FreeMarket: Shopping for free in Android applications

November 30, 2011

Abstract

In March 2011, Google launched In-App Billing (IAB) for Android applications, a service that lets application developers easily sell digital goods by having Google handle the complexity of user authentication and credit card processing. However, much complexity remains in implementing the IAB protocol securely in order to prevent piracy (i.e., users getting access to the digital goods without paying). Some of the security recommendations from Google significantly increase the complexity of development, testing and debugging, while performing server-side validation can even incur hosting and bandwidth costs. As a result, a significant number of IAB applications rely only on local security checks. In this paper, we design and develop techniques for a systematic study of IAB vulnerabilities in Android applications. We show that at least 174 applications (more than 50% of the applications we tested) in the Android Market do not perform server-side validation and can be automatically exploited in order to bypass the IAB payment system. We describe and implement the FreeMarket attack, which we carry out by creating a fake Market application and rewriting the application so that it binds to it instead of the real Market. Additionally, we rewrite the signature verification code in the application so that our fake Market appears to be the real one. As a result, every in-app purchase succeeds without incurring any form of payment, effectively making every item free.

1 Introduction

Since the introduction of In-App Billing in late March 2011, it has become an important way for Android developers to monetize their applications. It facilitates for developers the selling of digital goods in their applications, such as virtual items or additional levels in games, and digital content such as music, movies, and e-books. At the time of writing, 16 out of the top 24 grossing apps[1] are free to download and generate revenue through IAB. No official statistics report the number of apps using in-app billing, but according to 3rd-party statistics website AppBrain[2], about 100 new apps with IAB have been released each week, for a total of 1990 such applications available on the Android Market at the time of writing. In this paper, we discuss how application developers can mitigate IAB piracy, i.e., malicious users trying to access in-app digital goods without paying, and we show that a significant number of apps currently distributed on the Market are subject to automatic exploitation.

It is noteworthy that even though Google manages the user authentication and credit card processing with Google Checkout, the IAB protocol is intrinsically complex. This complexity is visible in the SDK sample, which has 1,571 lines of Java code as reported by SLOCCount [20]. Specifically, in addition to providing a user interface for purchasing digital items, developers must deal with a mix of synchronous and

[1] Top-grossing applications are applications that generate the most revenue through the Android Market, either because they are paid apps or sell digital goods through IAB.

[2] http://www.appbrain.com/stats/in-app-billing-android-applications

asynchronous messages, multiplexing, delays due to loss of Internet connectivity, cryptographic signatures and nonces, and purchases and refunds across devices. In addition, the documentation advises developers to design and implement their own solutions for server-side validation, tamper-resistance, secure storage of the purchase database, and obfuscation. We provide a technical overview of the IAB protocol in Section 2.

Most studies about Android security focus on protecting users against potentially malicious code [3,6,7, 9, 10, 21]. In this paper, we instead focus on protecting code against potentially malicious users. Protecting code that runs under an attacker's control is notoriously hard, and is not specific to smartphone applications. A classic attack is to patch out security checks, turning protected code into unprotected code, and has been applied for decades for removing copy protection in commercial software and games. The commonly accepted assumption is that stopping a dedicated attacker is impossible [5], so most protection schemes are only meant to increase the cost of releasing a pirated version of the application. This relies on the false assumption that pirating software is a craft that requires manual effort. In particular, we identify a class of security vulnerabilities prevalent in at least 58.98% of the applications we tested. We are able to automatically attack the weakest link in applications that otherwise use recommended security practices such as code obfuscation, reflection, native code, integrity checking on the local purchase database and remote content delivery. Therefore, the presence of at least one easy attack vector renders useless all the development effort that went into implementing, testing and debugging these protection layers.

We found that a large portion of IAB applications distributed on the Android Marketplace did not implement server-side validation and integrity checks (see Section 5 for experimental results). In order to demonstrate that no manual effort is required to exploit them, we design a new form of attack. We build a fake Android Market service called FreeMarket. We then rewrite vulnerable applications as follows: we download the executable image of an application, translate it to Java bytecode (we describe the translation process in section 4), rewrite the bytecode so that all purchases succeed without payment, and we finally recompile and resign the application using the usual Android toolchain. The rewriting, documented in Section 3, is minimal: we force the application to bind to a fake Android Market service called FreeMarket that responds to every purchase as if the user actually paid, and we also make sure that the spoofed messages appear authentic by patching the signature verification code. As a result, every purchase request will trigger a sequence of messages that will appear to be a successful purchase, and the application will consider the messages as coming from Google servers. This attack is immune to the use of code obfuscation, reflection, native code (although rewriting of native code is not in the scope of this paper) and secure storage of the purchase database.

Therefore, we strongly advocate *performing server-side validation* and *employing integrity checks*. These recommendations would not prevent dedicated attackers from pirating the application, but it would prevent the application from being *automatically* exploited, so they should be considered mission-critical for revenue-generating applications.

We make the following contributions:

- We identify a critical weak point in implementations of the In-App Billing protocol, and demonstrate that vulnerable applications are subject to automatic exploitation;
- We design and implement the first Android application analysis framework that is capable of rewriting Android applications and is compatible with existing Java bytecode analysis tools;
- We tested 295 free applications available on the Android Marketplace, and were able to automatically exploit 174 (58.98%). 60 of these are in the top 2000 grossing applications.
- We demonstrate that the weakness we identified is prevalent even in successful applications, and that

additional measures are needed to help developers secure their revenue-generating applications.

2 Technical overview of the In-App Billing service

In this section, we briefly introduce the IAB service, how messages are exchanged between the application and the Android Market, and highlight sensitive security operations.

2.1 IAB Protocol

Before items can be sold through IAB, they must be declared to Google on the Android Market publisher site. For each application, the developer creates a product list that associates product IDs and their price. The price is therefore known in advance by the Market server and is never directly part of the transaction; this differs significantly from other Cashier-as-a-Service systems [19]. The publisher site also generates an RSA key pair associated to the developer's account and provides the developer with the public key; this key will be used later on to authenticate messages from the Market server.

In order to use IAB, the application connects to the Android Market application on the device. Though the requests are handled by the Market server on the Internet, the application never needs to connect to it directly as all messages are exchanged through Inter-Process Communication (IPC) with the Market application. Figure 1 depicts the different entities involved in IAB transactions and how they communicate.

A simplified typical purchase then looks as the following:

1. When the user wants to purchase an item, the application connects to the Android Market application and sends a `REQUEST_PURCHASE` message with a product ID corresponding to this item. This string, for instance `sword_001`, must be in the product list of the application on the Market server.

2. The Market responds asynchronously with a `PendingIntent` that is used to display the Google Checkout UI to the user, with a price corresponding to the product ID. Internet connectivity is required for this step. Google Checkout processes the user financial information on behalf of the application.

3. Assuming the user proceeds with the payment, the Market then sends an `IN_APP_NOTIFY` message, with a notification ID. This message means that one or more orders have a new status (purchased, canceled or refunded), and the application should retrieve information about these orders.

4. The application requests this information with a `GET_PURCHASE_INFORMATION` message with the corresponding notification ID and a nonce.

5. The Market application responds with a `PURCHASE_STATE_CHANGED` message. This message contains the nonce and is signed with the private key associated with the developer's account.

6. The application processes the new state of the item and displays the result to the user.

2.2 Problem statement

For developers who want to to avoid IAB piracy in their applications, the main challenge involves maintaining the application's integrity while it runs in untrusted environments controlled by potentially malicious users. We can imagine a number of techniques available to users who wish to evade payments, many of who have superuser privileges on their devices: they could modify the code of the application, perform

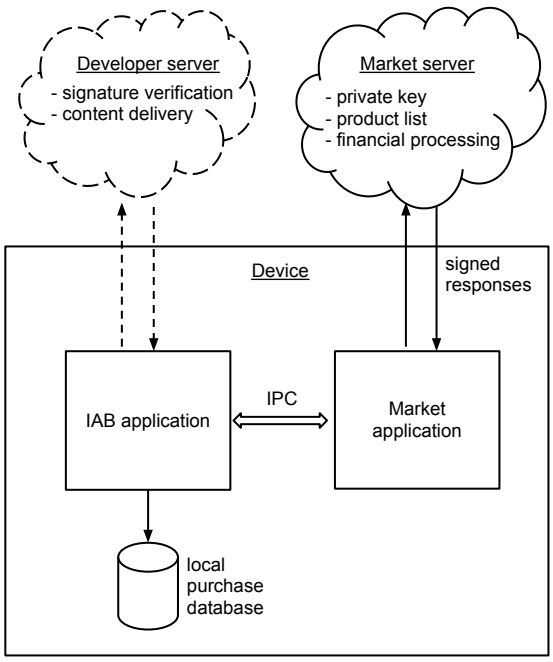

Figure 1: In-App Billing Architecture

man-in-the-middle attacks, replay or spoof messages from the Market billing service, and so on. In this paper, we play the role of the untrusted user and demonstrate that a large class of applications are vulnerable to automatic circumvention of the IAB payment system. The attack is described in Section 3 and evaluated experimentally in Section 5.

3 The FreeMarket attack

3.1 Principle of the attack

The security of the IAB protocol rests on ensuring that the application communicates with the real Market application. It does so using the following mechanisms: (1) by relying on the Android framework to route its messages to the correct application, (2) by checking the signature of incoming messages and (3) by checking that nonces have expected values (an attacker could replay a message with a valid signature, but this would fail with correct nonce generation and verification).

The attack we propose is to rewrite the application in order to remove checks that the user paid for IAB purchase requests. It works in two steps: we first force the application to bind to a fake Market service, and then we bypass the message authentication mechanism. In other words, we subvert assumptions (1) and (2), and we completely bypass (3) because we can reply with the correct nonces. The changes to the application are minimal and are performed automatically. We detail the technique in Section 3.2 and present an overview in Figure 2. The rewritten application runs on any Android device; it does not require the user to have a rooted phone or to modify its system. The only pre-requisite is to install the FreeMarket application

(i.e., the fake Market service).

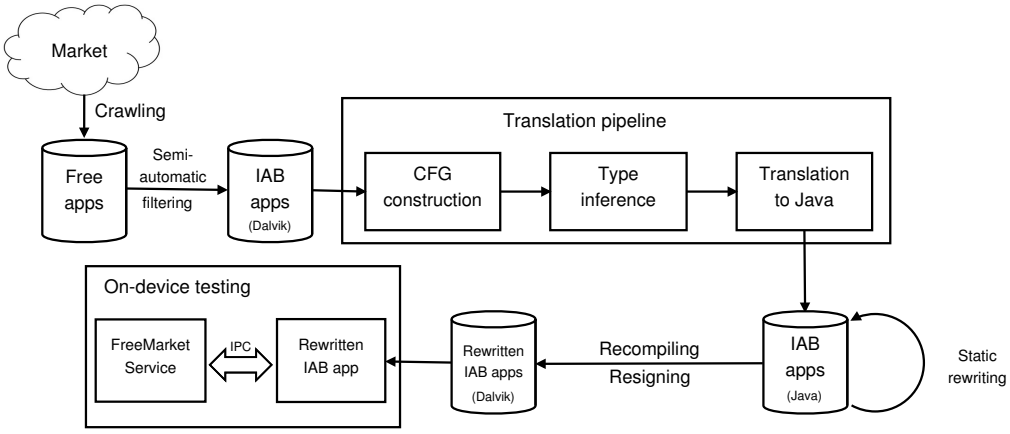

Figure 2: FreeMarket attack architecture

3.2 Implementation of the attack

We propose to inspect the code of the application and to patch relevant portions in order to circumvent the IAB payment system. In general, Android applications can be made of Dalvik bytecode (usually derived from Java source code), native code and LLVM bitcode[3]. We limit our scope to the Dalvik portion of the application, although in general our attack could be extended to also rewrite the native code.

We start by translating the application to Java bytecode using the procedure described in Section 4. We then use the ASM library[4] to statically rewrite the bytecode. The attack works in two steps: we must ensure that the application connects to FreeMarket instead of the real Market, and we must patch its signature verification logic. The rewriting approach we use is conceptually similar to user-level system call interposition [11, 12, 16], since we must ensure that we gain control dynamically at key points during the program's execution without modifying the underlying platform. Instead of using it for defensive purposes before potentially dangerous operations, we use it for offensive purposes to change the outcome of sensitive operations. Experimental results are presented in Section 5.

Spoofing the Market billing service. The Market billing service is an Android Service component that applications can bind to. The only documented way to do so is to use the method `bindService(Intent service, ServiceConnection conn, int flags)` in class `android.content.Context`, with an intent action of `com.android.vending.billing.MarketBillingService.BIND` (this can be thought of as the "address" in the Android framework of the Market billing service). This call will cause an implicit intent to be sent, that the Android framework will route to the appropriate service in the Market application. Since the address is a dynamic string value, we cannot statically rewrite it to point to FreeMarket. Therefore, in order to hijack the connection, we replace invocations of the `bindService` method with invocations of a proxy method that we provide. This method dynamically checks the address (i.e., the intent action) of outgoing bind requests, and re-routes requests to the Market billing service. As a result, the outgoing requests

[3] http://developer.android.com/guide/topics/renderscript/index.html
[4] http://asm.ow2.org/

never reach the real Market billing service; they are all handled by FreeMarket instead. Note that the call to `bindService` can not be obfuscated, although it can be called via reflection—which we handle separately.

Patching the signature verification logic. Since `PURCHASE_STATE_CHANGED` messages must be signed with the private key associated with the developer's account (which is stored on the Market server), the FreeMarket component cannot generate messages with valid signatures. Routing requests to FreeMarket is therefore not sufficient. We patch the signature verification code so that it always causes messages coming from FreeMarket to be considered valid. Given the complexity of cryptographic algorithms and the availability in the standard Java library of cryptographic APIs, most applications rely on them to perform signature verification. We automatically identify call sites to these public libraries and replace them with calls to a method that always returns the value expected after a successful verification. Specifically, we replace invocations of the methods `verify(Signature sig, byte[] signature, int offset, int length)` and `verify(Signature sig)` in class `java.security.Signature` with invocations of a method that always returns `true`. This relies on specific API calls and is therefore in general not robust, but gives good results in practice. In the future, we may add support for other third-party cryptographic libraries that could be used as a drop-in replacement for the Java standard library.

Handling reflection. The Dalvik runtime supports reflection, so it is possible to load classes and invoke methods dynamically by looking up their name. Resolving the string values passed to reflection APIs is undecidable, but we can always statically resolve calls to the `invoke(Object obj, Object... args)` method in class `java.lang.reflect.Method`. Therefore, we replace invocations of `invoke` with a proxy method that allows us to inspect dynamically which method is being called. If `bindService`, `verify`, or `invoke`[5] is being called, then we call the appropriate proxy method instead. With this technique, we handle reflection completely.

Resigning. After rewriting, the signature of the application is no longer valid, so we need to resign it before it can be installed and run on an Android device. There is no special handling for this case, we just reuse the existing SDK tools. This means that the rewritten application is no longer tied to the identity of the original developer.

4 Translating from Dalvik to Java bytecode

In the majority of cases, Android applications are developed in Java, compiled to Java bytecode, and finally compiled to Dalvik bytecode. In this section, we address the problem of performing the opposite transformation, i.e., from Dalvik to Java bytecode, so that we can leverage existing bytecode analyzers, decompilers and rewriters.

In order to do this, we must account for differences between the two architectures (Section 4.1), and in particular we must recover lost type information (Section 4.2). Once the appropriate operand types have been recovered, mapping Dalvik bytecodes to their Java counterpart is straightforward.

4.1 Differences between Dalvik and the Java Virtual Machine

At a high level, the logical structure of Dalvik and Java Virtual Machine (JVM) programs are very similar: a program is a collection of classes, each class containing typed fields and methods with typed arguments,

[5] recursion allows us to handle nested layers of reflection

typed return values, explicit control flow and exception handlers. Most of the development effort therefore went into accounting for a relatively small number of differences. Some of these differences are already documented in [8]:

- Per-class versus per-application constant pool: each JVM class has its own class file, with its own constant pool. In contrast, all constant pools in a Dalvik program are merged into a single application-wide constant pool.
- Stack-based versus register-based: the JVM instruction set architecture is usually described as stack-based, whereas the Dalvik ISA is register-based. This distinction is not entirely relevant however since local variables in JVM methods act like registers, and it is straightforward to translate Dalvik registers to JVM local variables.
- Fully-typed versus under-typed: some Dalvik instructions are under-typed, for instance `move v0, 0` (load 0 into register `v0`) could correspond to the JVM instructions `iconst_0` (push integer 0) and `aconst_null` (push null). We need to perform type inference to recover this information. We describe our algorithm in Section 4.2.

Another fundamental difference is the handling of exception handlers: for the JVM verifier [13], exceptions can be triggered anywhere in a try block, whereas for the Dalvik verifier exceptions can only be triggered at certain instructions. Figure 3 demonstrates a data flow that is valid for the Dalvik verifier (only the definition at line 2 reaches the catch block, because only the division instruction can throw an exception), whereas the JVM verifier would detect conflicting assignments to v in the catch block (both assignments at lines 2 and 4 reach the catch block). In order to translate the code in Figure 3 to JVM bytecode while conserving data flow properties, we place the instructions that can throw exceptions in their own try block[6].

```
1 try {
2         v = 1;
3         v = v / 0;
4         v = null;
5 } catch(Throwable t) {
6         use(v);
7 }
```

Figure 3: Data flow test case

4.2 SSA-based type inference

Performing type inference on Dalvik bytecode is not trivial, because even though source-level variables are statically typed, a Dalvik-level register can hold different variables along different code paths. Because of this, the path-insensitive Hindley-Milner algorithm [14] or Tenenbaum's bidirectional dataflow analysis [1, 18] would produce many type conflicts. This led to the intuition that using a form where reaching definitions are explicit such as Single Static Assignment (SSA) would greatly reduce the complexity of

[6]Since we use the same technique to account for this situation, we believe it is what has been described incorrectly as "type system violations" in [15], where in fact the Dalvik bytecode is valid with regards to the Dalvik type system.

the type inference algorithm [4]. Therefore we infer the types of under-typed variables in 4 steps: we disassemble the application using Google's dexdump tool, then for each method we build its control flow graph (CFG), convert it to SSA form using the algorithms in and finally perform type inference on this SSA representation [2].

CFG construction. We build intra-procedural CFGs in three phases: we first add a start block with synthetic definitions representing the arguments passed to the method, we split the code in basic blocks as usual and add control flow edges as usual based on jump targets, and finally we add exceptional flow edges.

Because of Dalvik's special handling of exception handlers described in section 4.1, we can not make the usual abstraction that exceptions can be thrown in the middle of a basic block, as this would introduce dataflow errors. Therefore, we use the following rule for exceptional flow edges.

Definition 1 (Exceptional flow edge rule) *Let i be an instruction that can throw exceptions and t be a try block such that i is in t. Then for every instruction j that directly precedes i (either by fall-through or by a jump), we add an exceptional flow edge from j to every handler of t.*

This can seem counter-intuitive since this rule can introduce exceptional flow edges from instructions that are not in a try block. A way to understand it is that when an instruction causes an exception to be thrown, then control flows to the exception handler before the instruction could kill any definition. This property is needed so that the conversion to SSA computes an appropriate renaming. Figure 4 shows the natural CFG that would be inferred with Java-style exception handling for the program in Figure 3, while Figure 5 shows the CFG we build with Dalvik-style exception handling.

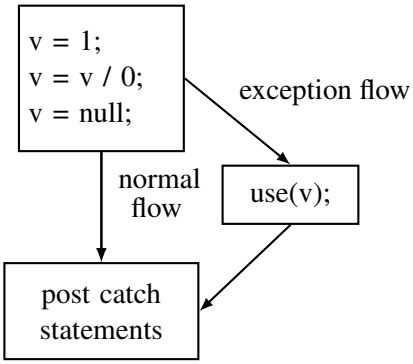

Figure 4: Java CFG

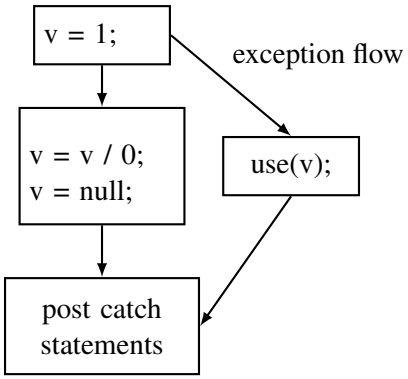

Figure 5: Dalvik CFG with exceptional flow edge rule

Type inference algorithm. We infer types based on the type lattice in Figure 6. It is defined by the relation $x < y \implies$ a value of type x can be assigned to a variable of type y. Therefore, the bottom element $\bot$ represents the "unknown" type, whereas the top element $\top$ represents the "conflict" type. It contains types that represent under-typed variables ($\bot$, *dword*, *zero*, *reference* and *primitive*) and over-typed variables ($\top$).

Since our algorithm takes a CFG in SSA form, every variable is defined exactly once and can be used zero or more times. Therefore, we will refer to the definition type of a variable and to its use types (i.e., the type inferred for it at instructions that use it). We use ϕ-functions to easily compute *reaching use sites*:

Definition 2 (Reaching use sites) *Let v_0 be a variable, its reaching uses $\mathcal{R}(v_0)$ are given by:*

- *if instruction i uses v_0, then $i \in \mathcal{R}(v_0)$*
- *if an instruction $v_n = \phi(v_0, v_x, \ldots, v_y)$ uses v_0, then $\mathcal{R}(v_n) \subset \mathcal{R}(v_0)$*

For the purpose of type inference, the *use types* of a variable will refer to the set of types for this variable at its reaching use sites.

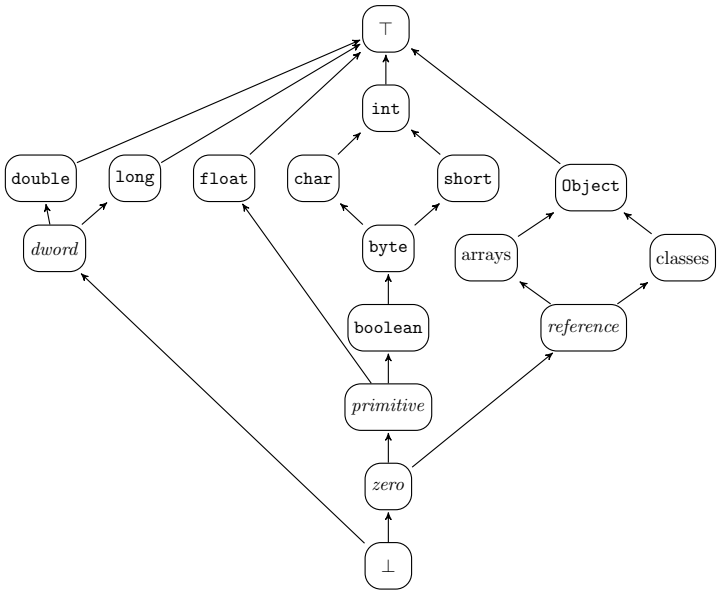

Figure 6: Type inference lattice

Definition 3 (Type inference problem) *The problem of type inference for method m is to assign a definition type d and a set of use types U to each variable v in m such that v is not under-typed or over-typed, and $\forall u \in U, d \leq u$.*

We propose Algorithm 1 for solving the type inference problem. In its current incarnation, it is a simple iterative algorithm, we may consider switching to a worklist algorithm instead if it becomes a bottleneck. It converges because it is monotonic (i.e., we only assign types that move up in the lattice) and it operates on a lattice of finite height.

Like the type inference algorithm described informally in [8], it matches under-typed definition sites (an ambiguous register declaration in their terminology) with use sites that expose the type. But instead of walking the CFG for each definition site, we rely on the SSA form to expose the def-use chains.

The procedure inferTypesFromOpcode returns a type for each variable based on the opcode of the instruction. Initially, some variables can be under-typed. For instance:

- `const v0, 0`: we infer that v0 has type *zero* (it could be a single-word primitive or a reference)
- `const v0, 1`: we infer that v0 has type *primitive*, since it can not be a reference
- `const-wide v0, 0`: we infer that v0 has type *dword*, it could be a double or a long

- `check-cast v0, type`: we infer that v0 has the indicated use type on the fall-through path of the instruction, and use type *reference* on the exceptional path
- `aget v0, v1, v2` (get element v0 at index v2 of array v1), we infer that v2 has a use type of *int*, v1 has a use type *reference* and v0 has a definition type of *primitive* (we know it fits in a single word and is not a reference, but we do not further know its type);
- instance and static field operations, method invocations, new-instance and new-array instructions all expose the types of their operands
- all operands of a ϕ-function are $\perp$

The procedures propagateDef and propagateUse are used to represent instruction-specific logic, for instance:

- `move v0, v1`: when the definition type of v0 (resp. the use type of v1) is resolved, the same type is propagated to v1 (resp. v0)
- `aget v0, v1, v2`: when the type of v1 is resolved to be an array of X, then X is propagated to the definition type of v0
- $v_n = \phi(v_x, \ldots, v_y)$: when the type of a used variable is resolved, we set $v_n = lub(v_x, \ldots, v_y)$ (lub is the type lattice least upper bound operator)

The fixRemainingTypes procedure assigns default types to variables that remain under-typed after the repeat loop. This happens for instance when a variable is never used or is only used in comparisons as described in [8]. In this case, it is safe to assign a default type of int or long, depending on the precision of the variable.

Algorithm 1 Type inference algorithm

```
Input: variables, the set of variables in the method
  (deftype, usetypes) ← inferTypesFromOpcode()
  repeat
    for var ∈ variables do
      d ← deftype[var]
      U ← usetypes[var]
      if d is under-typed then
        comment: glb is the type lattice greatest lower bound operator
        comment: it returns ⊥ when its input is the empty set
        inferred ← glb({u ∈ U s.t. u > d})
        if inferred > d then
          propagateDef(var, inferred)
      for (i, u) s.t. instruction i uses var with type u ∈ U do
        if d > u then
          propagateUse(var, i, d)
  until no variable changes
  fixRemainingTypes(deftype, usetypes)
  return (deftype, usetypes)
```

5 Experiments

In this section, we describe the experiments we performed in order to empirically establish the validity of our automatic rewriting and determine the prevalence of insecure practices in real applications that allow us to automatically carry out our attack.

5.1 Procedure

Collecting applications. Google does not provide a specific list of applications which use IAB. However, as they provide a top-2000 list of available applications by gross revenue, we can infer that the 161 applications which are free to download on that list use (or have used in the past) IAB. We also collected about 60,000 free applications from the Market, among which we considered applications declaring the `com.android.vending.BILLING` permission, a prerequisite for using IAB.

Filtering collected applications. For the following reasons, we exclude some of our collected applications from testing:

- Not all of our collected applications actually use IAB. Among the top-2000 applications, some may have stopped using IAB, yet remain on the list. Similarly, a developer may have declared the IAB permission for an application, but not implemented the functionality, or exposed it in the user interface.
- Some applications (18.98% of our test set) became inoperable (e.g., crashing immediately upon launch) after rewriting. We feel that these do not represent any fundamental limitations, and are manually investigating the causes to fix the responsible bugs in our infrastructure.
- Some applications were incompatible with our testing device, used a language none of the experimenters knew, or otherwise presented insurmountable obstacles to manually exercising the IAB portion.

Testing applications. To carry out the actual attack, we use our one-click tool on each application, which converts the application's Dalvik code to Java bytecode, automatically rewrites the Java bytecode, then translate the rewritten code back to Dalvik and repackage it with the original accompanying resources. Then, to verify that attack was successful, we install the modified application and manually exercise the IAB functionality within it to see if we can buy things for free. We performed our testing on Nexus S devices running Android 2.3.3.

5.2 Results

There were 161 applications among the top 2000 grossing list that were free to download, among which we tested 126 applications with IAB. In total, we tested 295 applications, by adding a randomly-chosen subset of the IAB applications we collected that were not in the top 2000 grossing list.

As shown in Figure 8, over all applications tested, we found that:

- **58.98%** (174 applications) are vulnerable to the FreeMarket attack.
- **22.03%** (65 applications) detected the attack and refused to process the transaction.
- **18.98%** (56 applications) became inoperable after rewriting.

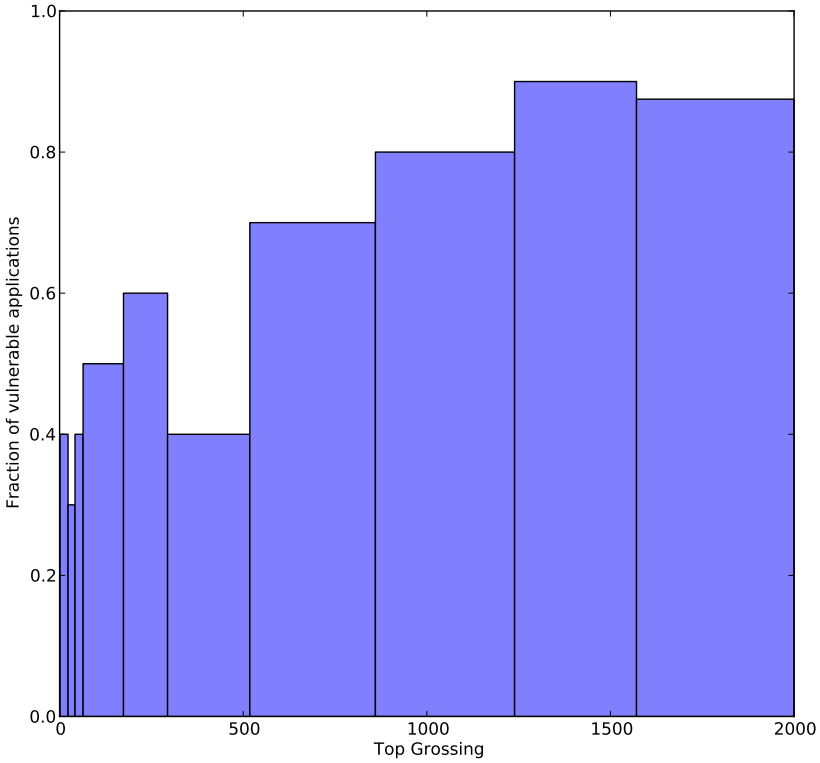

Figure 7: Applications grouped into bins of ten by their ranking on Google's top 2000 grossing list. Each bin has width corresponding to the range of rankings within the bin.

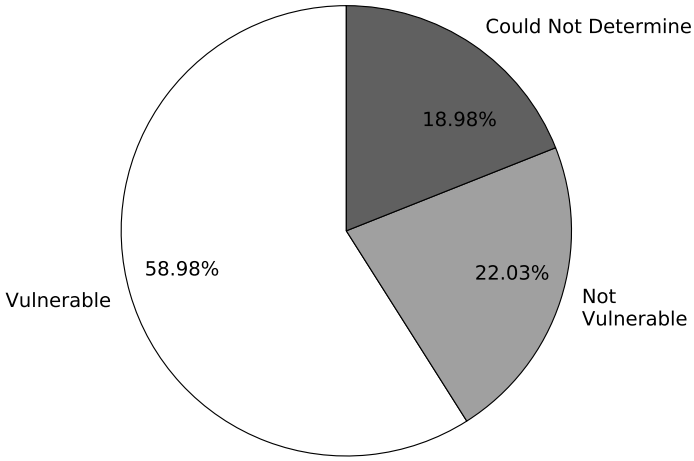

Figure 8: The distribution of applications tested that were found to be vulnerable, not vulnerable, or might be vulnerable (could not determine) to the FreeMarket attack.

Among the applications with IAB that were on the top 2000 grossing list, Figure 7 shows how our rate of success varies as the rank of the application changes. Each bar represents the fraction of applications which were vulnerable in a group of ten (all taken from a range of ranks on the top 2000 grossing list, indicated by the width of the bar).

5.3 Discussion of results

Causes of attack failure. Based on our experience, it seems that the applications that detect the attack do so mainly because they perform server-side validation. We also observed that some applications pass the purchase signature to native code through JNI, indicating that they perform validation in native code, which we do not handle. We have not seen any evidence of tamper-resistance mechanisms being employed.

Relation between success and popularity. As we can see in Figure 7, applications with greater revenue tend to be immune to our attack more often than applications with less revenue. Certainly, for developers who make significant revenue from their applications, greater incentives exist for developers to securely implement IAB—however more complicated it may be—in order to protect their income. Perhaps we can additionally that developers of higher-quality applications, which tend to attract more users, also pay greater attention to details of securely implementing the IAB protocol.

6 Related work

To implement the FreeMarket attack, we needed to be able to decompile, transform and recompile Android applications; and we also wanted to leverage existing program analysis frameworks. We looked for existing tools for Android application analysis, including dexdump (part of the Android SDK), dedexer[7], smali/baksmali[8] and dex2jar[9]. To the best of our knowledge, the only toolchain available to perform application rewriting was smali/baksmali. Unfortunately, these tools use a peculiar syntax based on Jasmin's Java assembler syntax, which means that any static analysis tools would have to be written from scratch for this syntax and for the Dalvik semantics. Instead, we considered translating from Dalvik to JVM bytecode because of the similarity between the two platforms and because of the availability of program analysis tools and decompilers that work directly on JVM bytecode. This led us to the same design choice as ded [8], which we implemented independently during the same time frame.

At the beginning of this project, only dex2jar was available for retargeting Android applications to JVM bytecode. It seems to be primarily used for manual inspection of Android applications, i.e., decompilation. Based on our experience, it is not suitable for recompilation, as the generated Java bytecode is not structurally correct and the JVM bytecode verifier refused to load classes generated by dex2jar in too many instances to be usable for our purpose.

Towards the end of the project, ded has been made available and we have been able to compare our tool against it. In [8], ded is used for one-way conversion from Dalvik to JVM bytecode. The JVM bytecode is further analyzed, optimized, and decompiled to Java statements. In contrast, we focus on producing bytecode that will be processed by the Dalvik compiler and yield a working Android application again. As a result, ded has likely not been tested for the round-trip conversion, and so we were unable to use it for recompilation. Even for simple applications such as the "Hello Android" SDK sample, the generated classes

[7] http://dedexer.sourceforge.net/
[8] http://code.google.com/p/smali/
[9] http://code.google.com/p/dex2jar/

are rejected by the Dalvik compiler. Since the source code of ded does not appear to be available, we did not diagnose the root cause(s) of this problem.

Redexer [17] is a tool that supports Dalvik-to-Dalvik rewriting. It is used to retrofit Android applications to use trusted APIs. It rewrites applications without translating them to an intermediate language, and thus does not meet our requirements.

We summarize our comparison with other Android application analysis and rewriting tools in Table 1.

Name	Targets a supported language	Allows rewriting
dexdump	no	no
dedexer	no	no
smali/baksmali	no	yes
dex2jar	yes	no
ded	yes	no
redexer	no	yes
our tool	yes	yes

Table 1: Comparison with existing toolchains

7 Discussion and recommendations

Even though its potential financial impact is important, the attack we performed is described in a comment in the IAB SDK sample:

> "Instead of just storing the entire [public key as a] literal string embedded in the program, construct the key at runtime from pieces or use bit manipulation (for example, XOR with some other string) to hide the actual key. The key itself is not secret information, but we don't want to make it easy for an adversary to replace the public key with one of their own and then fake messages from the server."

The recommendation misses the point however, since once the attacker has access to the code of the application, replacing the definition of the public key is not needed. It is sufficient and simpler to not use the key at all and replace the signature verification code with a tautology as we did. Therefore, the recommendation should be to *obfuscate the signature verification logic, not the public key generation logic*.

Unfortunately, the signature verification logic in the SDK sample is impossible to obfuscate, since it relies on the public class `java.security.Signature` in the Java standard library. An obfuscated version could be obtained by refactoring the code so that it links statically against a Java cryptographic library and then using an obfuscator to mangle the verification logic.

As a more robust way to mitigate our attack, we strongly recommend using server-side validation to verify cryptographic signatures and implementing tamper-resistance mechanisms.

Official security guidelines In Table 2, we present an overview of the security recommendations from Google found in [5] and in the "Security and Design" section of the IAB documentation and we discuss how they would impact our results.

Note that using the License Verification Library is not applicable in our context, since it is currently limited to paid applications only, while the IAB monetization model is more relevant for free applications.

Security recommendations from Google	**Mitigates the FreeMarket attack**
Perform signature verification tasks on a server	yes
Protect your unlocked content	no
Obfuscate your code	no
Modify all sample application code	no
Use secure random nonces	no
Implement a revocability scheme for unlocked content	yes
Protect your Android Market public key	no
Use reflection	no
Use native code	currently, yes
Use the License Verification Library	N/A
Use tamper-resistance mechanisms	yes

Table 2: Overview of security recommendations

8 Conclusion

We have demonstrated that a critical weakness in implementations of the In-App Billing protocol was prevalent in a majority of the applications we tested, including 3 of the top 10 grossing applications. Some of these applications use code obfuscation, native code, integrity checks on the local purchase database and remote content delivery but are still vulnerable to the FreeMarket attack.

A lesson learned is that even though developers realize that they need to follow security recommendations in order to mitigate piracy, they don't have accurate expectations about the capabilities of attackers. We have showed that a one-time investment for the attacker was sufficient to circumvent the payment system in a large portion of applications currently distributed on the Android Market.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[2] A. Appel and J. Palsberg. *Modern compiler implementation in Java*. Cambridge Univ Pr, 2002.

[3] A. Beresford, A. Rice, N. Skehin, and R. Sohan. Mock-Droid: trading privacy for application functionality on smartphones. *HotMobile 2011*, 2011.

[4] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[5] T. J. Dan Galpin. Evading Pirates and Stopping Vampires using License Verification Library, In-App Billing, and App Engine. In *Google I/O*, 2011.

[6] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. *Information Security*, pages 346–360, 2011.

[7] 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 *Proceedings of OSDI*, 2010.

[8] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. of the 20th USENIX Security Symposium*, 2011.

[9] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. Technical report, UCB/EECS-2011-48, University of California, Berkeley, 2011.

[10] A. Felt, H. Wang, A. Moshchuk, S. Hanna, E. Chin, K. Greenwood, D. Wagner, D. Song, M. Finifter, J. Weinberger, et al. Permission re-delegation: Attacks and defenses. In *20th Usenix Security Symposium, San Fansisco, CA*, 2011.

[11] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium*, pages 187–201, 2004.

[12] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. *Proc. Network and Distributed Systems Security Symposium*, 2000.

[13] T. Lindholm and F. Yellin. *Java virtual machine specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[15] D. Octeau, W. Enck, and P. McDaniel. The ded decompiler. Technical report, NAS-TR-0140-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, 2010.

[16] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th conference on USENIX Security Symposium-Volume 12*, pages 18–18. USENIX Association, 2003.

[17] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster. Application-centric security policies on unmodified Android. Technical report, 110017 UCLA Computer Science Department, 2011.

[18] A. Tenenbaum. *Type determination for very high level languages.* PhD thesis, New York University, 1974.

[19] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *2011 IEEE Symposium on Security and Privacy*, pages 465–480. IEEE, 2011.

[20] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount.

[21] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming Information-Stealing Smartphone Applications (on Android). *Trust and Trustworthy Computing*, pages 93–107, 2011.