VirtualSwindle: An Automated Attack Against In-App Billing on Android

Collin Mulliner
Northeastern University
Boston, MA
crm@ccs.neu.edu

William Robertson
Northeastern University
Boston, MA
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
Boston, MA
ek@ccs.neu.edu

ABSTRACT

Since its introduction, Android's in-app billing service has quickly gained popularity. The in-app billing service allows users to pay for options, services, subscriptions, and virtual goods from within mobile apps themselves. In-app billing is attractive for developers because it is easy to integrate, and has the advantage that the developer does not need to be concerned with managing financial transactions. In this paper, we present the first fully-automated attack against the in-app billing service on Android. Using our prototype, we conducted a robustness study against our attack, analyzing 85 of the most popular Android apps that make use of in-app billing. We found that 60% of these apps were easily and automatically crackable. We were able to bypass highly popular and prominent games such as Angry Birds and Temple Run, each of which have millions of users. Based on our study, we developed a defensive technique that specifically counters automated attacks against in-app billing. Our technique is lightweight and can be easily added to existing applications.

Categories and Subject Descriptors

K.6.5 [**Software**]: Security and Protection

General Terms: security

Keywords: Mobile Application; App Protection; Payment; Smartphone Security

1. INTRODUCTION

Since 2008, centralized mobile application markets for smartphones such as Apple's App Store and Google's Play Store have radically changed the way applications are installed on mobile devices. Users can now easily search for, read reviews on, and install mobile apps from central repositories that come pre-configured on the device. Furthermore, such centralized market places also provide the advantage that updating installed applications is simpler and more efficient.

One recent development in the competition for providing new app market-based services and, hence, creating new revenue streams for app market providers is *in-app purchasing*. This feature first appeared in Apple's iOS 4 in 2010, and allows users to pay for options, services, subscriptions, and virtual goods from within apps themselves. In March 2011, Google followed suit, and provided a service they term in-app billing [12].

The idea behind selling commercial services in the apps themselves is to allow users to first try out, use, and test an application before they are offered the option of buying additional services. For example, the first three levels of a game might be offered for free, but the user might be asked to pay for the ability to activate and access subsequent levels in the game. Similarly, some games allow users to play for free, but offer to improve the gaming experience by selling game-related tools to the player. For instance, in an aerial dogfighting game, a fighter plane that is higher performance might only be available to the player through an in-app purchase. Of course, in-app purchasing is not only useful for game developers. For example, ComiXology's Comics is a free app, but the comics themselves cost money and must be acquired through in-app purchases.

In Android's in-app billing, any developer can integrate the service in their application without needing a special contract with Google. The only requirements are the possession of *a*) a Google play developer account, and *b*) a Google Wallet merchant account. Hence, it is very easy for any developer to integrate in-app billing functionality in her application, and she is not unnecessarily burdened with having to go through a tedious registration process. For Google, making in-app billing accessible and easy to use is important because it receives a 30% sales commission on each in-app purchase. The advantage for the developer, on the other hand, is that she does not need to be concerned with credit card transactions and billing cycles. Furthermore, the in-app billing service offered by Google provides libraries that are straightforward to use, and that can be integrated into existing apps with ease.

One important question with respect to in-app billing is how secure these operations are. In a detailed document, Google discusses the security and design of in-app billing, and provides best-practice recommendations and guidelines to app developers who are planning to use the in-app billing service [13] on Android. One of the recommendations is that, if practical, the developer should perform signature verification to authorize Google-supported purchases on a remote server, and not on the device itself. The explanation provided is that implementing purchase verification on a remote server makes it difficult for attackers to break the verification process by analyzing the application code. Another recommendation is that the developer should obfuscate in-app billing code using a tool such as ProGuard so that it is difficult for an attacker to reverse engineer security protocols and other application components. Furthermore, developers are encouraged to perform method inlining, construct strings on the fly instead of defining them as constants, and use Java reflection to call methods as further obfuscation techniques.

ASIA CCS'14, June 4–6, 2014, Kyoto, Japan.

http://dx.doi.org/10.1145/2590296.2590335.

The key insight behind these obfuscation recommendations is that using these techniques will help minimize attacks that give users access to protected virtual goods while bypassing the in-app billing implementation.

Note that the discussion in the in-app billing security and design recommendations provided to app developers assumes that the majority of anticipated attacks will directly target application code through reverse engineering and manual analysis. This is a time-consuming and tedious task if the code is obfuscated. In fact, this assumption also seems to be popular among some developers. For instance, the discussion attached to Google's official recommendations document contains claims that it is not important to worry about the small number of people who have the time and knowledge to hack one instance of the app [13].

In this paper, we present the first fully-automated attack against the in-app billing service on Android. Instead of taking the manual approach of trying to reverse engineer, decompile, and patch individual apps that use in-app billing services, our generic attack uses a dynamic Dalvik instrumentation approach we have developed to inject arbitrary code into a running process. Hence, we are able to modify the behavior of any Dalvik-based Android app at runtime. Our attack code is launched as a simple app named VirtualSwindle. Our attack app runs in the background and, when invoked, automatically attacks every app using in-app billing that relies upon on-device signature verification, allowing the attacker to access digital content and services without paying for them. Hence, the bar for gaining illicit access to paid services in Android apps is significantly lowered.

Using our prototype, we conducted a robustness study against our attack, and analyzed 85 popular apps that made use of in-app billing. Our findings show that 60% of the apps we analyzed employed on-device signature verification, and were therefore easily crackable. Only 36% of the applications that we tested took in-app billing security seriously by performing the signature checks on a remote server as recommended in the in-app billing developer guidelines. Note that our paper *does not show a weakness* in Google's in-app billing architecture. Rather, it shows that many app developers *do not seem to be aware* of how easily the in-app billing functionality in their applications could potentially be bypassed if they do not rigorously follow Google's guidelines.

From an economic perspective, the amount an attacker could steal in pirated in-app items can easily add up to several hundred dollars, equivalent to the cost of a new Android phone. We found that items offered through in-app billing cost between $1 and $99 (see Figure 8), while a new Android phone (e.g., Nexus 5) costs around $399. Hence, given an average observed item cost of $20 which we draw from our empirical evaluation, an attacker need only steal 20 items to offset the cost of a new phone. In practice, attackers could quite easily steal significantly larger sums.

This paper makes the following contributions.

- We present and describe a novel dynamic Dalvik instrumentation approach. Our approach is able to instrument Dalvik applications by intercepting and mapping Dalvik methods to equivalent, attacker-provided native methods. Furthermore, our tool is able to load and inject arbitrary Dalvik code into a running process. Hence, we are able to change the behavior of any Dalvik-based Android process at runtime without tampering with application code signatures.

- We present a novel attack against Google Play's in-app billing service on the Android platform. Our attack is generic and, once launched, automatically attacks every application that uses in-app billing on the targeted device that relies upon on-device signature verification. As our attack is dynamic, any static protection mechanism that uses code obfuscation (e.g., reflection or variable renaming) is ineffective and easily evaded.

- We counter the popular developer folk wisdom that following the simple obfuscation guidelines in the in-app billing documentation will significantly deter attackers from compromising their apps. We performed empirical experiments, testing our attack on 85 popular apps that make use of in-app billing, and found that 60% of the apps we analyzed were automatically and easily crackable using our attack prototype. Among the crackable apps were well-known, extremely popular apps such as Angry Birds and Temple Run. We present detailed reports on the in-app billing defenses implemented by these 85 popular Android apps.

- We developed a lightweight countermeasure that prevents fully automated attacks like ours. Our countermeasure is based on replicating and obfuscating code specific to the billing process. The aim of the technique is to force attackers to perform manual analysis of applications in order to crack them. Our method can be integrated into existing applications with low effort.

- We have posted an anonymous video of our attack demo on YouTube at `http://www.youtube.com/watch?v=Jx5GGINNGoc`. We hope that the video will be educational for developers, and will motivate them to better secure their applications.

The rest of this paper is organized as follows. The next section gives background information on in-app billing. Section 3 discusses the threat model we assume in this work. Section 4 describes our automated attack against in-app billing. Section 5 presents our evaluation and experiments. Section 6 presents and discusses our lightweight countermeasure. Section 7 discusses related work, and Section 8 concludes the paper.

2. BACKGROUND: IN-APP BILLING

In this section, we first provide background information on in-app billing for the Android platform before presenting details of our attack.

2.1 The In-App Billing Architecture

The architecture of the Android in-app billing service consists of four components; three are mandatory and one is optional. The first component is the Android Play Store application, which is used to download applications from the Play Store. The Play Store app exposes the `MarketBillingService` interface that is used by applications to interact with the billing system. The second component is the back-end Play Store server that is hosted by Google. The aim of the Play Store server is to perform the actual financial transaction requested by remote apps. The third component in the architecture is the application on the device that uses the in-app billing service. The application communicates with Google's Play Store by using the `MarketBillingService` API. The fourth optional component is a server-side service that can be deployed by the application developer. The application developer can use this optional service to provide services for users of his apps such as content distribution or data storage. In the context of in-app billing, the developer can also use it to authorize in-app billing transactions. Figure 1 depicts the architecture of the in-app billing service.

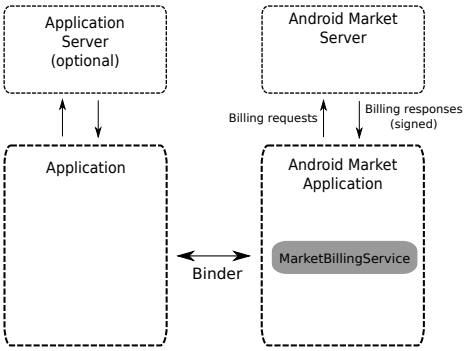

Figure 1: The Android in-app billing architecture [12].

2.2 In-App Billing: Developer's Perspective

Developers that wish to include in-app billing services in their apps need to perform three basic steps.

1. **Add in-app billing code to their application**. This is the code that talks to the Play Store app on the Android device that provides the in-app billing service.
2. **Register items in the Android Play Store**. These items are the digital services that users can purchase inside the application. For example, an artifact could be the ammunition that is needed for a powerful weapon that the user has picked up in a game. Note that every item that can be purchased has to be registered on the Play Store back-end server so that it can be sold as a digital service through in-app billing.
3. **Generate a public key pair**. For in-app billing to be activated, a private key needs to be uploaded by the developer to the Play Store server. It is then used to sign each in-app purchase. The public key is embedded into the application by the developer, and is used to verify the signature of the purchase data that is provided by the Google in-app billing service.

Once these steps are completed by the developer, in-app billing can be used in the app. Note that applications do not need to go through any testing process, and they do not need any special permissions from Google to be able to use in-app billing. Clearly, Google, as the in-app billing provider, is interested in making this process as easy and as low-overhead as possible for developers.

2.3 Purchasing an Item with In-app Billing

An application that has in-app billing functionality uses two interfaces to the Android Play Store app. The main interface is the `sendBillingRequest` method call. This method takes a `Bundle` (i.e., an object that holds key-value mappings) as an argument, and returns another Bundle as the result. The second interface is a broadcast receiver. The broadcast receiver is used to receive `Intents`, which are abstract descriptions of an operation to be performed, from the Android Play Store app. Note that the in-app billing process is highly asynchronous and, therefore, all state changes are signaled via `Intents`.

To purchase an item, three steps have to be taken. First, the application issues the `REQUEST_PURCHASE` command using the `sendBillingRequest` method. The Play Store replies with a Bundle that contains a `REQUEST_ID` and a `PendingIntent`, which is a special `Intent` that is used to display the payment screen. Figure 2 shows a screenshot of the payment screen from the popular game Temple Run.

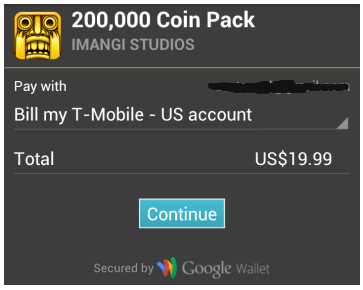

Figure 2: Payment screen from the game Temple Run.

Once the user completes the payment process, the Play Store application sends an `IN_APP_NOTIFY Intent` to the application, and indicates that the payment has been successfully received. The application responds with the `GET_PURCHASE_INFORMATION` command. The Play Store application then follows with a `PURCHASE_STATE_CHANGED Intent`. This `Intent` contains detailed information about the completed in-app purchase. That is, it contains purchase information that has been generated by the Play Store server, and that has been signed with the developer's private key.

After the application receives the signed purchase information, it checks the signature of the data sent by the Play Store app, and confirms the sale with a `CONFIRM_NOTIFICATIONS` command. At this point, the purchase is now complete. Figure 3 shows the command and Intent exchange between an application that uses in-app billing and the Android Play Store app.

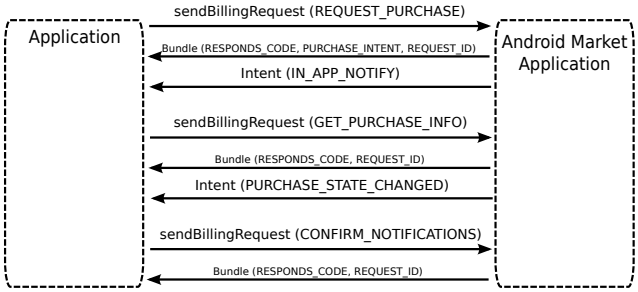

Figure 3: Method calls and `Intents` for a payment [12].

3. THREAT MODEL

In this section, we describe the threat model that we assume for our attack against in-app billing. We first discuss the motivation of the attacker, and then describe the prerequisites for the attack.

3.1 The Attacker

The attacker is the owner of an Android device, and is interested in using in-app services without paying for them. Hence, by launching an attack, the attacker aims to automatically subvert the in-app billing process. The defenders are *a*) the in-app billing service on the device that is responsible for in-app purchases and that receives a commission on each sale, and *b*) the application developer who is interested in making her application as difficult to crack as possible (e.g., but following security recommendations in the in-app billing documentation [13]).

3.2 The Prerequisites

For our attack to succeed, there are two prerequisites that need to be fulfilled. The first prerequisite is that the attacker has full con-

trol over her own device, and that she can gain root access. We believe that this assumption is realistic, as rooting an Android device is straightforward, and a wealth of documentation on the topic is available. Furthermore, popular tools such as SuperOneClick [6] allow technically unsophisticated users to automatically root Android devices. Note that unlike Apple's iOS, Google and most Android device manufacturers tolerate users acquiring root access to their devices. For example, HTC has an official site to unlock the bootloader of their devices [14]. An unlocked boot loader allows one to easily root a phone by installing a modified firmware such as CyanogenMod [7].

Rooting a device normally consists of installing an `su` (superuser) binary with setuid root permissions in `/system/xbin`. This process is as simple as downloading a ZIP archive to the `/sdcard` path of an Android device, and installing its contents using the Android recovery boot loader that is present on all devices.

Also, note that rooting an Android device does not mean that the device's security guarantees are necessarily void. For example, SuperSU [4] has a GUI application that allows the user to selectively give super-user privileges to individual applications. Thus, the user keeps control over which applications have root privileges on her device.

The second prerequisite for a successful automated attack is that the application developer performs signature verification on the device using the functionality provided by Google's in-app billing service. As our experiments in Section 5 demonstrate, this assumption is realistic based on empirical measurements. That is, the majority of the apps we analyzed – including highly popular games such as Angry Birds – performed on-device signature verification, relied on existing libraries for this task, and were therefore vulnerable.

4. ATTACKING IN-APP BILLING

Our attack against in-app billing is straightforward, and works by emulating and subverting the `MarketBillingService` inside the Android Play Store application on the device itself. The `MarketBillingService` has one function, `sendBillingRequest`, that is exposed to applications as an Inter-Process Communication (IPC) endpoint. As discussed in Section 2, `sendBillingRequest` takes a `Bundle` as an argument, and returns another `Bundle` as a result.

In the attack, we dynamically subvert and modify the Play Store application on the device. In other words, at runtime, we replace the `sendBillingRequest` method with our own code. Our version of `sendBillingRequest` then generates the answer `Bundles` and `Intent` that make the calling app believe that the purchase was successful.

For our attack to be successful, we only have to overcome a single security check located inside the in-app billing-enabled victim app. In particular, this check is the signature verification of the purchase data returned to the application by the Android Play Store. To overcome the signature check, we replace the standard Dalvik library method `java.security.Signature.verify` with our own method that always returns true, indicating success. Thus, we can disable signature verification with ease if the programmer has chosen to do on-device verification, and relies on the provided functionality. Note that we replace the verification function globally on the device. Hence, we do not need to attack vulnerable applications individually. In order to not break signature verification for the rest of the device we only return true if the input is our fake signature.

To be able to launch a dynamic attack against the in-app billing service, we had to develop novel tools for the dynamic instrumentation of Dalvik binaries. In this section, we describe the implementation of our attack and the tools we developed in detail.

4.1 Dynamic Dalvik Instrumentation

As a part of our attack, we created a dynamic Dalvik instrumentation approach, and implemented a library called *libddi*. Our library consists of several components, and allows us to intercept and map any Dalvik method to an alternative native function implementation provided by the attacker. Furthermore, we are also able to inject additional, arbitrary Dalvik code into a running process, and perform full dynamic instrumentation of any process that is running on the Android platform.

The Dalvik Virtual Machine on Android devices supports the invocation of native code that is loaded from shared libraries using the Java Native Interfaces (JNI) mechanism. The basic idea behind our instrumentation approach is to abuse the JNI layer of Dalvik, and to modify any interpreted Dalvik method and to replace it with a corresponding native variant that is provided by the attacker.

In the following, we describe the main features of our instrumentation tool in more detail, and discuss some of the technical challenges we faced.

4.1.1 Library Injection

Our instrumentation tool is bootstrapped through a well-known library injection technique (i.e., [5]) that we had to adjust for the ARM platform. This technique is based on executing a few instructions on the stack that invoke *dlopen(2)*, and that load the shared library into the process. The code is executed once the dynamic linker loads the shared library, and executes the library's *init* function.

4.1.2 Redirecting Dalvik Methods to Native Code

Just like the Java VM, the Dalvik VM supports calling native code loaded from shared libraries using the JNI mechanism. The key insight behind our instrumentation approach is that Dalvik calls can be replaced with native calls. In fact, this is functionality that has to exist in the VM as a feature to be able to implement and provide JNI functionality.

A method in the Dalvik VM is represented by a *method struct*. The method struct contains several fields that are related to the actual code being executed. The `insns` field points to the Dalvik bytecode if this is a Dalvik function, and the `nativeFunc` field points to a JNI helper function if the method is native; in this case, `insns` will point to the actual native code. Furthermore, the `registersSize` and `jniArgInfo` fields contain information related to method parameters, and the `accessFlags` field holds information concerning method visibility (i.e., public vs. private) and whether the method is implemented using native or managed code.

To modify an existing method in Dalvik and map it to a different native call, all of these fields must be adjusted accordingly. For example, the `jniArgInfo` field has to be set so that the JNI helper will scan the method signature to determine how the arguments are passed, and the `registersSize` field has to be adjusted with the number and type of arguments. Furthermore, the `accessFlags` have to be modified to indicate that the method is native. Finally, the actual function pointer has to be set to the field `insns`, and the field `nativeFunc` has to be pointed at the JNI helper function. The last step is performed using the `dvmUseJNIBridge` function in the Android `libdvm` library.

Figure 4 depicts an example of how the existing `verify` method of the `java.security.Signature` class can be redirected to an arbitrary native implementation. In the first step, a handle to the class `java.security.Signature` has to be obtained using the Android `dvmFindLoadedClass` function. In the second step, a method handle has to be acquired for the method in question. This is done by invoking the libdvm function `dvmFindVirtualMethod-`

```
cls = dvmFindLoadedClass("Ljava/security/Signature;");
Method *m = dvmFindVirtualMethodHierByDescriptor(
  cls, "verify", "([B)Z");
m->registersSize = 2;
m->jniArgInfo = 0x40000000;
m->accessFlags = m->accessFlags | 0x0100;
dvmUseJNIBridge(m, verify);
```

Figure 4: An example of redirecting a Dalvik method to native code. Here, the method `boolean java.security.Signature.verify(byte[])` is hijacked.

```
int verify(JNIEnv* env, jobject obj, jobject bytearray)
{
  return 1;
}
```

Figure 5: Native code to replace `java.security.Signature.verify(byte[])` with a function that always returns true.

`HierByDescriptor` and passing the method's name and signature. As described above, the method meta data is adjusted and redirected to the arbitrary, native implementation of `verify`.

Figure 5 shows the replacement function for the method `java.security.Signature.verify`, which follows the JNI call standard. The first argument is the JNI environment. This is followed by the instance object and the actual method parameters – in this case, a byte array.

4.2 Loading Arbitrary Dalvik Classes into an Existing Process

The ability to redirect any existing Dalvik method and map it to an arbitrary native call is useful and powerful in terms of dynamic instrumentation. However, our instrumentation approach also allows us to load additional, arbitrary Dalvik classes into an existing process. In other words, we are able to write instrumentation and attack code in Java. Clearly, being able to write attack code in Java has the advantage that it is easier to build new functionality as it makes it straightforward to interact with existing Dalvik libraries. In comparison, developing code in C/C++ has a much higher overhead.

To load classes into a running process, the Dalvik executable (known as a DEX file) has to be loaded from disk. This is done using the `openDexFile` function, provided by Android's libdvm, through our libddi instrumentation library. Second, we have to define the class and tell the class loader which class to load from the DEX file. This is done by calling the libdvm function `defineClass`. After the class has been loaded, it can be used just like any other class in the VM. One can create new instances, dispatch methods to those instances, and also call static methods.

Figure 6 depicts our instrumentation library libddi in operation. After the library is injected into a running process, it intercepts and patches methods that the attacker wishes to instrument, and then loads attacker-provided Dalvik classes into the target process.

4.3 Calling Patched Methods

Once a Dalvik method has been intercepted and redirected to a native method provided by the attacker, there exists no direct path to call the original method. That is, every time the method is looked up through standard Android functions such as `dvmFindVirtualMethodHierByDescriptor`, the method structure that is returned

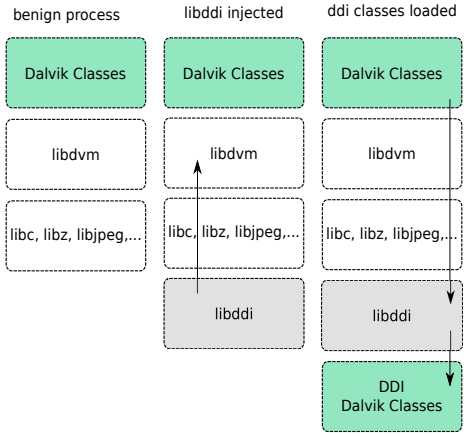

Figure 6: The DDI instrumentation library in operation. The benign process consists of native libraries, the Dalvik VM, and the Dalvik classes. In the first step, our libddi library is injected into the process. Then, libddi uses the Dalvik VM to change the Dalvik methods to point to native code. Once the native code is executed, arbitrary Dalvik classes can be loaded and executed.

points to our alternative native method implementation. The only way to call the original, replaced Dalvik code is to patch the internal method structure to make it point back to the original implementation.

4.4 Obtaining a Context

Unfortunately, interacting with the Android system from the instrumented code is not always straightforward. In many instances, an `android.content.Context` class is needed. A `Context` object allows the sending of `Intents` using the `sendBroadcast` method. Normally, a developer does not need to be concerned with obtaining a handle on the `Context` object because every Android application is a subclass of `android.content.Context`. However, if additional attacker-provided classes are loaded into a running process, the loaded classes do not have a reference to the existing `Context` instance, and hence cannot interact with the Android framework.

We had to design two techniques for obtaining a reference to the current application's `Context` object. These techniques work well in practice, and allow us to launch successful attacks against the in-app billing infrastructure on the device. In the following, we describe the two techniques we use.

4.4.1 Using the ActivityThread

If the injected code is running inside an application that is currently in the foreground, we use a static method belonging to the `ActivityThread` class to obtain a `Context` reference by calling the method `currentApplication`. The `ActivityThread` class manages the main thread of an Android application process. This technique is generic, and works in most situations. Furthermore, it can be implemented purely in Java.

4.4.2 Scraping the Instance Object

Unfortunately, in some cases the application does not run in the foreground. Hence, the `ActivityThread` approach does not work.

If the instrumented method is not a static method, the method call always has the instance object as the first argument passed to the method. We now have multiple options for acquiring a `Context` reference depending on what kind of class we are dealing with.

a) If the method is part of a class or a subclass of `Context`, then the instance object provides a valid `Context`. Well-known subclasses of `Context` are the three cornerstones of the Android application framework, namely `Service`, `Activity`, and `Application`.

b) If the class is an inner-class of any one of those classes, we simply need to access the enclosing class. The Java compiler generates a synthetic member field for such purposes named `this$0` that allows us to obtain a `Context` reference.

c) If the class has any member variable that stores a reference to a `Context`, this variable can be accessed to obtain a handle to that `Context`. In fact, this is frequently used by application developers as they also need an easy method to acquire a `Context` object to be able to interact with the Android framework.

All of the three above methods can be implemented using reflection using Java or C/C++. However, if the `Context` is acquired via C/C++, it has to be transferred to the Java world through the JNI interface.

4.5 Attacking In-App Billing

The implementation of the automated attack against in-app billing is based on the dynamic Dalvik instrumentation approach we discussed in Section 4.1. That is, we use libddi to subvert in-app billing.

In the first step, the `com.android.vending` process is hijacked by injecting libddi into the process, where `com.android.vending` is the Android process that is responsible for authorizing in-app purchases. To interact with the application process and the Dalvik VM, we need our injected code to execute in an application thread. Therefore, we hijack a common, frequently called function. We determined that `epoll_wait`, a low-level function that waits for events on a set of file descriptors, is a suitable choice since it is called frequently by the Android framework, and thus by every application. Once our version of `epoll_wait` is called, the in-app billing method `sendBillingRequest` is redirected to our own native implementation.

In the second step, whenever `sendBillingRequest` is invoked, our attack code is executed. The attack code then loads Dalvik classes that we have written into the vending process. The ability to use Dalvik classes in the attack is important because we need to be able to communicate with in-app billing victims using the standard Android communication mechanisms (e.g., `Intents`).

Our Dalvik attack engine is responsible for emulating the Android billing process, and has four basic components. A handler component first parses the request received via `sendBillingRequest` from the in-app billing victim application. Here, we extract application-specific values such as the purchased item names. A second component then generates the appropriate answers based on the extracted request data. A third component is responsible for sending back answers to the victim app as `Bundles` and `Intents`. Finally, a fourth component provides a state tracking mechanism that is used by our native attack code to query the state of the Dalvik component of the attack.

Patching `java.security.Signature.verify` is done by loading a small native library into the *zygote process*. Note that on Android systems, this process is automatically mapped into all other Dalvik processes. Hence, using this approach, our attack code only needs to be injected once into the zygote for it to be automatically propagated to every process running on the system.

4.6 Launching the Attack

The attack against in-app billing is executed as soon as the victim application sends a `REQUEST_PURCHASE` request through the `sendBillingRequest` call to the Play Store application. Our native, subverted implementation of `sendBillingRequest` operates as a central dispatcher. It receives requests from the victim applications as `Bundles`, and forwards these `Bundles` to our request handler in the attack engine. The attack consists of four main steps.

4.6.1 *Interception of* `REQUEST_PURCHASE`

Once the purchase request is intercepted, the corresponding `Bundle` is parsed and relevant data such as the package name and the item ID are extracted. `REQUEST_PURCHASE` is a special request in the context of our attack in the sense that it indicates a valid and legitimate purchase request from the victim app. Hence, the result that is returned to the victim has to make sense. Therefore, we pass this request and its parameters to the original, legitimate implementation of `sendBillingRequest`. We then take the answer generated by the Play Store app and pass it back to the victim app. The Play Store then builds a `PendingIntent` as shown in the example in Figure 2.

4.6.2 *Sending* `IN_APP_NOTIFY`

This `Intent` is be sent to indicate that the purchase was carried out by the user. The `Intent` needs to be delayed until `sendBillingRequest` has returned and the victim application has a defined state. Therefore, we create a thread and delay it for five seconds before sending the `Intent` to the victim application. We chose five seconds as the delay time as it worked well in our experiments.

4.6.3 *Handling* `GET_PURCHASE_INFORMATION`

The handling of this request is an important part of the attack. Here, we generate the JSON object that contains all the purchase information such as the product ID, the state of the purchase, and the order ID.

Under normal conditions, this JSON object would be generated and signed by the remote Play Store server. However, since we are not really performing a purchase and are only making the victim app believe that the purchase has been successful, we need to generate this JSON object ourselves. For the attack to succeed, we do not care about having a valid signature and, hence, fill the signature field with a fake signature that we Base64 encode. This is done to ensure that the field passes the Base64 decoding check in the victim app.

The two fields `inapp_signed_data` and `inapp_signature` are sent via a `PURCHASE_STATE_CHANGED Intent` back to the victim application. Figure 7 shows a real-world example of data that is sent in the `inapp_signed_data` field for the popular game Temple Run.

4.6.4 *Confirming the purchase*

If satisfied with the signature verification step, the victim app can now send a `CONFIRM_NOTIFICATIONS Intent` through the `sendBillingRequest` method. We can then detect that our attack has succeeded. Note that as we intercept and subvert the on-device signature verification libraries, any app that relies on this functionality will be automatically cracked.

4.7 Dealing with the Play Store User Interface

Our attack against in-app billing works completely automatically, without any user interaction, immediately after the `REQUEST_PURCHASE Intent` is received by our subverted version of `sendBillingRequest`. Under normal conditions, whenever an in-app purchase succeeds, the payment dialog shown by the Play Store app

```
{ "nonce": -9149971711919728712,
  "orders":
  [{
    "notificationId": "1",
    "packageName": "com.imangi.templerun",
    "orderId": 6120396686557988119,
    "purchaseState": 0,
    "purchaseTime": 0,
    "productId": "com.imangi.templerun.iap.coinpack.d"
  }]
}
```

Figure 7: Automatically-generated purchase data for the game Temple Run.

is closed as soon as the payment process is completed. However, since our attack does not actually complete the payment process in the Play Store application, pressing the back button in the Play Store app to go to a different view would normally cancel the purchase. Hence, a `CANCEL` message would be sent to the victim app. In our experiments, we observed that some apps will cancel an in-app purchase if they receive a `CANCEL` message from the Play Store after a transaction. To mitigate this problem, we also patched the `sendResponseCode` method in the Play Store application to suppress the `CANCEL` message from being sent when the payment dialog is closed via the back button.

4.8 Cheating with In-Game Currency

We adapted our attack for all games that have in-game currency. In such apps, a currency is defined in the game that allow users to save and purchase in-game items. For example, a magic sword might cost the player 100 gold coins, and these gold coins might be purchased using traditional currency through in-app billing. With this simple extension, we can create automated loops to continuously buy currency for cheating purposes, or for jumping ahead in the game's high-score rankings.

This extension requires sending the `IN_APP_NOTIFY Intent` and restarting the in-app purchase process as soon as the last step of payment is completed. Hence, an in-app item can be bought hundreds of times without having to manually interact with the user interface for each purchase.

4.9 Making the Attack Easy to Use

To make the attack prototype easy to launch and use, we created an Android app called VirtualSwindle that an attacker can launch with the simple touch of a finger. The app runs in the background, and automatically cracks every in-app billing app that relies upon on-device signature verification as discussed in the previous sections. The reader is referred to an anonymous YouTube video located at `http://www.youtube.com/watch`?v=Jx5GGINNGoc for a demonstration of the attack in action.

5. EVALUATION

The aim of our evaluation is to determine how effective our automated attack would be against popular apps that use in-app billing, and to gain insights into how security-aware app developers are today. We tested VirtualSwindle on 85 apps from the Google Play Store. Furthermore, we manually analyzed and reverse engineered these apps to determine what kinds of security precautions, if any, they were taking.

In the following, we describe how we selected the in-app billing apps we tested for robustness, how we executed the tests, and how we performed our security analyses.

5.1 Ethical Considerations

Checking popular applications to see if their in-app functionality can be bypassed could be considered an ethically sensitive issue. Clearly, one question that arises is if it is ethically acceptable and justifiable to test an attack tool on real applications. We believe that realistic experiments are the only way to reliably estimate success rates of attacks in the real world. Unfortunately, criminals do not have any second thoughts about discovering vulnerabilities in the wild, or launching attacks. We note that VirtualSwindle only performed a client-side attack on each application, and the attack had no server-side effect on the tested applications (i.e., an in-app transaction was never really completed). Furthermore, if the attack succeeded, we stopped interacting with the app, and made sure that no server-side logs were affected (e.g., online high scores). Also, the vulnerable apps we tested only had in-app currency that was generated on the device, and the virtual currency was deleted once the app was uninstalled.

5.2 Selection Process for In-App Billing Apps

Determining if an application is using in-app billing is not a straightforward process. Although in-app billing does require a special permission (i.e., `com.android.vending.BILLING`), it is not possible to search the Play Store for all apps that support this feature. Furthermore, just because the permission is present and the application asks for it does not necessarily mean that the application uses in-app billing.

One possible, straightforward idea to discover which apps use in-app billing would be to crawl the Play Store and to look at the descriptions of the apps in the Play Store. However, app developers are not obliged to disclose that they have in-app billing enabled in their applications. Furthermore, currently, Google does not make an effort to mark applications in the Play Store as being in-app billing enabled.

To select applications for testing, we developed some simple heuristics to narrow down the selection choice to apps that had a high probability of having in-app billing functionality. To determine if an app has in-app billing services, we first check if the billing permission is present. Furthermore, if we determine that an application has any of the following three characteristics, we decide that it will likely support in-app billing.

a) The game has an in-app currency, and the player has to acquire it. It is likely that she has to buy it through in-app billing.

b) The game has multiple levels. In many games, in-app billing can be used to buy additional levels.

c) The game is ad-supported. In many cases, in-app billing can be used to remove ads, or to upgrade to the full version.

Using these simple heuristics, we selected 100 candidate apps based on popularity, and downloaded them from the Play Store's `Top Free` and `Top Grossing` categories.

Once an application is installed and executed, we can observe calls to the vending API by intercepting `sendBillingRequest` invocations to determine if the app is indeed using in-app billing. However, once again, this is not always as straightforward as one might expect. About half of the applications we tested use the `CHECK_BILLING_SUPPORTED` request during startup to determine if the device is capable of supporting in-app billing. Unfortunately, the other half check for in-app billing only just before the purchase task is about to be executed.

Hence, we had to manually interact with most of the apps to determine if they were in-app billing capable. This entailed playing

Figure 8: Purchasing coins in the game Big Win Slots.

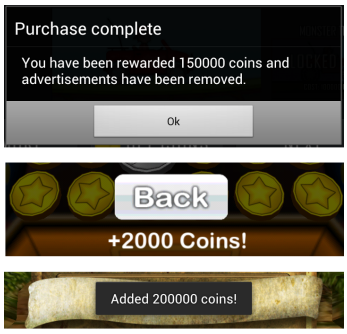

Figure 9: Purchase confirmations from three different apps: Hill Climb Racing, Coin Dozer, and Temple Run.

many games and attempting to, for example, buy digital content. In 85 apps, we were able to trigger the in-app billing functionality, and used our attack prototype on these apps. Note that our list of apps includes highly popular games such as Angry Birds, Temple Run, and Flow Free that have millions of users. For instance, Angry Birds is reported to have 500 million users.

In our test set, we observed item prices between $0.99 USD and $99 USD. It is not uncommon for games to sell currency packs for differing amounts. Figure 8 shows a screenshot from the game Big Win Slots where the user is prompted to buy in-game coins in exchange for real currency. In contrast, the removal of ads and game levels usually cost around $2.00 USD.

5.3 Attack Experiments

Once we had selected 85 popular apps that were confirmed to use in-app billing, the next step of the evaluation was to determine how effective our attack was against these applications. If our attack was successful (i.e., the fake in-app purchase worked), we quit the application, and restarted it to see if it remembered the items that we had purchased. If, on the other hand, the application was not vulnerable to our attack, we inspected the Android Debug Bridge (ADB) log to identify countermeasures the application might have implemented. Furthermore, we reverse engineered the application to gain a deeper understanding of its countermeasures.

In our experiments, we used multiple indicators to determine if an application was vulnerable. First, an obvious indicator of a successful attack is positive visual feedback from the app after a purchase that indicates that the items have successfully been purchased. Figure 9 depicts the purchase confirmation messages from three applications drawn from our test set. Second, we observed the log output from our code that runs inside the vending process. The log shows us the exact state of the purchase action. Third, the log output of our patched signature verification function served as a useful indicator for on-device verification.

If an app was not vulnerable, the first obvious indication that the attack had failed was that there was no visual confirmation of success. Furthermore, the log message for the `CONFIRM_NOTIFICATIONS` command would be missing, and there might be general log messages indicating the use of remote signature verification. Note that if the signature verification is performed remotely, the log file for the subverted `java.security.Signature.verify` method will be empty.

We manually ran VirtualSwindle for each test app. To trigger the in-app purchase request, in some cases we had to go through a lengthy interaction with the app in question.[1]

[1] We note that many virtual zombies were killed in the course of this research.

5.4 Analysis of Security Countermeasures

Of the 85 apps we analyzed in our experiments, 51 (60%) were automatically cracked by VirtualSwindle. 33 apps were not cracked automatically, and we determined that one application still contained the Play Store test item that is provided for testing purposes (`android.test.purchased`).

One of the goals of our evaluation was to investigate to what extent popular apps follow Google's in-app billing security recommendations [13]. For our test set, we first downloaded all 85 applications to our test device. Note that this was not straightforward for all apps: three apps used App Encryption, a content security feature that was introduced with Android 4.1. To bypass App Encryption, we had to execute the applications and manually copy the decrypted files from the `/mnt/aesc` directory to the `/sdcard` directory before we could download them to our test device.

Second, we disassembled each of the 85 applications using Apktool [1]. Apktool unpacks the APK files and disassembles the DEX classes, among other functions. We were then able to manually analyze the disassembled applications and investigate the security countermeasures employed by the applications. The reverse engineering and analysis effort was largely manual, but we did use some simple guidelines.

5.4.1 *Locating the billing code*

In many cases, obfuscated apps will still contain the well-known names of the calls to the Android framework and imported interfaces (e.g., `sendBillingRequest`).

5.4.2 *Checking if Java reflection is used*

If `sendBillingRequest` cannot be located, it is likely that the application is using Java reflection [16] to dynamically resolve classes and methods to further obfuscate the application. The use of reflection can often be quickly ascertained by checking the code for the string "reflect".

5.4.3 *Checking for the use of dynamic strings*

Google advises developers to manually assemble strings that are related to in-app billing such as `inapp_signed_data` and `inapp_signature` [13]. If these strings are not found, it is likely that the application is constructing them on the fly, or is obfuscating them in some other way.

5.4.4 *Checking network traffic*

For all applications in our test set that we were not able to crack automatically, we also analyzed the network traffic they produced during the purchase process. To accomplish this, we installed and executed tcpdump directly on the Android device.

5.5 Findings and Key Insights

Table 1 in the Appendix summarizes our key findings for each of the 85 apps that we analyzed. It lists whether or not the application is vulnerable to our automated attack, and the type of countermeasures we identified in each application.

Seven of the vulnerable applications in our test set included countermeasures such as code obfuscation and reflection. We discovered these countermeasures only after manually analyzing the apps, since our attack was dynamic, and thus not affected by such techniques.

One finding that was surprising was that Big Win Slots, a popular app with over one million installs, does not perform any signature verification at all. Based on our testing, we knew that the application was vulnerable. However, we discovered that the on-device signature was never triggered during these tests. Further investigation revealed that the developer, in fact, was not checking for valid purchase signatures.

In our test set, we found evidence that some developers indeed take in-app billing security seriously. For example, Grab Money Slots uses on-device signature verification, but implements it in native code. Naturally, such code is more difficult to attack automatically. As another example, Outdoor Atlas, an app that provides access to maps that can be purchased via in-app billing, performs signature verification on a remote server, and does not deliver maps to the device there until the purchase had succeeded. Clearly, the developers of this app were following the remote signature and content distribution recommendations in the Google in-app billing guidelines [12].

We found one app, KungFu, that still contained the in-app billing test item (`android.test.purchased`). We conclude that the developer probably evaluated in-app billing, added in-app billing functionality, but had not yet enabled it.

All other non-vulnerable applications implemented server-side signature verification. In some rare cases, only server-side signature checks are performed. In most cases, though, the server-side signature check is an addition to the on-device signature checks. Hence, some developers seem to be doing a good job in securing their in-app functionality.

We also observed that in a large number of applications, the disassembled billing code was identical to the code used in other applications. We discovered that this was because many applications were using the in-app billing example code provided by Google. Note that as a security precaution, Google explicitly warns developers against reusing the provided example code as-is. Specifically, 38 of the vulnerable apps we analyzed reused Google's example in-app billing code verbatim, while 15 of the vulnerable apps not vulnerable to our attack also showed evidence of code reuse.

6. COUNTERMEASURES

After determining that most countermeasures we found during our evaluation did not stop our attack, we investigated possible countermeasures. We recognize that there is no silver bullet defense against offline attacks based on manual reverse engineering. The only way to counter static code patching attacks is to design applications in a way that they require heavy interaction with a server component, thus making client-side patching approaches useless. But many applications do not require a server component for their actual functionality and, therefore, developers do not wish to invest time and money to include a server component in their application. Therefore, we investigated ways to improve the client-side security of the billing process against automated attacks. We focused on methods that are lightweight, easy to adopt, and do not incur the additional costs of a server component.

Our work shows that the use of reflection and obfuscation do not protect against instrumentation-based automated attacks. The weak point of the applications cracked by our automated attack is the easily identifiable and patched purchase data signature verification routine. We concluded that a viable working countermeasure is to not rely on signature verification code provided by the Android libraries.

6.1 Hardening the Signature Check Code

A basic assumption of instrumentation-based automated attacks is that all important symbol, class, method, and attribute names are known to the attacker. This is especially true if the names are part of a standard API. Our proposed defense against VirtualSwindle-like attacks is to introduce artificial diversity by replacing fixed names with dynamic names. This forces the attacker to invest time to disassemble and analyze every individual application, and drives up the per-target costs for the attacker.

Our approach for replacing fixed function names with dynamic functions names requires multiple steps. First, the developer must inline the functionality required for signature verification to the application. Second, the application must be modified to use the inline signature verification code instead of that provided by the Java framework via `java.security.Signature`. Third, the names of classes and methods comprising the signature verification code must be randomized on a per-application basis.

6.2 Hardening Implementation

We implemented and evaluated our approach by creating an example application named `com.example.paymenttest`. For the signature verification, we leveraged SpongyCastle[2], an Android version of the well known open source Java cryptographic library BouncyCastle. We implemented our approach in three steps: *a*) we renamed the package `org.spongycastle` to `com.example.payment test`, *b*) copied the library source files into the application's source directory, and *c*) randomized the imports in the application's source to reflect the renaming of the SpongyCastle package.

To implement the renaming of classes and methods, we used the well-known obfuscation tool ProGuard. ProGuard is already deeply integrated into the Android build process and can easily be enabled in the application's build configuration. However, switching on ProGuard is not sufficient for our purposes since, by default, ProGuard renames symbols in a deterministic fashion. For example, `org.example.cryptotest.crypto.digests.SHA1Digest` might always be mapped to `org.example.cryptotest.b.a.b`. To address this, we implemented randomized renaming through ProGuard's ability to accept a user-provided dictionary to map input symbols to obfuscated symbols. Therefore, instead of renaming `crypto.digests.SHA1Digest` to `b.a.b`, we can rename it to a random obfuscated symbol that changes with each application build.

The result of our compile-time obfuscation technique is that the signature verification code now is part of the actual application, shares the same namespace, and all known symbols are removed from the library. We verified our method through unpacking and inspecting a number of APKs created using these steps. We found no common names besides the applications MainActivity class, which serves as an application entry point specified in the application manifest that must remain unmodified.

6.3 Discussion

We believe our approach for hardening the signature verification code against automated attacks provides a beneficial trade-off between developer effort and improved security. Not every appli-

[2]https://github.com/rtyley/spongycastle

cation developer has the resources to deploy a back-end server for his application. Furthermore, our approach can be easily added to existing applications with minimal effort.

7. RELATED WORK

It is well-known that strong security properties are difficult to guarantee on remote devices controlled by untrusted and potentially malicious users. Much work in the trusted systems domain has examined this problem [19, 17, 23]. Unfortunately, at the same time, security systems have been built in the past that have relied solely on client-side enforcement. For example, smart-card systems that are widely used by European governments often rely on functionality provided by the operating system, and can often easily be bypassed [20]. Our work shows that many app developers seem not to be aware of dynamic attacks against their applications that can easily subvert in-app purchasing. We show that the current difficulty bar for the attacker is very low, automated attacks are possible, and that the popular protection techniques such as reflection and obfuscation can often easily be evaded.

To the best of our knowledge, we are the first to examine the security of Android's in-app billing in detail and to present a real-world, practical attack that compromises its security. There has been some recent work that has looked at the security of Apple's in-app purchasing service, however. For instance, the In-AppStore project [26] created an attack against the Apple iOS in-app payment system. This attack is different than ours, as it is based on redirecting the device's network traffic to the In-AppStore servers. For the redirection, an attackers has to change the DNS settings on the iOS device and install a custom SSL certificate. Note that the security measures of Apple's in-app payment architecture are not comparable to Android's in-app billing before iOS version 6 – for instance, Apple did not digitally sign any of the purchases prior to that iOS release.

Other work has demonstrated attacks against in-app billing [18], but this approach is based creating an alternate Android store that contains modified APKs. The techniques described in this work require disassembling, modifying, and re-packaging all applications. In the process, they render about 20% of the applications non-functional. Additionally, they break the APK signature which renders the application update mechanism provided by the Google Play Store non-functional. The user must also download applications from their store and not Google Play. As a result, their attack is far from being executed in reality and does not pose a real threat to Google and developers.

In contrast, our attack is fully dynamic and only takes place on the device. We do not need to modify application APKs. Users can install applications from the Google Play Store. Our attack can be switched on and off on the device since it does not carry out any permanent changes. Finally, we do not need to touch the target application APK as it exists on device storage at all.

Our work is also related to research that has been performed on game security. For example, previous work has investigated the potential for cheating in games by passively monitoring application memory in order to extract data that is helpful for the cheater [3]. Other work has looked at ways to improve the resilience of games against cheating by building in server-side checks [15, 2]. These works, however, are not concerned with in-app billing. In a recent DEFCON talk, Stracener and Barnum presented ways to cheat in YoVille, a popular game on Facebook [21]. As part of this work, they were able to create and steal in-game items. Unlike VirtualSwindle, however, the presented attacks were manual and game-specific.

Our dynamic Dalvik instrumentation approach shares similarities with other work that was carried out in parallel, specifically the Xposed [22] framework and Cydia Substrate for Android [11]. These two projects were created for device modding and require replacing system components such as zygote. Our approach targets stock Android devices and does not rely on replacing core components of the Android system. We released the source code for our DDI implementation at `http://github.com/crmulliner/ddi/`.

There has also been some work that has investigated approaches for the static instrumentation of Dalvik code. I-ARM [8] builds a reference monitor that is statically patched into an application binary. The authors disassemble the application's DEX classes, and statically modify their existing behavior by adding their own classes and replacing method calls to point to their newly added code. After the modification, they have to recompile the application.

Similarly, Aurasium [24] builds a reference monitor into application binaries. The Dalvik code is not patched, but new classes and native code are added to ensure that the instrumentation code is run first. Once the instrumentation code runs, functionality that the authors are interested in (e.g., native socket accesses) can be intercepted. Clearly, such approaches are not effective if the code is obfuscated and protected against static analysis and disassembly. Also, note that the package signature of the instrumented applications are broken when they are patched statically. In comparison, our approach does not need access to source code. Our modifications are in-memory only, and thus we do not break code signing. Furthermore, our modifications can be inserted or removed at runtime.

DroidScope [25] presents a dynamic malware analysis system for Android. The system is based on an Android emulator that the authors instrument in order to be able to track the execution of binary samples. The instrumentation is lightweight and allows the authors to produce Dalvik execution traces. In comparison, our Dalvik dynamic instrumentation library provides the capability to perform in-depth instrumentation of Dalvik classes as well as the native code of running Android applications as well as the Android system itself. In contrast, DroidScope only instruments the VM, while our main focus is the instrumentation of the applications (although we can also instrument the VM).

Recently, there have also been work that has investigated the privacy behavior of Apple as well as Android apps. For example, Enck et al. [10] present TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. Using TaintDroid, the authors monitor the behavior of 30 popular third-party Android applications, and discovered 68 instances of potential misuse of users' private information across 20 applications. Similarly, Egele et al. [9] present PiOS, a tool for statically analyzing iOS apps for privacy threats. PiOS uses static analysis to detect data flows in Mach-0 binaries compiled from Objective-C code. Our work is orthogonal to these projects, and focuses on subverting existing functionality on an Android system to break the security of in-app billing.

8. CONCLUSION

Android's in-app billing service allows developers to provide mobile apps for free, but charge users for digital services from within the apps themselves. In this paper, we present VirtualSwindle, the first fully-automated attack against Google Play's in-app billing service. To the best of our knowledge, the work we present in this paper is the first to provide an in-depth analysis of how app developers are using Android's in-app billing service, and to measure the robustness of these apps against automated attack.

We present an attack that is able to automatically compromise the security of many popular Android apps. One of the goals of this work is to *raise awareness* among developers, and to counter the popular folk wisdom that following the obfuscation guidelines in Google's in-app billing documentation will significantly deter attackers from compromising their apps. We show that this is indeed not the case, and that dynamic attacks can be performed against many popular apps with ease.

We performed empirical experiments and tested our attack prototype on 85 popular apps that make use of in-app billing. We present detailed reports on the in-app security defenses implemented by these apps. 60% of the apps we analyzed were automatically and easily crackable using our attack prototype.

Based on our findings we created a lightweight countermeasure that protects against automated attacks. Our countermeasure is easy to adopt and improves the security of existing applications against automated attacks such as VirtualSwindle.

Acknowledgements

This work was supported by the Office of Naval Research (ONR) under grant N000141310102. Engin Kirda thanks Sy and Laurie Sternberg for their generous support.

9. REFERENCES

[1] Apktool Developers. android-apktool - A tool for reverse engineering Android apk files. `http://code.google.com/p/android-apktool/`, November 2012.

[2] Bethea, D., Cochran, R. A., and Reiter, M. K. Server-side verification of client behavior in online games. In *17th ISOC Network and Distributed System Security Symposium (NDSS)* (2010).

[3] Bursztein, E., Hamburg, M., Lagarenne, J., and Boneh, D. OpenConflict: Preventing Real Time Map Hacks in Online Games. In *IEEE Symposium on Security and Privacy* (May 2011).

[4] Chainfire. SuperSU. `https://play.google.com/store/apps/details?id=eu.chainfire.supersu&hl=en`, November 2012.

[5] Clowes, S. injectso - Modifying and Spying on running processes under Linux and Solaris. `http://www.blackhat.com/presentations/bh-europe-01/shaun-clowes/bh-europe-01-clowes.ppt`, 2001.

[6] CLShortFuse. SuperOneClick. `http://forum.xda-developers.com/showthread.php?t=803682`, November 2012.

[7] CyanogenMod Developers. CyanogenMod. `http://www.cyanogenmod.org/`, November 2012.

[8] Davis, B., Sanders, B., Khodaverdian, A., and Chen, H. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *Workshop on Mobile Security Technologies (MoST)* (May 2012).

[9] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Network and Distirbuted Systems Security Symposium (NDSS)* (2 2011).

[10] Enck, W., Gilbert, P., Chun, B., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Symposium on Operating Systems Design and Implementation (OSDI)* (2010).

[11] Freeman, J. Cydia Substrate for Android. `http://www.cydiasubstrate.com/`.

[12] Google. In-app Billing. `http://developer.android.com/guide/google/play/billing/`, November 2012.

[13] Google. In-app Billing Security and Design. `http://developer.android.com/guide/google/play/billing/billing_best_practices.html`, November 2012.

[14] HTC. Unlock Bootloader. `http://htcdev.com/bootloader/`, November 2012.

[15] Mitterhofer, S., Platzer, C., Kirda, E., and Kruegel, C. Server-side Bot Detection in Massively Multiplayer Online Games. *IEEE Security and Privacy Magazine* (5 2009).

[16] Oracle. The Reflection API. `http://docs.oracle.com/javase/tutorial/reflect/index.html`, November 2012.

[17] Petroni Jr, N., Fraser, T., Walters, A., and Arbaugh, W. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security Symposium* (2006).

[18] Reynaud, D., Song, D., Tom Magrino, E. W., and Shin, R. POSTER: FreeMarket: Shopping for free in Android applications. In *ISOC Network and Distributed System Security Symposium (NDSS)* (February 2012).

[19] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating System Principles (SOSP)* (2005).

[20] Spalka, A., Cremers, A., and Langweg, H. Trojan Horse Attacks on Software for Electronic Signatures. In *Informatica* (2002).

[21] T. Stracener and E. A. Smith and S. Barnum. So Many Ways to Slap a Yo-Ho: Hacking Facebook and YoVille. `http://www.defcon.org/images/defcon-18/dc-18-presentations/Stracener-Smith-Barnum/DEFCON-18-Stracener-Smith-Barnum-So-Many-Ways.pdf`, August 2010.

[22] Vollmer, R. Xposed. `http://repo.xposed.info/`.

[23] Watson, R. N. M. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *USENIX Annual Technical Conference, FREENIX Track* (2001), pp. 15–28.

[24] Xu, R., Saidi, H., and Anderson, R. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium* (August 2012).

[25] Yan, L. K., and Yin, H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security Symposium* (August 2012).

[26] ZonD80. Getting started to receive your in-app for free on iOS. `http://system.in-appstore.com/`, Ocrober 2012.

Appendix

Table 1 below summarizes our key findings for each of the 85 apps that we analyzed.

#	Name	Notes
1	**Angry Birds**	
	Angry Gran Run	
	Bad Piggies	
	Big Win Slots	does not verify the signature
	Clouds & Sheep	
	Coin Dozer	
	Contract Kill 2	
	Dead Trigger	
	Death Dome	
10	**Defender II**	
	Design My House	
	DH Reloaded	
	Drag Racing Bike Edition	reflection + obfuscation
	Family Feud & Friends	
	Flow Free	
	GYRO	
	Happy Street	reflection + obfuscation
	Hill Climb Racing	
	Infinite Monsters	
20	**Jaws**	reflection + obfuscation
	Jellyflop	
	Jetpack Joyride	
	Millionaire Slots	
	Monster Pet Shop	
	My Country Online	
	Ninja Fishing	
	NinJump Deluxe	
	Plague Inc	
	Prize Claw	
30	**Robinson**	
	Rule the Kingdom	
	Slot Machnie Delux	
	Slots Journey	
	Smurfs' Village	
	Stardom	
	Style Me Girl	
	Subway Surf	
	Super Dynamite Fishing	
	Tank Hero	
40	**TapFish**	
	Temple Run	
	The Tribez	
	Tiny Monsters	reflection + obfuscation
	Tiny Tribe	
	Tiny Village	reflection + obfuscation
	Top Stylist	
	Tractor Pull	
	TripleTown	
	Zombie Frontier	reflection + obfuscation
50	**Zombie**	reflection + obfuscation
	Zombirds	

(a) Listing of 51 popular Android apps that were successfully automatically cracked by VirtualSwindle. An empty notes entry indicates that no security countermeasures were present.

#	Name	Notes
1	Arcane Empires	server-side verification
	BigWin Football	server-side verification
	Bingo Bash	server-side verification
	Bingo Blitz	server-side verification
	Bubble Mania	server-side verification
	Camelot	server-side verification
	Crime City	obfuscation + reflection + server-side verification
	Crime Inc.	server-side verification
	DerbyDays	obfuscation + server-side verification
	Dragon Story	server-side verification
10	Fatal Frontier	server-side verification
	Grab Money Slots	signature check in native code
	Hello Kitty Cafe	server-side verification
	Indestructable	server-side verification
	KungFu	only contains the developer test item `android.test.purchased`
	Legned	server-side verification
	Life is Crime	server-side verification
	Little Dragons	server-side verification
	Live Holdem Pro	obfuscation + reflection + server-side verification
	Modern War	server-side verification
20	Outdoor Atlas	server-side verification (content needs to be downloaded)
	PIMD	server-side verification
	Pumkins Vs Monsters	obfuscation + reflection + server-side verification
	Pumpkin Festival	server-side verification
	Restaurent Story	server-side verification
	Slot City	obfuscation + reflection + server-side verification
	Slotmania	server-side verification
	Slots	server-side verification
	SongPop Free	server-side verification
	Strikefleet Omega	server-side verification (blocks billing after failed attempt)
30	Stupid Zombies	server-side verification
	Texas Poker	obfuscation + reflection + server-side verification
	TinyFarm	server-side verification
	Zynga Poker	obfuscation + reflection + server-side verification

(b) Listing of 34 popular Android apps that employ countermeasures against in-app billing attacks.

Table 1: Security analysis of 85 Android apps that use in-app billing.