 invokes the DED [11] decompiler to create Java classes from the Android application code. Next, A5 uses Soot [30] to obtain an Intermediate Representation (IR) and a Control Flow Graph (CFG). This abstract IR is known as Jimple [31] and is useful because it eases the burden of dealing with the more complex Java bytecode.

Each node in the CFG represents one Java statement, and the graph edges correspond to the relationship between the statements in the malware. A5 traverses the CFG in order to find nodes that represent a known Android interaction point.

Each CFG node is further decomposed into an Abstract Syntax Tree (AST) representing individual components of the statement. Specifically, A5 looks for calls to `android.content.Context.registerReceiver()` and `android.app.Activity.startActivityForResult()`. Calls to `android.content.Context.registerReceiver()`, as shown in line 6 of Figure 3, result in the application becoming eligible to receive Intents with a specified Action (i.e. a particular string). Similarly, calls to `android.app.Activity.startActivityForResult()` result in the application making a call to another application, but with an embedded Intent for the callee to make a callback to the target application. When A5 discovers one of these calls, the CFG is recursively traversed in order to resolve variable definitions. These definitions must be resolved in order to capture Intents that represent interaction points. For example, Figure 4 shows a call to `registerReceiver` at line 5, however, the AST node only contains the component for variable `bootcif`. A5 recursively traverses the CFG to determine the variable definition at line 2.

Much like the stage 1 static analysis, the output of the bytecode static-analysis is a set of receivable Intents for use during the dynamic analysis.

3.4 Dynamic Analysis

The ingestion process and static analysis components execute relatively quickly, but the dynamic analysis portion is more time-consuming. Fortunately, it also lends itself to parallel execution. Figure 1 depicts many workers on a single A5 host. *Worker* processes on each A5 host retrieve jobs from the shared job queue for processing. Once a new job has been reserved from the queue, the Worker inspects a pool of candidate Android instances available to that particular host attempting to reserve a compatible instance. A compatible instance is one in which the malware sample is expected to run. For example, a mobile application that declares a minimum SDK (Android API) level of 8, will not run on a level 4 device. Even if the application were to be modified by A5 to specify level 4 prior to instance selection, the application may actually rely upon a feature not available until level 8. Assuming a compatible instance is available, the Worker continues with dynamic analysis. If no compatible device is available, the job is placed back into the queue with a delay in order to reduce the chances that Workers repeatedly reserve the same job when no compatible instance is available – effectively de-prioritizing other pending samples.

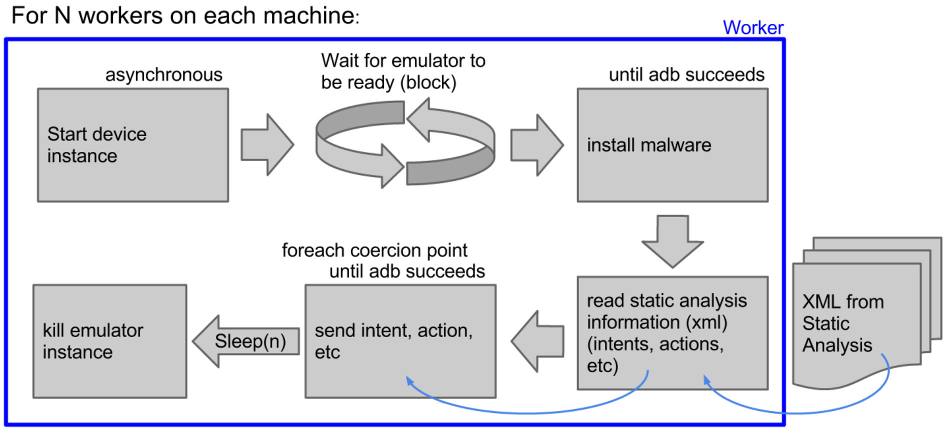
For N workers on each machine:
Worker
asynchronous
Start device instance
Wait for emulator to be ready (block)
until adb succeeds
install malware
foreach coercion point until adb succeeds
kill emulator instance
Sleep(n)
send intent, action, etc
read static analysis information (xml) (intents, actions, etc)
XML from Static Analysis

Figure 5: Worker process flow. Communication with the device instance is performed using the Android Debug Bridge (ADB), and output from the static analysis is utilized in dynamic analysis interaction.

The Worker then boots the instance and follows the process depicted in Figure 5. Communication with a running instance is performed with the SDK debugging tool known as the Android Debug Bridge (ADB)[2]. Since the boot may take some time, the worker initiates the boot process, then, using ADB, blocks until the device is fully booted. Once booted, A5 uses ADB to install the malware sample into the instance. Once the sample is installed, the Worker coerces malicious behavior from the instance, again using ADB. After a set period of time the Worker terminates the instance and returns it to a known state.

3.5 Instance Pools

When retrieving a new job, the Worker must locate a device instance compatible with the malware sample. For this reason, A5 maintains pools of devices on each host. Each instance in the pool may be a physical or virtual instance, as detailed below. Workers synchronize the use of instances by maintaining instance state in a data structure shared among Workers on each host.

Virtual instances have the benefits of being low-cost, easy to automate and generally flexible. On the other hand, virtual devices are not typically used in everyday computing, so malware that can detect the

[2] http://developer.android.com/tools/help/adb.html

virtual environment may elect to exhibit alternate behavior. In this light, the use of physical devices may be warranted.

3.5.1 Virtual Instances

Virtual instances in A5 are realized with modified versions of the emulator distributed with the Android SDK. These instances all are are stored and executed on commodity computer hardware. Using virtual instances allows A5 to scale easily by simply creating larger instance pools as needed. Resetting a virtual instance to a known state is as simple as starting the instance with a "wipe data" flag or deleting and recreating the instance image from scratch.

However the emulator offers a subset of features found on a real device. The lack of features can be coarsely grouped into two classes: features not implemented by the emulation system and software that is not present in the emulated device image. An example of the former is Bluetooth, which is not implemented in the emulator, but is present on most devices. An example of the latter is the Google Play application, which is pre-installed in nearly every Android device sold, but is not present in the default emulated device image.

The lack of features presents a fundamental problem to a dynamic analysis system: if malware makes use of one these missing features, the malware will not run properly. This change in behavior may be explicit (malware employs "virtualization detection") or implicit (the malware happens to try to use a missing feature). In either case, the desired behavior from the sample will not be realized.

On traditional computers, virtualization detection was initially not found at all. It was later employed more frequently to evade dynamic analysis systems: If the malware was detecting it was running on a virtual machine, the malware would demonstrate benign behavior. The increased use of virtualization detection by malware in turn led to creating dynamic systems employing physical machines [28]. However, as virtual machines and, more generally, virtualization, is increasingly employed on laptops and desktop computers, running in a virtual machine is not a give-away that the platform is attempting to analyze a piece of malware. In other words, new forms of malware may paradoxically employ virtualization detection *less* frequently.

In today's mobile computing paradigm, the devices physically move with the user and data bandwidth at any given time varies greatly. Contrary to traditional computing platforms, there is not yet a clear use-case for virtualization in typical end-user environments. Resources such as bandwidth or power are far more constrained and devices are typically not shared among multiple users. However, if systems like A5 are increasingly deployed – as we believe they will be – virtualization detection in mobile malware will become a reality.

Manual analysis of malware families in 2012 [39] revealed that the current generation of mobile malware does not yet employ virtualization detection. Even so, Vidas and Christin explored possible methods such detection might be implemented and provided a taxonomy of several methods [33]. Following this work, recent malware is starting to employ such detections "in-the-wild". For instance, a recent Android Malware (Jan 21, 2014), Android.hehe [9], implements two checks: (1) the nonexistance of an IMSI - a unique cellular subscriber numer and (2) the existance of `Build.` strings that are exactly "sdk" or "google_sdk" as shown in Figure 6. Similarly, Android Malware "Oldboot" (Apr 2, 2014) identifies is running location of the malware instance (`/sbin/meta_chk`) and exits if the path is not as expected or if there is no sim card present.

In A5, the emulator software is built from source, and subsequently the resulting emulator is similar to the emulator distributed with the binary Android SDKs. However, we enhanced the emulator to evade some virtualization detection features. For example, an unmodified emulator will always return the same values for APK calls such as `TelephonyManager.getDeviceId()` (all zero's) or `Settings.Secure.ANDROID_ID` (null). By modifying the virtual instances such that values indicating a physical rather than a virtual device is in use, A5 becomes less detectable by malware seeking to deter-

```
String v0 = TelephonyUtils.getImsi(((Context)this));
if(v0 == null) {
    return;
}

public static boolean isEmulator() {
    boolean v0;
    if((Build.MODEL.equalsIgnoreCase(”sdk”)) || (Build.MODEL.
        equalsIgnoreCase(”google_sdk”))) {
    v0 = true;
    }
    else {
        v0 = false;
    }
    return v0;
}
```

Figure 6: Simple evasions are starting to appear in real malware observed “in-the-wild.” This example is from Android.hehe malware, which attepts to evade analysis by detecting that an instance is an emulator via Build strings common in developer SDKs, and by checking for the lack of a device subsciber ID which is common in the Android emulator.

mine if execution is occurring in a virtual sandbox. A5 makes such changes in `Build` parameters, the `TelephonyManager` class, and the default networking configuration. A5’s emulator instances have configurable settings in these cases each instance returns values that simulate values observed on real devices.

However, even with modifications that make the emulated Android instance more like a physical device, there are large voids in the emulated environment. Malware need only check for one of the many hardware features not currently implemented such as Bluetooth, Wi-Fi, sensors, etc. These hardware features are very common in physical devices and are simply not present in the emulator. Implementing entire systems to emulate these is a large undertaking and has not been done as part of the current A5 implementation. For this reason, it is currently trivial for malware to detect that the malicious application is currently running in a virtual environment and not a real device. To address this issue, we complement our virtual environment with physical device pools.

3.5.2 Physical Instances

By using real, physical devices in A5 device pools, we prevent malware from being able to trivially determine from hardware presence that the sample is being processed by a dynamic analysis system. The real devices possess actual hardware for systems that are missing in the virtualized environment, such as Bluetooth. A5 systems relying upon physical instances can scale linearly by purchasing more devices.

Physical devices embody a wide range of software features and hardware capabilities. As with typical computers, more recent devices have more computing power and are distributed with more recent software. In order to process all samples with physical devices, at least one device is needed for every Android SDK target version.

Resetting a physical device to a known state is not as simple as resetting a virtual instance. Android devices do not typically have boot modes that can be controlled remotely, such as PXE found on many modern network adapters. In fact, typically the only boot-time interface to Android devices is the USB/charging port. Various manufacturers support proprietary flashing protocols, but some devices employ the more approachable *fastboot* protocol. Critically for A5, fastboot supports erasing from and writing to device data partitions. A device may enter fastboot mode prior to loading the operating system when a person uses a special hardware key combination. However, ADB can also be used to reboot a running device directly into fastboot mode. In this way A5 can programmatically return a physical device to known state prior to each execution. In order to write data to a partition, the device must be "unlocked." Manufacturers each have different positions on whether a consumer should have the ability to write data to a device. Developer-friendly devices, such as the Nexus-branded devices, can all easily be unlocked. Similarly, these devices also support fastboot making them ideal devices for automated use in A5.

Unlike virtual devices, physical devices are capable of actual physical medium communication. Physical devices can communicate over cellular, Wi-Fi, Bluetooth, NFC, etc. To permit devices to exhibit network behavior, A5 devices are configured to connect to a wireless access point that routes to the Internet. This wireless connection provides a method for for network package capture, but can also leak sensitive information or be used by a miscreant to attack a local resource. Furthermore, the wireless environment around the device may change due to outside circumstances. For example, a neighbor may install a new wireless access point to which the device may connect. For all of these reasons, physical device pools should be placed into an radio frequency (RF) isolated environment along with the routing access point.

3.6 Malware Interaction

Regardless of the instance type, once the malware is installed, A5 must interact with the instance to coerce the malicious behavior. Of course, A5 can, and does, start the main activity of applications that place an icon in the application list for users to click. However, A5 employs other techniques for interaction.

The primary method for interaction is via receivable Intents ascertained during stage 1 and stage 2 static analysis as described in sections 3.3.1 and 3.3.2. Using ADB, a Worker can send intents to a running instance. Intents are sent to the device sequentially and feedback from ADB can be used to verify that an Intent was successfully registered on the instance.

A5 could simply send every type of Intent available for the SDK version of the instance. However, this coarse style of fuzzing presents two problems: inability to discern custom Intents and poor performance. Each version of the SDK specifies dozens of Intents and permissions [4], simply iterating through all of these takes considerable time which would lower the overall performance of A5. More importantly, applications can specify custom Intents [4], which may not be known a priori. Figure 7 depicts the creation and receipt of custom Intent, `some.custom.intent.FOO`. For these reasons, A5 employs a more granular system, precisely issuing Intents derived from static analysis.

Some Intents do not require any additional information. Sending the `BOOT_COMPLETED` Intent unambiguously indicates that the device has finished its startup procedures. Other Intents lose meaning without additional information. For instance, consider a text message (SMS), which requires associated telephone numbers and message content. In order to handle these more complex situations, A5 uses a custom library, libIntent, to create Intents that consist of well-formed data. Continuing with the SMS example, when static analysis has indicated that an application may receive an SMS, A5 uses libIntent to send text messages via ADB to the device with random message content and random, valid 10-digit telephone numbers as the source. In A5's current implementation, libIntent handles SMS messages, power events, battery level changes, network events (delay, speed, cellular type, etc), GPS data, Voice calls, and the ability to pass through generic events in case users wish to use libIntent independently from A5.

The final method of interaction is via a software feature known as a "monkey." Android's developer software distribution contains a program named `monkey` for use in user interface testing. The `monkey` program "generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events" [4]. By using the `monkey`, A5 can automatically simulate user input.

```
 1 Intent i = new Intent();
 2 i.setAction(some.custom.intent.FOO);
 3 context.sendBroadcast(i);
 4
 5 public class IncomingReceiver extends BroadcastReceiver {
 6   public void onReceive(Context context, Intent intent) {
 7     if (intent.getAction().equals(some.custom.intent.FOO)) {
 8       //some action
 9     }
10   }
11 }
```

Figure 7: An example of a custom Intent, line 2. A broadcastReceiver as defined in lines 5-11 can receive such an Intent by observing the custom string in the Intent's action (line 7).

This rather coarse method of interacting with the application can change the state of the installed program, possibly making the instance more likely to respond to interaction. For instance, consider an application that blocks access using a modal dialog with a EULA. A `monkey` may be invoked to "click" many times, bypassing the access restriction.

3.7 Network Traffic

Any network communication destined for the Internet from an instance is routed to the Internet by A5. This is a design decision that permits malware that connects to remote servers to communicate with these servers so that the network traffic can be captured and used to inform network countermeasures. Other commercial and academic sandbox systems operate in a similar fashion. Some systems selectively route some traffic to the Internet and direct other traffic to controlled servers or honeypots [6].

For virtual instances, A5 utilizes the network capture feature of the emulator. Using the emulator feature is not only convenient to initiate, but also results in a single data file for each input sample.

For physical instances, the devices are connected to a Wi-Fi network which is monitored. Unlike the emulator feature, network capture is performed on a shared medium. To obtain device-specific data, the network capture is filtered using the device's MAC address.

Regardless of the instance type, the resulting capture file will contain a combination of malicious traffic, administrative traffic (such as ADB), and, in the case of repackaged applications, legitimate traffic. Each of which presents a unique problem with regard to the creation of network countermeasures. The administrative traffic is generally very easy to filter. For example, ADB traffic can typically be filtered based on the TCP port. Since the ADB ports vary by the instantiating of each instance, it is unlikely that a TCP port-based filter will omit any data erroneously. This simple filter can reduce the network capture substantially. For instance, filtering ADB traffic reduces the network capture for the "NotCompatible" piece of malware [26] by 65%.

Unfortunately, filtering legitimate traffic is more difficult as the characteristics of the legitimate traffic are not known a priori. Repackaging of applications in order to add malicious functionality creates additional difficulty. One might imagine learning algorithms that observe similar behavior across a family of malware. However, with repackaging, the shared behavior may be the legitimate traffic from the original application or it may indicate the same malicious software was injected into several different applications.

In the current implementation of A5, the network captures are pruned such that much of the traffic that is not beneficial to network countermeasures is filtered out. From the network captures, analysts can then

```
1 alert tcp any any -> any any (msg:”autocreated dns->host rule”; content:”
      Host|3A20|”; content:”notcompatibleapp.eu.|0D 0A|”; within:64;)
2 alert udp any any -> any 53 (msg:”autocreated dns->dns rule”; content:”
      notcompatibleapp.eu.”; nocase;)
3 alert udp any any -> any 53 (msg:”autocreated NotCompatible data decryptor”;
       content:”notcompatibleapp.eu”; nocase;)
4 alert udp any any -> any 53 (msg:”autocreated NotCompatible data decryptor”;
       content:”3na3budet9.ru”; nocase;)
5 alert tcp any any -> any 8014 (msg:”autocreated NotCompatible data decryptor
      ”; flow:established, to_server; dsize:13; content:”|04|”; depth:1;
      content:”|01 05 00 00 00 00 07 00|”;)
```

Figure 8: Candidate Snort signatures automatically created by A5's post-analysis framework following the analysis of a NotCompatible sample. Lines 1 and 2 were created by default plugins, whereas lines 3 - 5 were created by a custom plugin designed specifically for the NotCompatible malware family.

create, for instance, intrusion detection system signatures; while the process is not entirely automated, A5 facilitates this through an extensible plugin framework.

3.8 Plugin Framework

A5 further assists the analyst through a modular post-analysis framework. Plugins are Python scripts that follow a naming convention and conform to a simple API, requiring each plugin to only define a handful of functions and follow a naming convention. Each plugin is loaded dynamically by A5 during malware execution and evaluated against dynamic analysis results or submitted samples, or both. Plugins generate two types of output, signatures intended to be used in an IDS, and freeform text that is intended to provide additional context to an A5 report. An analyst can elect to deploy any suggested signatures in light of the A5 report including any information added by plugins.

The default plugins create candidate signatures blindly—that is, without having any prior knowledge of the malware. For example, malware may issue a DNS query in order to determine the IP address of a Command-and-Control server. A perfectly acceptable IDS signature may look for this particular DNS query and block the communication, thus preventing the malware from further interaction with the remote controller. On the other hand, a DNS query may be legitimate traffic issues with the purpose of determining the IP address of a server that is used to save high score data in a game. Default plugins are a simple attempt to automate the analyst's work when a sample has never before been encountered by A5. Since the default plugins are fairly generic, they may have limited value, but they do automate the signature creation process and may help prevent costly typographical mistakes in signature creation.

Unlike newly encountered malware, the plugin framework can also be employed to automate signature creation for known malware families. The framework facilitates programmatic access to the data collected during dynamic analysis and the associated Android application. In this way, malware can be identified using the submitting application, or via captured traffic, or both. Subsequently, prior knowledge can be applied to new variants of malware to automate many analysis tasks.

As we will show in 4, a plugin can be used to automate specific identification, data extraction, decryption, and the crafting of IDS rules well suited for a malware family. A5 plugins provide a general method of applying actions that may otherwise take considerable time for an analyst.

We provide an example of candidate signatures created by the post-analysis framework in Figure 8. While the plugin here creates Snort signatures, other IDS (e.g., Bro) could be supported in a similar manner.

The signatures have been automatically created from the dynamic analysis of the NotCompatible malware [26]. An analyst would still be charged with identifying that the candidate on line 1 is likely of little value, but that lines 2–5 may be useful. Lines 1 and 2 were created by default plugins that assume that DNS queries still present in the pruned network traces search for malicious domains, and thus a DNS based rule or an HTTP rule based on the DNS name may be useful. However, the NotCompatible malware doesn't use the HTTP protocol, so a signature looking for a Host header is not relevant.

The real power of the plugin framework is actually more evident in lines 3–5. Rather than a specific piece of malware, NotCompatible actually defines a family of related pieces of malware—we have observed dozens of different samples in the wild. Using a previously seen variant of NotCompatible, our plugin was able to produce a custom rule for this (unseen before) variant of NotCompatible in lines 3–5. In other words, our plugin is able to automatically generate IDS signatures for new variants of NotCompatible as long as the overall networking protocol used in the malware family does not change.

More precisely, the plugin, whose source can be found in the appendix, identifies NotCompatible malware based on the unique networking protocol employed by the malware. Once the malware is identified, the plugin can then automatically decrypt the command-and-control server data stored in the APK. This is very useful for the analyst since the dynamic analysis system is likely to only create network traffic to the initial command-and-control server, therefore never having to resort to the backup server.

4 Evaluation

Here we evaluate an implementation of A5. We evaluate our design goals with the specific implementation, measure performance of the system, and compare A5 with an existing Android runtime analysis system.

4.1 Design Goals

In section 3.1, we enumerated five criteria that our A5 must embody in order to be a successful sandbox. Here we discuss how A5 fulfills each of these metrics in turn.

Autonomy and scalability. A5's distributed and expandable architecture permit A5 to grow linearly with the growth of mobile malware samples. Additional Workers can be added to an A5 installation by adding resources to an existing node or by adding additional nodes. Further, the periodic ingestion process along with on-demand web service submission affords A5 the flexibility of interacted with a user dynamically or autonomously processing malware feeds without user interaction.

Whether samples are processed automatically or on-demand, A5 attempts to ease the burden placed on the analyst by filtering out innocuous traffic and presenting candidate signatures for the remaining network activity. For identified malware families, additional post-processing can be performed in order to further automate analysis and countermeasure creation. Ultimately, human interaction is still required for final determination of network countermeasures, but this interaction is substantially less than if the analyst had to perform any analysis manually.

Mobile-specific interaction. The methods of malware coercion certainly vary between mobile and PC-centric malware. Here A5 employs a variety of mobile-specific coercion techniques. The most straight-forward interaction is actually running the mobile application. For most applications, the interaction is similar to that of PC-malware: starting the default Android Activity. However, through two stages of static analysis, A5 determines many types of coercion that would otherwise be omitted from a dynamic analysis system. Intents determined from static analysis are then issued in an Android instance dynamically, effecting the malicious behavior.

Evasion resistance. When interacting with malware, A5 takes steps to present plausible information to the application under evaluation. For instance, text messages appear to originate from random, valid numbers.

By using plausible data and avoiding predictable data, A5 seeks to be less identifiable to malware that inspects the high level state of the device.

Knowing that virtualization detection has a mature presence in targetting other malware sandboxes, it is a foregone conclusion that Android malware will employ similar tactics. Therefore, virtual instances used in A5 employ a modified Android emulator that takes steps to hide it's virtual nature by embodying programmatic characteristics of a physical device. Even so, some types of virtualization detection are simply too difficult to implement as a modification to the standard Android emulator. A5 allows system implementers to mitigate this risk by using physical instances thus avoiding use of an emulator altogether. By using real devices, the malware actually runs on hardware making sandbox detection much more difficult for malware authors.

In another way, A5 avoids a type of evasion in that malware targeting any particular Android API version will run in A5. If A5 only supported a subset of API versions then malware might not run in A5 at all, rendering no analysis and effectively evading the entire system. By allowing device pools to cover the entire range of Android APIs, among any combination of virtual and physical instances, A5 can start a Worker for any API version required by a particular sample.

Network-level indicator collection. The collection of network indicators is critical to enabling enterprises and network providers to protect their networks, even when individual mobile devices may have little or no security-oriented software. For virtual instances, A5 makes use of a feature built-in to the Android emulator to collect network activity during dynamic analysis. For physical instances, A5 makes use of commodity network capture tools use on a shared WIFI access point. Network traffic capture on the shared network is filtered based on the known MAC addresses of the physical devices installed in the A5 instance. For either instance type, A5 creates candidate network countermeasures for never before seen samples and performs extended post-analysis on recognizable malware families leading to additional network countermeasures.

Modularity A5 is written in a very modular, object-oriented way. Interaction with dependent software (such as ADB) are written as libraries providing a re-usable interface. Use of support systems such as database and job queuing are also done in very standard ways using mature subsystems. In these ways, A5 is adaptable to existing research environments that have different workflows or special needs.

However, for the general user of A5, the more useful form of modularity come from the post-analysis plugin framework. This framework facilitates in automating tasks upon dynamic analysis results and submitted malware samples. Plugins can be used to automatically create candidate network signatures, decode sections of network traffic, decrypt malware payloads, and any number of other uses.

Plugins currently implemented in A5 focus on Snort [24] IDS format, though modifying for other formats is often merely a matter of syntax. Since any given user may prefer a particular IDS, each of which may have very custom syntax, preprocessors, configurations, etc. the default plugins are likely of limited value to any given user. However, the modularity of the plugin framework allows simple creation of new plugin such that candidate rules can be readily deployed to a particular system.

We measured A5 performance using 1260 samples from the dataset described by Zhou and Jiang [39]. The A5 system is on an 8 processor (Intel Xeon E5620 @ 2.4 GHz) Linux machine with 64GB of RAM. A5 was configured to use 5 Workers, and the arbitrary sleep time for each sample (see Figure 5) was set to 60 seconds. The entire device pool was set to be virtual instances, with two device instances for each SDK version. As shown in Table 1, the average runtime for samples was 149.1 seconds, requiring just over 50 hours of cumulative execution to run all 1260 samples [39]. By using 5 Workers, the time required to process the entire set was reduced to just 10 hours.

Stage 1 Static Analysis performed correctly for all samples. Stage 2 Static Analysis prematurely terminated on 10% of samples. In particular, the DED decompiler [11] failed to decompile 10% (131) of samples. Of the 1129 applications that successfully decompiled, Stage 2 Analysis discovered Intents in 4.5% (51) of samples (14 of 50 families).

```
1   12:14:56.918698 IP 10.0.2.15.22219 > 10.0.2.3.domain: 22666+ A?
        notcompatibleapp.eu. (37)
2   0x0000:  4500 0041 a717 4000 4011 7b83 0a00 020f
3   0x0010:  0a00 0203 56cb 0035 002d 7610 588a 0100
4   0x0020:  0001 0000 0000 0000 106e 6f74 636f 6d70
5   0x0030:  6174 6962 6c65 6170 7002 6575 0000 0100
6   0x0040:  0120
```

Figure 9: ASCII representation of network traffic captured by A5 from NotCompatible malware. NotCompatible's namesake domain is clearly present (line 1) in this DNS query from 10.0.2.15, an A5 device.

A5 Stage	**Min Time**	**Max Time**	**Average Time**
Stage 1 Static Analysis	1.9	5.6	2.6
Stage 2 Static Analysis	10.0	120.7	12.7
Dynamic Analysis	124.5	202.3	133.8
Total runtime	136.4	328.6	149.1

Table 1: A5 runtimes per sample (in seconds). Each time is was calculated by processing the entire dataset described by Zhou and Jiang [39].

We also analyzed some more recent malware using A5 and verified the results manually. For example, NotCompatible is Android malware that does not "root" the infected device, but nonetheless is able to act as a TCP proxy, tunneling malicious traffic through an infected device [26]. This malware has typical botnet like features such as the ability for a remote actor to control the software through a publicly accessible server, known as a command-and-control server. A5 was able to identify remote servers associated with NotCompatible malware. The primary remote server was immediately observable in the DNS request (Figure 9) and subsequent malware communication. Manual static analysis also revealed the remote address, but required significant time to understand the Dalvik classes and required decrypting a data file. One operation of the NotCompatible malware allows a remote server to update the remote destination address, so the next remote communication would happen with a different server. Static analysis would not be able to discern this next server address.

The NotCompatible malware also highlights the usefulness of A5's Android-specific coercion techniques. NotCompatible malware does not execute upon installation. Instead, the malware waits for certain system events, namely `BOOT_COMPLETED` or `USER_PRESENT`. In other words, NotCompatible's malicious service will only start when the device is rebooted, or when the user unlocks the device's screen lock. A5 is able to determine the necessity to mimic these events and successfully coerce the malware.

The post-analysis plugin is useful to create the candidate signatures (shown earlier in Figure 8). In this case, the plugin automatically determines that, in addition to the traffic observed to the primary server, `notcompatibleapp.eu`, other infected devices may attempt to connect to the backup domain `3na3budet9.ru`, which is encrypted and stored within the malware. This specific sample uses TCP port 8014 on both servers, which A5 was able to automatically detect, and create IDS rules, based on the generic template created from a higher-level description of the malware family.

4.2 Existing sandbox comparison

In order to highlight some of A5's advancements we can evaluate some metrics with another Android sandbox. Unfortunately the source code and design of other sandboxes are not publicly available, so for comparison, we submitted samples to Andrubis via its web interface [2]. We observe that when a sample is submitted, the estimated job completion time is always reported to be about eight minutes. If we submit samples very quickly, multiple submissions queue sequentially causing a delay before the submitted sample is processed, indicating that Andrubis does not process samples in parallel. For this reason, we purposefully submitted samples one at a time, waiting for a current sample to complete before submitting another. Even though this is a public web service, we did not observe any other user of Andrubis during out testing.

We created an Android application that extracted virtualization information mentioned in 4.1. Namely, the Android Build strings and `TelephonyManager` identifies Andrubis' runtime software as an Android emulator. From this, we conclude that Andrubis does not appear to employ physical devices nor emulator detection mitigations. We also submitted a second application that was identical to this first application except for an appended null byte in order to give the otherwise identical application a different file size and associated cryptographic hash. This was meant to ensure that submitted applications are actually executed by Andrubis and to observe similarity between two similar samples submissions. The reports were very similar even though execution time for the two samples were reported to be 227 seconds and 337 seconds respectively, so there is some variance in Andrubis' application runtime.

We then created 14 variations of the test Android application, each different only in that the settings for minimum, maximum and target SDK as configured in the `AndroidManifest.xml`. It seems that Andrubis only supports a subset of Android SDK versions supported by TaintDroid, namely 2.1 and 2.3. For example, submitting an APK with the minimum, maximum and target SDK each set to 8, denoting Android 2.2, in the AndroidManifest causes dynamic analysis to fail. Andrubis fails in this way for many combinations of SDK values causing the sample to not be analyzed at all.

Another difference is that Andrubis does not appear to use the static analysis portions to inform the dynamic analysis. This is easily evidenced by submitting an application that takes no action except upon some other, outside event, such as the receipt of a text message, or activation of the screen lock. Unfortunately, this means that for malware like NotCompatible, Andrubis will not observe any of the malicious behavior.

In practice, Andrubis often does not start processing samples immediately, in fact, only once did a sample start processing immediately. Instead, of the 15 submissions, Andrubis reported an average delay of 92 minutes (min: immediate, max: 16 hours). Overall, the execution time across our submissions averaged 253 seconds (min:227, max:337 seconds). Given that both Andrubis and A5 must incorporate certain delay in dynamic processing, the difference in average runtime is not significant.

As a final difference, A5 reports contain only meta information about the sample, but also contain IDS signature candidates. In this way A5 provides immediately actionable results. Conversely, Andrubis reports do not contain IDS signatures and may require more evaluation from an analyst, who then must manually create IDS signatures.

5 Limitations

In section 3, we detailed several components of our analysis system. Design decisions and implementation choices affect the capabilities of each of these components. Here we address the current limitations for each component of A5.

5.1 Static Analysis

In addition to general limitations of dynamic analysis systems, A5 has several additional limitations. In static analysis, A5 depends upon tools such as Androguard [1] and Soot [30]. Deficiencies in these tools may also manifest in A5.

The bytecode static analysis is limited to finding only behaviors that are statically defined and extractable from the bytecode itself. For instance, a given application component such as an Activity is started either by the Android middleware, or by another application, by sending an appropriate Intent to the application. Ideally, we would want to know all Intents receivable by the application, so we can coerce its behavior. Unfortunately, for methods which receive Intents, the method is provided the Intent via an `android.content.Intent` object, which contains a string with the name of the Intent used to trigger the target application. As the value in this string is filled in dynamically at runtime when the Intent is created, A5 is unable to statically determine the string content. In order to statically identify the Intent that triggered the activity or other application component, every Intent would need to be of a different type which is statically specified. However, this is not the case, and A5 is unable to extract or identify Intents received by receivers. Hence, the bytecode static analysis is able to extract only Intents which the application registers to receive, as the Intent action string is set statically when notifying the system of the Intents to receive.

The traversal of the CFG is intra-procedural, and the CFG analysis is flow-insensitive meaning all branches and loops are ignored. While this a priori could result in unsound analysis, this has not been an issue in the context of A5. Indeed, the two types of interaction points currently sought are method invocations and interaction points are extracted by observing the values of arguments passed to the method. Typically, the argument construction leading to the method invocation would be linear code in the same basic block without any conditional statements. In this case, flow-insensitive analysis would be sufficient for analysis. Flow-insensitive analysis may also introduce false positives. However, since in the case of additional false positives the malware coercion would only require a subset of identified interactions which would result in effective coercion. The primary issue with false positives is decreased performance due to unnecessary coercion.

In addition, by virtue of static analysis being static, any dynamically received code updates by the application cannot be analyzed. For instance, the static analysis will have no access to bytecode or Java/Dalvik classes received at runtime and subsequently executed. However, A5's dynamic-analysis phase will still be able to collect network indicators of the dynamically executed code. This type of *update attack* [39] has been detected in malware in the wild,

5.2 Dynamic Analysis

A5 exhibits well-known limitations similar to any dynamic analysis system. For example, if the malware behavior changes based on time of day, the success of A5 would depend greatly on when the sample happened to be selected by a Worker. Similarly, if malware targets a specific manufacturer other than the manufacturer employed in dynamic analysis, the malware may exhibit alternate behavior during analysis. Generally, it is just not possible to ensure that all possible functionality of a sample is explored during execution [14]. As with any sandbox system, the primary use is to quickly process volumes of malware – handling most samples correctly, not focusing on individual accuracy of a given sample.

There is some evidence that randomly issuing input to the user interface may not yield successful results [18]. Gilbert et al. conclude that random input, as is expected from an Android Monkey, may yield as low as 40% coverage of all possible UI input. On the other hand, more structured input is explored by Zheng et al. [38] which exhibits its own drawbacks, in particular the inability to handle certain logic when interacting with the UI. This particular implementation [38] has other drawbacks, but many can likely be overcome. Much as the hybrid static and dynamic analysis employed by A5, the best UI interaction is likely a combination of random and structured input. Additionally, new methods of UI interaction are increasingly available as Android's SDK evolves. Devices using API 16 or higher have a "Dump view hierarchy" that could be used to inform a UI automator. This hierarchy presents a tree of UI components that could be traversed as part of dynamic analysis. Continuing with the previous example of the EULA modal dialog, using such a UI automator could enable a dynamic system to precisely "click" an accept button instead of randomly happening to "click" the screen in a correct location.

In practice, ADB is not reliable, often not performing the desired action, yet returning a successful return code. In order to perform in lieu of these faults, A5 monitors ADB execution repeating actions until the desired effect is observed.

6 Conclusions

We have presented a system, A5, to completely automate the dynamic execution of Android malware. Our system extracts information from the malware prior to execution in order to better understand how to interact with the malware. In this way, A5 is able to better coerce malicious behavior from the malware and ultimately capture network threat indicators. These indicators can be used simply to quickly, better understand the malware, and to inform network-oriented countermeasures.

The implementation we have described uses novel methods of interacting with mobile applications, extracting interaction points during static analysis to inform the dynamic analysis process.

The distributed, parallel design of A5 allows instances to scale with the growth of mobile malware. A5 is not only highly parallel but also modular in design, allowing wholesale replacement of components in favor of newer, better performing options.

Any sandbox implementation for Android must be aware that current virtualization capabilities are incomplete, and analysis will likely soon encounter malware that employs virtualization detection techniques. A5 aims to partially address this issue both by making virtual instances more evasion-resistant and by having the flexibility to use actual physical devices during dynamic analysis.

We also presented performance measurements of a specific implementation of A5 using a public dataset of 1,260 unique Android malware samples. On modest hardware, the implementation was able to process the malware set averaging 149 seconds per sample.

A5 represents a new generation of automated analysis able to cope with large volumes of mobile malware. A5 is able to better coerce malicious behavior by interacting with mobile malware in mobile-specific ways. By focusing on network threat indicators, A5 is immediately useful to cellular providers and operators of wireless networks.

An implementation of A5 is publicly available at `http://dogo.ece.cmu.edu/a5`.

Acknowledgments

This work was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by the National Science Foundation under IGERT award DGE-0903659, and under award CCF-0424422 (TRUST).

References

[1] Androguard: Reverse engineering, malware and goodware analysis of android applications. `http://code.google.com/p/androguard/`.

[2] Andrubis. `http://anubis.iseclab.org/`.

[3] android-apktool: a tool to reverse engineer Android apk files, 2011. `https://code.google.com/p/android-apktool/`.

[4] Android developer guide 4.0 r1. `http://developer.android.com/reference/`, Oct. 2011.

[5] android4me: J2ME port of Google's Android, 2011. `https://code.google.com/p/android4me/downloads/list`.

[6] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[7] T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *MALWARE, 5th International Conference on*, pages 55–62. IEEE, 2010.

[8] E. Chin, A. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *9th Annual MobiSys*, 2011.

[9] H. Daharmdasani. Android.hehe: Malware now disconnects phone calls. http://www.fireeye.com/blog/technical/2014/01/android-hehe-malware-now-disconnects-phone-calls.html, Jan. 2014.

[10] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: an Information-Flow tracking system for realtime privacy monitoring on smartphones. In *OSDI 2010*.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX Security Symposium*, San Francisco, CA, Aug 2011.

[12] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, Jan 2009.

[13] F-Secure Labs. Mobile Threat Report Q3 2012. `http://www.f-secure.com/static/doc/labs_global/Research/MobileThreatReportQ32012.pdf`, Nov. 2012.

[14] D. Farmer and W. Venema. *Forensic discovery*, volume 18. Addison-Wesley, 2005.

[15] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638, 2011.

[16] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM SPSM*, pages 3–14, 2011.

[17] Gartner. Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent. `http://www.gartner.com/it/page.jsp?id=2237315`, Nov. 2012.

[18] P. Gilbert, B.-G. Chun, L. P. Cox, and J. Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011.

[19] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286, 2004.

[20] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM Computer Communication Review*, 34(1):51–56, 2004.

[21] P. Lantz, A. Desnos, and K. Yang. DroidBox: Android application sandbox. `http://code.google.com/p/droidbox/`, 2012.

[22] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *NDSS*, Feb. 2013.

[23] Lookout Mobile Security. State of mobile security 2012. `https://www.lookout.com/resources/reports/state-of-mobile-security-2012`, 2012.

[24] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA, pp229–238*, 1999.

[25] C. Rossow, C. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. Freiling, and N. Pohlmann. Sandnet: Network traffic analysis of malicious software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pages 78–88. ACM, 2011.

[26] F. Ruiz. 'Android/NotCompatible' Looks Like Piece of PC Botnet. `http://blogs.mcafee.com/mcafee-labs/androidnotcompatible-looks-like-piece-of-pc-botnet`, May 2012.

[27] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 2010.

[28] J. Stewart. Truman-the reusable unknown malware analysis net. *http://www. secureworks. com/research/tools/truma. html*, 2006.

[29] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerabilty discovery*. Addison-Wesley Professional, 2007.

[30] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[31] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations. *Sable Technical Report 1998-4, Sable Research Group, McGill University*, 1998.

[32] T. Vidas and N. Christin. Sweetening android lemon markets: Measuring and curbing malware in application marketplaces. In *Proceedings of the third ACM CODASPY*, 2013.

[33] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. *Proc. CCS*, 1, 2014.

[34] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *IEEE W2SP*, volume 2, 2011.

[35] VirusTotal. VirusTotal - Free Online Virus, Malware and URL Scanner. https://www.virustotal.com/.

[36] Wicherski, G. MWcollect. http://www.mwcollect.org, Feb 2011.

[37] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2):32–39, 2007.

[38] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM SPSM*, 2012.

[39] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy, Symposium on*, 2012.

[40] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

A NotCompatible A5 plugin

```
#filenames for plugins must start with the string "plugin_" and end in ".py"

#plugins always return a tuple (pluginName,listOfCountermeasures,listOfComments)
#where the first value is a string and the second two are each a python List
#plugins should have no printed output

#pluginName is a required variable for plugins
#this is simply a name for the plugin that is used in logging and stdout

pluginName = "NotCompatible data decyptor"

#if true, the plugin will be used, if false it will not
enable = True

#type is a required variable for plugins
#type is simply a string that is used to group plugins by category,
#used for unit testing, in production the type often doesn’t matter
type = "known"

#logger is optional, if the plugin requests a logger like this, logging entries will end up in the sh
#import logging
#logger = logging.getLogger(__name__)

class PluginClass:
    msg = "autocreated NotCompatible data decryptor"

    def get_dns_rule(self,domain):
      return (’alert udp any any -> any 53 (msg:"%s"; content:"%s"; nocase;)’
             % (self.msg,domain))

    def get_notc_rule(self,port):
      return (’alert tcp any any -> any %s (msg:"%s"; flow:established,
             to_server; dsize:13; content:"|04|"; depth:1;
             content:"|01 05 00 00 00 00 07 00|";)’ % (port,self.msg))

    def run(self,pcap,apk):
        ruleList = list()
        commentList = list()

        if pcap is None or apk is None:
          commentList.append("this plugin requires a pcap file and an apk to work")
          logger.error("plugin requires a pcap and an apk...but didn’t get em")
          return (pluginName,None,commentList)

        try:
          from scapy.all import PcapReader,hexdump,ls
          import sys

          my_reader = PcapReader(pcap)
          if(self.findNotCompatiblePhoneHome(my_reader)):
            pt = self.decryptNotCompatibleData(apk,ruleList,commentList)
            (primary,secondary,pport,sport) = pt.split(’|’)
            commentList.append("new notcompatible sockets: %s:%s , %s:%s" % (primary,pport,
                                 secondary,sport))
            ruleList.append(self.get_dns_rule(primary))
            ruleList.append(self.get_dns_rule(secondary))
            ruleList.append(self.get_notc_rule(pport))
            ruleList.append(self.get_notc_rule(sport))
        except IOError:
          logger.error("Failed reading pcap")
          return (pluginName, None, None)

        return (pluginName, ruleList, commentList)
    def decrypt(self,passkey,param):
        from Crypto.Hash import SHA256
        from Crypto.Cipher import AES
        keyhash = SHA256.new(passkey)
        key = keyhash.digest()

        cipher = AES.new(str(key), AES.MODE_ECB, "ignored")

        buf = buffer(param,0,len(param))
        return cipher.decrypt(buf)

    def decryptNotCompatibleData(self,apk,rules,comments):
        from scapy.all import hexdump

        #directly from the zip (relies upon working zip imlementation in python!)
        import zipfile
        file = zipfile.ZipFile(apk,"r")
        data = file.read("res/raw/data")

        key = "ZTY4MGE5YQo"
        comments.append("key: %s" % key)
        plaintext = self.decrypt(key, data)
        comments.append("decrypted: %s" % plaintext)

        return plaintext

    def findNotCompatiblePhoneHome(self,reader):
      from scapy.all import TCP,Raw,hexdump
      for pkt in reader:
        if pkt.haslayer(TCP):
          if pkt.haslayer(Raw):
            data = pkt.getlayer(Raw).load
            if len(data) == 13:
              if data[0] == ’\x04’:
                initialcall = bytearray.fromhex("01 05 00 00 00")
                if data[3:8] == initialcall:
                  return True
      return False
```