# DIVILAR: Diversifying Intermediate Language for Anti-Repackaging on Android Platform

Wu Zhou *, Zhi Wang †, Yajin Zhou *, Xuxian Jiang *
*North Carolina State University †Florida State University
wzhou2@ncsu.edu zwang@cs.fsu.edu yajin_zhou@ncsu.edu jiang@cs.ncsu.edu

## ABSTRACT

App repackaging remains a serious threat to the emerging mobile app ecosystem. Previous solutions have mostly focused on the postmortem detection of repackaged apps by measuring similarity among apps. In this paper, we propose DIVILAR, a virtualization-based protection scheme to enable self-defense of Android apps against app repackaging. Specifically, it re-encodes an Android app in a diversified virtual instruction set and uses a specialized execute engine for these virtual instructions to run the protected app. However, this extra layer of execution may cause significant performance overhead, rendering the solution unacceptable for daily use. To address this challenge, we leverage a light-weight hooking mechanism to hook into Dalvik VM, the execution engine for Dalvik bytecode, and piggy-back the decoding of virtual instructions to that of Dalvik bytecode. By compositing virtual and Dalvik instruction execution, we can effectively eliminate this extra layer of execution and significantly reduce the performance overhead. We have implemented a prototype of DIVILAR. Our evaluation shows that DIVILAR is resilient against existing static and dynamic analysis, including these specific to VM-based protection. Further performance evaluation demonstrates its efficiency for daily use (an average of 16.2% and 8.9% increase to the start time and run time, respectively).

## Categories and Subject Descriptors

K.5.1 [**Hardware/Software Protection**]: Copyrights

## Keywords

Android; Anti-Repackaging; Virtual Machine

## 1. INTRODUCTION

In recent years, mobile apps have gained unprecedented adoption due to the increasing popularity of smartphones and other mobile devices. For example, recent data shows that Google and Apple have accumulated more than 48 billion and 50 billion app downloads from their online app store, respectively [8, 25]. Online app store provides many effective ways for developers to monetize their apps, such as paid apps, in-app purchase, and ads. Unfortunately, app repackaging, in which legitimate apps are repackaged and sold without the knowledge or consent of their rightful owners, poses serious threats to this new economy model since its inception: app repackaging can lead to proliferation of app piracy, causing monetary lose for developers [12, 22, 29, 53, 73, 74]; even worse, repackaged apps are often used as the media for stealthy malware [73–75]. Recent research shows that repackaged apps have mostly plagued third-party app stores, which are virtually unregulated in the distribution of those apps [29, 74]. With little help available from the operators of those stores, app developers have to resort to technologies that can enable *self-defense* for their apps.


*CODASPY'14,* March 3–5, 2014, San Antonio, Texas, USA.

ACM 978-1-4503-2278-2/14/03 ...$15.00.
http://dx.doi.org/10.1145/2557547.2557558.

```
const-string v0, "content://com.example.notepad.provider.NotePad/notes"
invoke-static {v0}, Landroid/net/Uri;->parse(Ljava/lang/String;)Landroid/net/Uri;
move-result-object v0
sput-object v0, Lcom/example/android/notepad/e;->a:Landroid/net/Uri;
```

Figure 1: Snip of code processed by Proguard (only app's class names are changed).

In light of these serious threats, researchers have proposed different approaches to address the problem. For example, a number of systems aim at detecting repackaged apps by measuring similarity among a large set of apps, particularly between third-party stores and the official store [12, 29, 53, 73, 74]. They are designed to provide postmortem detection and jurisdiction, and thus cannot prevent apps from being repackaged at the first place. To thwart app piracy, Google and others introduced server-side license verification [32, 52] that can be employed by linking to provided libraries. However, plain deployment of these libraries can be easily bypassed. There even exist automated tools just for that purpose [43–45]. Server-side verification is thus advised to be deployed with other protections such as obfuscation to make it difficult to be bypassed [32]. Existing Android-based obfuscators [36, 41] are relatively easy to bypass due to the fact that Dalvik bytecode contains rich semantics of an app. Figure 1 shows a short snip of code processed by Proguard [41], arguably the most popular obfuscator for Android due to its inclusion in the official Android developer's guide. Notice that calls to the Android framework APIs are virtually not changed, providing "valuable" information to the repackager of where and how to make modifications (although it is technically viable to obfuscate these calls with Java reflection and string transformation, it likely will lead to unacceptable overhead because such calls are highly frequent.) In this paper, we propose DIVILAR(DIVersified Intermediate Language for Anti-Repackaging), an Android protection scheme that diversifies the Dalvik bytecode to significantly raise the bar against app repackaging.

DIVILAR draws its inspiration from an effective and reliable technology on the traditional desktop systems: virtual machine(VM)

based protection, in which the target binary is re-encoded in a randomized virtual instruction set and packed together with a specialized interpreter for the instruction set. At run-time, the interpreter translates and executes the protected app (for brevity, we call it the guest app). By encoding the guest app in an unknown instruction set, VM-based protection immediately disables existing tools that expect the original instruction set as input. It thus becomes necessary for the attackers to first reverse-engineer the VM interpreter in order to recover and manipulate the guest app. However, an interpreter could be implemented in a highly convoluted and obscure way. For example, the original QEMU (before TCG) [6] demonstrates how complicated a *benign* emulator (a type of virtualization) could be. Moreover, virtual instructions can be readily randomized and customized for individual software (or software revisions), further increasing the reverse-engineering efforts. As such, VM-based protection has been successfully applied by both malware authors and developers to protect their "intellectual property" from being analyzed and reverse-engineered [21, 46, 61, 62, 65].

Successful deployments of VM-based protection have been largely limited to native binaries, but not for managed code such as Java Bytecode or Microsoft Intermediate Language (MSIL) [21, 46, 61, 62, 65]: performance overhead caused by the extra layer of instruction interpretation is prohibitive for an interpreter in the managed code, but has little effect over a native interpreter. A native interpreter not only executes faster (because it is executed natively), but also can employ mature technologies such as fast binary translation [4] and just-in-time compilation [40] to improve performance. In contrast, managed code already requires a layer of interpretation/translation. The additional layer of execution can further significantly slow down the system. For example, Proteus [2], a VM-based protection system designed for MSIL, reports from $50\times$ to $3500\times$ performance overhead. Such an overhead is even more prohibitive on a mobile platform due to its limited computation power and battery life.

To address this challenge, we observe that Android already has a mature and efficient interpreter for its apps, Dalvik VM. Dalvik VM is loaded as a native library into each app's address space, which is shared with the app's own native libraries. The latter thus can freely access and manipulate Dalvik VM at run-time. DIVILAR leverages this accessibility to dynamically hook into Dalvik VM and decode the virtual instructions right before they are loaded for execution. By doing so, the interpreter for the virtual instructions is merged into the existing execution engine of Dalvik bytecode. This not only significantly reduces overhead of the extra layer of execution, but also allows DIVILAR to be much harder to reverse-engineer: compared to interpreters implemented in managed code [2], DIVILAR's interpreter is implemented as a native binary. Therefore, many coding/obfuscation techniques not available to managed code (e.g., direct memory manipulation) can be employed and a large body of existing obfuscation tools can be directly applied to it.

We have implemented a prototype of DIVILAR. It accepts an Android app as input, transforms its Dalvik bytecode into a randomly generated intermediate language, and wraps the resulting binary together with a lightweight virtual instruction interpreter. DIVILAR is intended to be used by app developers at the last stage of development before releasing the app into public online app stores. DIVILAR-protected apps need no special treatment from Android or the app store. They can be released, installed, and executed in the same way as normal Android apps. Our evaluation demonstrates that DIVILAR is robust against common countermeasures, including existing static and dynamic analysis, and even these specific to VM-based protection. DIVILAR also has a small and acceptable performance overhead with an average of $16\%$ increase to app start time and $8.9\%$ to run time.

In summary, this paper makes the following contributions:

- First, we propose a diversified virtualization-based protection for Android to thwart app repackaging. DIVILAR-protected apps are compatible with the existing Android ecosystem. To the best of our knowledge, it is the first VM-base protection system for Android.
- Second, we design a lightweight in-app hooking mechanism for DIVILAR's interpreter to composite virtual instruction and Dalvik bytecode execution. Evaluation shows that our prototype interpreter is robust and incurs small performance overhead.
- Third, we have implemented a prototype of DIVILAR. Our evaluation demonstrates that DIVILAR is effective against common countermeasures, including static analysis, dynamic analysis, and particularly these specific to VM-based protection or obfuscation.

The rest of the paper is organized as the follows: we give background information and state the app repackaging problem in Section 2, and describe the DIVILAR design and implementation in Section 3 and Section 4, respectively. After that, we present the robustness analysis and evaluate its performance in Section 5. In Section 6, we discuss the system's limitations and suggest some possible improvements. Lastly, we present related work in Section 7 and conclude the paper in Section 8.

## 2. PROBLEM STATEMENT

In this section, we introduce key concepts of Dalvik VM, the execution engine for Android apps, and the widespread issue of app repackaging in the Android platform.

### 2.1 Android App and Dalvik VM

Most Android apps are written in the Java programming language with some components optionally implemented as native libraries. At compiling time, the Java source code is compiled into Dalvik bytecode, which are then assembled into a single Dalvik executable (the `.dex` file). Each Android app can optionally load native libraries and interact with them through the JNI interface [14]. To release an app, all files of the app are compressed into an `apk` file that is subsequently signed with the developer's private key. The developer's key is usually self-certified without involvement of a central certificate authority. The signature of an app therefore cannot be used as an indication of trustworthiness of the developer, or that the signer is the rightful owner of the app. Instead, Android security model uses the signature to safely allow apps of the same developer to share certain privileges.

Android apps are executed by Dalvik VM, a register-based Java virtual machine designed by Google. Figure 2 shows the high-level architecture of Dalvik VM. There are two major components: class loader and execution engine. Class loader is responsible for loading Dalvik classes for execution. Specifically, when a new class needs to be executed, class loader reads its definition from either the app's `dex` file which contains all the classes of the app, or from system libraries. The class definition includes information such as fields, methods, and bytecode for each method (except abstract method). For safety, class loader needs to perform a fairly complicated verification to make sure the class is well-formed and does not violate constrains set by the Java specification [42].

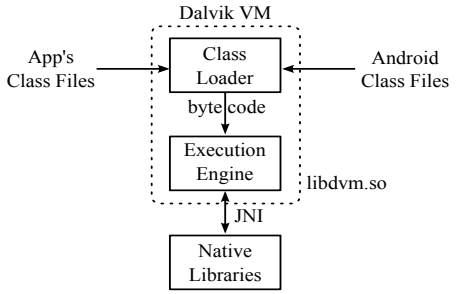


Figure 2: Architecture of Dalvik Virtual Machine

Since DIVILAR uses a completely different instruction encoding than Dalvik bytecode, it is necessary for DIVILAR to bypass the verification of class loader.

Execution engine of Dalvik VM decodes and executes the bytecode of Java methods. Dalvik bytecode has a different encoding scheme than the standard Java bytecode: it is register-based (the standard Java bytecode is stack-based) and has variable lengths ($1, 2, 3$, or $5$ code units, each code unit is 16-bit). For example, the `add-int` instruction has an opcode of $90h$ and three register operands encoded in 2 code units. By re-encoding an app in a virtual instruction set, DIVILAR will lead existing Android analysis tools (e.g., the baksmali disassembler [19]) to misinterpret the instruction boundaries and thus fail to decode the instruction stream. The original implementation of Dalvik VM executes bytecode through opcode-dispatching, in which it fetches and decodes the bytecode into opcode and operands, and dispatches execution to the corresponding handler for the opcode. Recent versions of Dalvik VM speed up the execution by supporting just-in-time compilation, where frequently-executed bytecode is compiled into native code and executed natively.

Dalvik VM is shipped as a native library (`libdvm.so`) and loaded into each app's address space, which it shares with other app-provided native libraries. The virtual instruction interpreter of DIVILAR is also a native library (`libhook.so`). At run-time, `libhook.so` hooks into Dalvik VM and manipulates its internal data structure for its own purposes.

## 2.2 App Repackaging in Android Platform

App repackaging has plagued the Android platform. For example, a recent survey of top 100 paid and free apps on the Android (and iOS) platform shows that most of these popular apps suffer from app repackaging [64]. App developers are also finding their apps pirated, leading to considerable revenue loss [16,22,60]. Even worse, researchers found that 86% of Android malware samples are distributed to their victims through repackaged apps [75]. A few reasons contribute to the proliferation of app repackaging in the Android platform: first, most Android apps are written in Java whose bytecode retains rich semantics of the apps. This makes Android apps relatively easy to understand and modify without source code. Second, as mentioned earlier, Android apps are signed with self-certified keys. The signature of an app does not establish integrity or ownership of the app. Third, Android apps can be distributed through non-official channels such as third-party app stores where virtually no regulation or vetting process exists. Android users are also free to install apps of unknown origin (e.g., downloaded off the web).

With the help of publicly-available tools such as smali/baksmali [19], repackaging an Android app is straightforward: the attacker only needs to disassemble the app to locate points of modification. The modification can be as simple as changing the ad client ID (in order to redirect ad revenues to the attacker) or as complicated as tweaking of internals of the app [1]. After that, the attacker can reassemble the app and sign it with *his or her* own key. The repackaged app can now be distributed in (third-party) online app stores. This process is so well-defined that automated tools have been designed to repackage apps [70]. Given the easiness of this process, there is a pressing need to prevent app repackaging in the first place, in addition to the postmortem detection and jurisdiction [12,29,53,72–74]. To this end, DIVILAR applies the effective VM-based protection to Android apps. It aims to significantly increase the efforts needed to reverse-engineer and modify the app. By encoding apps in a different virtual instruction set, it immediately disables all existing tools that are based on Dalvik bytecode. The attacker has to reverse-engineer the DIVILAR execution engine for virtual instructions in order to manipulate the app.

# 3. DESIGN

## 3.1 DIVILAR Overview

DIVILAR aims at protecting Android apps from being repackaged by re-encoding them in a diversified virtual instruction set. We have three design goals for DIVILAR:

**Robustness**: although fundamentally it is an arms race, we expect DIVILAR to be robust against existing countermeasures, including commonly available static analysis, dynamic analysis, and these specific to VM-based protection. DIVILAR should also be architecturally flexible to accommodate new countermeasures.

**Compatibility**: DIVILAR-protected apps should be compatible with the current Android ecosystem. Specifically, the developers should be able to release their protected apps in the official Google play store or any third-party app stores, and users can download, install, and execute these apps just like normal apps. Compatibility is the key to the adoption of DIVILAR. Any requirements to change the Android framework will significantly limit the usability of such solutions.

**Performance**: DIVILAR introduces an extra layer of execution to managed code. With a straightforward design (e.g., implementing DIVILAR in the managed code), such a system will lead to prohibitive overhead [2]. Excessive performance overhead will render DIVILAR unsuitable for everyday use. As such, we design a lightweight hooking mechanism to merge virtual instruction execution into Dalvik VM to minimize its performance overhead.

Figure 3 shows the overall architecture of DIVILAR. It has four components: virtual instruction selector, bytecode transformer, virtual instruction interpreter, and Apk packager. Virtual instruction selector decides a virtual instruction set and produces a transforming rules (and its inverse) to describe how to translate Dalvik bytecode into virtual instructions (and vice versa). These two rules are used by the second and third components to guide conversion between Dalvik and virtual instructions. Specifically, bytecode transformer applies the transforming rule to an Android app to convert its classes from Dalvik bytecode to the selected virtual instruction set. The interpreter is an execution engine for the virtual instruction set. It is customized by the inverse transforming rule so that it can reverse virtual instructions back to Dalvik bytecode. Moreover, the interpreter is a native library to be bundled together with the guest app. At run-time, it will be loaded into the app's address space to execute the guest app. The last component, apk packager, packs the guest app and its interpreter inside a shell

[1]Java's late-binding makes this process considerably easier than native binary as symbols are dynamically resolved.

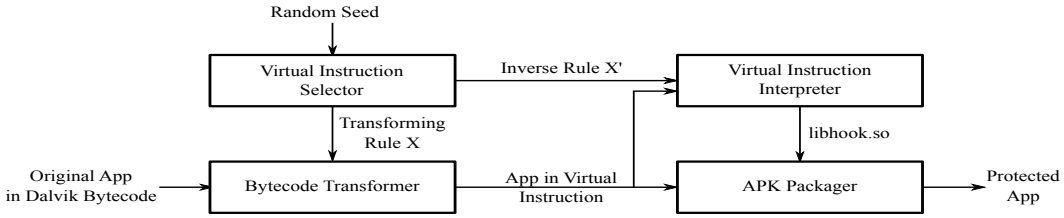

Figure 3: DIVILAR Architecture Overview

app. The shell acts as a proxy for the guest app (it is needed because the guest app is no longer recognizable by Android.) It is responsible for preparing the running environment for the guest app such as loading the interpreter. The shell also creates a custom class loader for the guest app, and uses it to load the guest's classes for execution. From Android's point of view, the shell is a well-formed app, while the guest app and its interpreter are just resources loaded by the shell. As such, a DIVILAR-protected app can pass the verification and be released, downloaded, and installed in the same way as a regular app.

DIVILAR is architecturally flexible. Various components can have varying levels of design and implementation. For example, the selector defines the conversion between Dalvik bytecode and virtual instructions. It could use a straightforward one-on-one mapping or implement complicated transformation by mixing cryptography algorithms as long as the process is reversible. Similarly, the interpreter can decode virtual instructions in different granularity: one method at a time, one basic block at a time, or one instruction at a time. Different granularity leads to different trade-off between performance (finer granularity has higher overhead) and information leakage (finer granularity leaks less information to an adversary who can monitor memory of the running app). In the rest of this section we will give more details about these four components of DIVILAR.

## 3.2 Virtual Instruction Selector

DIVILAR transforms an app encoded in Dalvik bytecode to that in a diversified virtual instruction set. Consequently, attackers are forced to first reverse-engineer virtual instructions as implemented by the interpreter, a native binary that can readily be obfuscated. In DIVILAR, each protected app is encoded in an unique set of virtual instructions decided by virtual instruction selector. The selector produces two rules: a transforming rule $X$ to convert Dalvik bytecode to virtual instructions, and its inverse $X'$ to reverse the process. DIVILAR does not pose any restrictions on the selector as long as the conversion is reversible. For example, it is possible to choose fixed or variable instruction length, apply different encoding scheme for opcodes and operands, or even use different styles of bytecode (stack-based v.s. register-based). Different choices of virtual instructions will affect DIVILAR's capability in resisting attacks that intend to reverse-engineer the interpreter.

| Dalvik bytecode | Original opcode | Target opcode | Transforming rule | Inverse rule |
|---|---|---|---|---|
| add-int | 90 | 7101 | $90 \rightarrow 7101$ | $7101 \rightarrow 90$ |
| invoke-static | 71 | 3202 | $71 \rightarrow 3202$ | $3202 \rightarrow 71$ |
| if-eq | 32 | 9003 | $32 \rightarrow 9003$ | $9003 \rightarrow 32$ |

Table 1: Example Mapping Rules for Opcode (in hex)

In our prototype, the selector is designed as a plugin of DIVILAR so that selectors with different levels of sophistication can be readily used and new selector can be created in response to unforeseen attacks. Our prototype implements a linear mapping between Dalvik opcode and virtual opcode. For easy of use by the other modules, the selector generates two mapping rules $X = \{R_1, R_2, ..., R_n\}$ and its inverse $X'$ to formally describe how to translate from and to the Dalvik bytecode. Specifically, each member of $X$ defines how the original Dalvik bytecode is transformed into a randomized virtual instruction. Table 1 gives some examples of the mappings for opcode. For example, Dalvik opcode $90h$, an `add-int` instruction, will be replaced by a two-byte opcode $7101h$. A tool that expects Dalvik bytecode will interpret this two-byte opcode as the `invoke-static` instruction (opcode $71h$) with $01h$ being the first byte of the operand [56]. Semantically, this "instruction" tries to invoke a static function (specified by 2 following bytes) with zero parameter. This mapping will also shift the instruction boundary around because $90h$ (`add-int`) has 4 bytes while $71h$ (`invoke-static`) has 6 bytes instead. Our evaluation shows that even this linear mapping can disable existing dynamic and static analysis tools, including these specific to VM-based protection. The produced set of rules will be used in bytecode transformer and virtual instruction interpreter to guide conversion between Dalvik bytecode and virtual instructions.

## 3.3 Bytecode Transformer

Bytecode transformer converts an Android app (an `apk` file) encoded in Dalvik bytecode to the instruction set decided by the selector. It first extracts from the app its Dalvik executable file (`classes.dex`), which contains definitions of all the app's classes. The transformer then decomposes the `dex` file into a list of classes and further parses them into fields and methods. A method consists of a method prototype and its associated Dalvik bytecode. At this point, the transformer converts the Dalvik instructions (including its opcode and operands) into virtual instructions guided by the transforming rule $X$. For example, in our linear mapping rule, opcode $90h$ will be converted to $7101h$, and opcode $71h$ will be converted to $3202h$ and so on. As an example of more sophisticated transformation, our prototype also supports an opcode chaining mode in which (the first byte of) the last virtual opcode is xor'ed with the next Dalvik opcode to get the input to the transforming rule $X$. For example, if the last target virtual opcode is $9003h$ (Table 1) and the next Dalvik opcode is $E1h$. The input to $X$ is $71h$ ($90h \oplus E1h$), and the target opcode will be $3202h$. Such small tweaks are effortless to implement but will make the interpreter harder to reverse-engineer. After transformation, the resulting classes will be re-assembled into a DIVILAR-executable file. This file will be packaged into the final app as a data asset to be loaded and executed by the interpreter, described in the next section.

## 3.4 Virtual Instruction Interpreter

Using virtualization to protect apps is in general an expensive operation. For example, even in the native $x86$ mode, simple implementations could incur tens of times of overhead [17]. When an extra layer of virtualization is added to managed code, the performance simply becomes unacceptable for any real-world use [2], especially for mobile platforms where computation power and battery life are severely limited. Most Android apps consist of

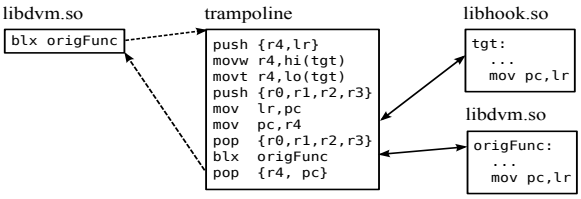


Figure 4: Function hooking in DIVILAR

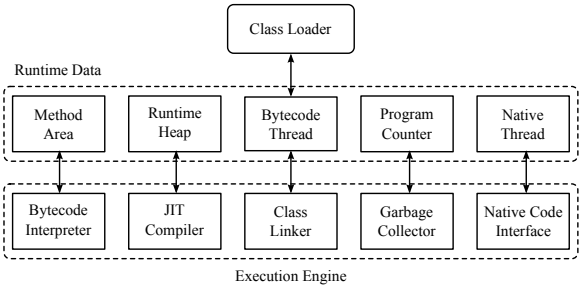


Figure 5: Components of Dalvik VM

managed code and run on mobile devices such as smartphone or tablets. Therefore, it is critical for DIVILAR to minimize performance overhead. The overhead comes mainly from the extra layer of bytecode execution. Observing that Android already contains a mature execution engine for Dalvik bytecode (Dalvik VM), DIVILAR tackles this problem by merging its interpreter into Dalvik VM, thus eliminating one layer of execution. More specifically, Android provides a Java virtual machine called Dalvik VM to execute the bytecode of apps. Dalvik VM is a mature intermediate language execution engine with rich features such as just-in-time compilation. Over the years, Dalvik VM has gradually stabilized with few updates [2]. In Android, it is shipped as a native library (`libdvm.so`) and loaded into the address space of every Android app, where it co-exists with other user-supplied native libraries (Android apps are free to load native binaries.) There exist no security boundary to isolate them (Figure 2). Therefore, it is feasible for DIVILAR to load its interpreter as a native library into the protected app, and manipulate Dalvik VM to piggy-back the execution of virtual instructions. The interpreter needs to be written in native code so it can directly operate on Dalvik VM's memory and issue system calls such as `mprotect` to change memory attributes.

To facilitate manipulation of Dalivk VM, DIVILAR provides a light-weight hooking mechanism (Figure 4) similar to Detours [31]. In DIVILAR, hooking points are function calls whose functionality needs to be extended or intercepted (`blx origFunc` in the figure). The call instruction will be overwritten with another instruction to redirected control flow to a trampoline. The trampoline is a short sequence of ARM instructions. It first saves the current CPU states (caller-saved registers $r_0 - r_4$ and the link register $lr$ that contains the return address) to the stack, and loads the address of the extending function (`tgt` function in `libhook.so`) into one of the available registers ($r_4$) for execution. After `tgt` returns, the trampoline calls the original function so the call's functionality can be maintained and augmented. Finally, the trampoline restores the CPU state and returns back to the call site. Alternatively, we can insert a detour at the beginning of the original function (instead of the call site) to hook the function. We choose the current design because it provides more fine-grained control over when and where to hook a function.

[2]This alleviates the concern that DIVILAR is too dependent on the internals of Dalvik VM and thus is less compatible.

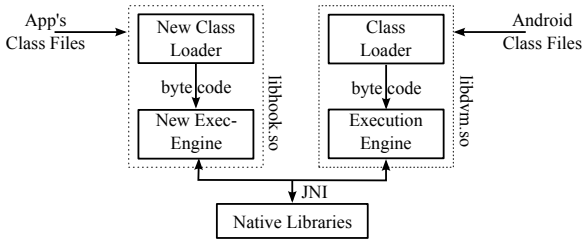


Figure 6: Architecture of Dalvik VM after DIVILAR loaded

With this light-weight hooking mechanism, DIVILAR can hook into Dalvik VM to intercept and modify its execution in order to composite execution of virtual instructions and Dalvik bytecode. Manipulation of VM states at run-time requires clear understanding of its internals and careful planning. Figure 5 shows the major components of Dalvik VM: class loader and execution engine. Both components can operate on the shared run-time data such as method definitions and the heap. Execution engine has a number of components itself such as the bytecode interpreter (to execute bytecode), JIT compiler (to compile hot spots into native instructions), and class linker (to link classes from different Java libraries). Architecturally, DIVILAR can accommodate different designs of virtual instruction execution. For example, an implementation can provide a complete execution engine ( possibly based on the Dalvik VM source code) and totally bypass Dalvik VM. Our prototype implements a pre-execution decoding scheme. More specifically, it hooks into functions that execute a method and decodes the method by reversing the opcode and operand mapping before its execution. In addition to these functions, we also need to modify class loader (so it knows how to load the guest app) as well as the meta-data in the method area. Section 4 contains details about our prototype. Figure 6 shows the logical architecture of Dalvik VM after the interpreter (`libhook.so`) is loaded. In the figure, `Libhook.so` is responsible for the loading of the guest app's classes because these classes are encoded in virtual instructions and cannot pass the sanity check of Dalvik's class loader. DIVILAR's execution engine hooks into and leverages Dalvik's execute engine to run the app. Technically, the interpreter contains complete information required to recover the original app. It is thus important to prevent it from being reverse-engineered. To this end, various obfuscation technologies for native binaries can be readily applied to `libhook.so`, such as packer/unpacker [28, 48].

## 3.5 APK Packager

One of our design goal for DIVILAR is compatibility so that DIVILAR-protected apps do not require special install-time processing or modifications to the Android framework. As such, APK packager, the last component of DIVILAR, wraps all components necessary to execute the guest app into a single shell app. The interpreter (`libhook.so`) and the guest app are both contained in the shell as data assets. The shell app is a simple wrapper of the guest app, synthesized by APK packager. From Android's point of view, the shell app *is* the main body of the final app. Therefore, it can pass the install-time and run-time verification by the Android system. The shell app inherits the permissions required by the guest app so it can execute the guest app with enough permissions. It also needs to provide a wrapper for every entry point in the guest app, such as activities, content providers, and services defined in the guest app's manifest file [35]. These entry points are automatically loaded by Dalvik VM when starting an app. At run-time, when the shell's entry points are called, they simply load the corresponding guest app classes into memory and

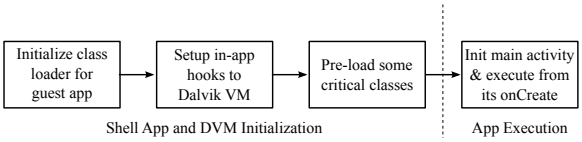


Figure 7: Execution Flow of the Protected App

dispatch them for execution. Note that this loading process does not decode virtual instructions. Instead, they are decoded only when necessary by `libhook.so`. After collecting all the required components, APK packager reassembles them into a single `apk` file which can be released, downloaded, and installed as a normal app.

The final app produced by DIVILAR consists of the shell, the guest app, and the interpreter. Figure 7 shows the run-time execution flow of the app. The shell first initializes the execution environments for the guest app. Specifically, it creates a custom class loader for the guest app's classes, loads the interpreter in the memory and hooks it to Dalvik VM, and finally pre-loads some critical classes of the guest app because Android expects these classes to be loaded upon returning from initialization. These steps need to be performed before executing the first instruction of the guest app. For this purpose, the shell app extends a special Android class called `android.os.Application`. This class (or its subclass) is guaranteed to be executed just after the (*shell*) app's classes are loaded and before the first bytecode of the main activity is executed.

# 4. IMPLEMENTATION

We have implemented a prototype of DIVILAR based on Android 4.0.4. Our prototype is a standalone Java application. It accepts an existing app as the input and generates an functionally-equivalent protected app. DIVILAR is intended to be used at the last stage of app development. The resulting app can be released in the online app stores for users to download and install. As mentioned earlier, the DIVILAR architecture can accommodate different levels of sophistication in the implementation. In this section, we describe details of our prototype in the order of DIVILAR's four components: instruction selector, bytecode transformer, the interpreter, and apk packager.

Our prototype adopts a linear mapping for Dalvik bytecode's opcodes and operands (Section 3.2). To guarantee that each protected app is encoded differently, the instruction selector uses a random number to permute the opcode/operand mapping. This generates two rules: the transforming rule $X$ and its inverse. For example, $X =< 7503, 83, ... >$ specifies that opcode $01h$ will be converted to $7503h$, opcode $02h$ will be converted to $83h$, and so on. These rules are subsequently used to guide conversion between Dalvik and virtual instructions for the second and third components, bytecode transformer and the interpreter.

Bytecode transformer re-encodes the guest app in the virtual instruction set as specified by the transforming rule $X$. Our implementation utilizes the library of baksmali/smali [19], an open-source Dalvik disassembler/assembler. Specifically, the guest app is first parsed into a list of classes with their methods disassembled to the string format. Bytecode transformer then reassembles these classes into a `dex` file except that the opcode/operands have been adjusted according to rule $X$ (file `org/jf/dexlib/Code/Opcode.java` of the baksmali library is changed to write out the custom opcode and operands). For example, the bytecode $01h$ is first parsed into its readable format (`move` in this case). DIVILAR then converts `move` to the new opcode of $7503h$ when assembling the file. All classes of the guest app will be assembled together into a single `dex` file, and further packed into the final app as a data asset for the interpreter to execute.

The next step of the prototype generates a virtual instruction interpreter customized by rule $X'$. At run-time, the interpreter hooks into Dalvik VM in order to composite the virtual and Dalvik instruction execution. The interpreter is a native binary written in the C++ programming language. When loaded for execution, it first uses system call `mprotect` to make Dalvik VM writable so that it can be hooked. It then searches the code section of Dalvik VM for call sites of some specific functions, as determined by the hooking strategy. Each of these call sites will then be replaced by a call to the corresponding trampoline, as shown in Figure 4. A wide range of hooking schemes can be employed. In our prototype, we choose to extend the class initialization module, which is responsible for bytecode loading and method initialization (in file `vm/oo/Class.cpp` and `vm/oo/Object.cpp`). We add to it the pre-execution conversion of virtual instructions to Dalvik bytecode. To reduce the risk of the decoded methods being extracted from memory, our prototype flushes the Dalvik code cache randomly from time to time.

At the last step, APK packager assembles the guest app and the interpreter into a final app that can be publicly released later. It first parses the manifest file (`AndroidManifest.xml`) of the original app to get the list of requested permissions, entry points, and other resource files. It then synthesizes a wrapper (the shell) for the guest app. The shell inherits the requested permissions from the original app, and has a wrapper class for each entry point. It is also responsible for loading and executing of the guest app, a task performed in its subclass of `android.os.Application` because it is guaranteed to be executed before any other classes.

# 5. EVALUATION

DIVILAR aims to protect Android apps from being repackaged by applying VM-based protection. In this section, we first analyze DIVILAR's robustness against existing static and dynamic analysis, and countermeasures specific to VM-based protection in particular [11, 20, 57, 59]. We also evaluate the performance overhead introduced by DIVILAR.

## 5.1 General Analysis

DIVILAR re-encodes Android apps in a secret and diversified virtual instruction set. The guest app is wrapped as a data asset in the resulting app together with the interpreter for these virtual instructions. The interpreter hooks into the complicated Dalvik VM to composite execution of virtual and Dalvik instructions. By introducing this additional layer of indirection, existing tools that expect Dalvik bytecode will immediately be disabled. Attackers have to shift their focus from understanding Dalvik bytecode into reverse-engineering the interpreter first (presumably, the interpreter contains adequate information about the virtual instructions). In this sub-section, we will analyze DIVILAR's protection strength against app repackaging.

**Rooted in native code:** managed code is in general harder to obfuscate than native code because it contains rich semantics of the app. For example, intermediate languages like Java bytecode often use late binding, in which the target of a call is resolved at run-time *by name*. As such, call instructions in an intermediate language normally contain a reference to the method name as shown in Figure 1. Moreover, managed code has numerous run-time and compile-time checks to ensure program safety. This significantly limits the operations that can be performed by the program. Operations such as complex pointer arithmetic or direct memory access

```
unknown opcode encountered - 73. Treating as nop.

UNEXPECTED TOP-LEVEL EXCEPTION:
org.jf.dexlib.Util.ExceptionWithContext: 72
  at org.jf.dexlib.Util.ExceptionWithContext(ExceptionWithContext.java:54)
  ......
  at org.jf.dexlib.Section.readFrom(Section.java:143)
  at org.jf.dexlib.DexFile.<init>(DexFile.java:431)
  at org.jf.baksmali.main.main(main.java:280)
Caused by: java.lang.ArrayIndexOutOfBoundsException:72
  ......
  ... 6 more
Error occured at code address 140
code_item @0x784
```

Figure 8: Exception when applying *baksmali* on a transformed app

are almost always disallowed. Native code instead runs directly on the physical machine and has few restrictions in operations and program structures. Native code thus has more freedom in the support of obfuscation. Particularly, in native code, code and data can be mixed together; instructions can directly access any memory mapped in the program; a program can also mutate itself with the self-modifying code. Obfuscation in native code can leverage these features to make it more reliable. For example, Giffin et al. [23] propose to strengthen software self-checksumming using self-modifying code; Jacob et al. [37] leverage overlapped instructions in the x86 architecture to implement a stronger tamper-proof solution. These technologies make obfuscated native binaries more difficult to reverse-engineer, usually requiring laborious human efforts.

DIVILAR has its robustness rooted in the native code. Instead of implementing the interpreter in Dalvik bytecode, the interpreter is a native executable. This not only reduces the overhead, but also allows advanced obfuscation technologies [23, 37] to be applied, making the interpreter hard to reverse-engineer.

**Varied virtual instruction selections:** architecturally, DIVILAR can accommodate a wide range of virtual instruction selections as long as it can be reversed. The instruction selector works as a plugin of DIVILAR to support different encoding schemes, such as fixed-length v.s variable-length instructions, stack-based v.s register-based instructions, or even overlapped instructions [37]. Moreover, DIVILAR randomizes the virtual instruction set for each individual app. Breaking one protected app thus does not necessarily lead to compromise of other protected apps. In our prototype, we choose a linear mapping for virtual instructions. The combination of such a mapping is sufficiently large so that the brutal-force attack (to try every possible combinations) is simply infeasible. Meanwhile, our prototype can confuse a parser of Dalvik bytecode with shifted instruction boundaries. This makes frequency analysis ineffective because it relies on a parser to decode the app first (Section 5.4). Therefore, even with this simplified virtual instruction selection, our prototype provides a strong protection against existing countermeasures.

**Diverse interpreter design:** to reduce overhead, DIVILAR hooks into Dalvik VM in order to composite virtual and Dalvik instruction execution. A wide range of interpreter design (hence hooking schemes) can be supported. For example, it is possible to even write a standalone interpreter and completely bypass Dalvik VM. Different interpreter design needs to modify different components of Dalvik VM (Figure 5). Our prototype implements a pre-execute decode scheme in which virtual instructions are reverted back to Dalvik bytecode before execution. An adversary monitoring the memory of the app might be able to extract the recovered bytecode. To reduce the time window of possible exposure, our prototype will randomly flush the code cache. To further reduce the window, we may hook into code that execute instructions in a finer granularity, such as basic block level or instruction level. To protect the interpreter from reverse-engineering, the interpreter should be obfuscated as mentioned before. A large number of existing obfuscation schemes for native binaries can be applied.

In addition, with the interpreter implemented in native code, performance overhead will not be a concern for DIVILAR as a variety of optimization techniques can be applied [1, 3, 9, 67], such as ahead-of-time compilation or adaptive optimization.

**Compatible with other schemes:** DIVILAR encodes the guest app in a diversified virtual instruction set. This kind of transformation is compatible with a wide variety of other protection mechanisms. For example, AppInk [72] embeds watermarks into an app to identify the source of an app. DIVILAR can be applied to the app after watermarks are embedded. Obfuscation techniques can also be applied before or after DIVILAR. For example, Java based obfuscation can be attempted first, and then DIVILAR is applied, or DIVILAR is used first and native code obfuscation can be used to obfuscate the interpreter in the resulting app. Moreover, DIVILAR aims to provide a strong preventive mechanism for apps. It does not provide services such as integrity check in itself. These services can also be combined with DIVILAR. For example, self-checksumming is often used to ensure program integrity. Due to various constrains of managed code, self-checksumming is challenging to implement in managed code. With DIVILAR, it can be implemented in the native code and be used at various stages of Dalvik VM execution. This would make self-checksumming for Android not only feasible but also more reliable against adversaries.

In the following of this section, we analyze DIVILAR's robustness against various specific static and dynamic analysis, including experiments with popular tools frequently used in Android app repackaging.

## 5.2 Static Analysis

To repackage an app, attackers often apply some form of static analysis to understand how the app is organized and where to make necessary modifications. All these tools include some form of disassembler to parse the app into machine- or human-readable format, such as the mnemonic representation of Dalvik bytecode. Baksmali [19] probably is the most popular disassembler for Android apps. Its source code has been embedded in many other Android analysis/reverse-engineering tools [24, 55]. Meanwhile, many static analysis tools for Android, such as Dare [51] and smali2java [24], re-target Android apps to Java bytecode (Java bytecode is stack-based while Dalvik bytecode is register-based) in order to leverage a rich set of existing Java analysis tools [66]. In this subsection, we will evaluate DIVILAR's resilience against two representative static analysis tools: baksmali [19] and Dare [51]. Specifically, we encode seven Android apps into virtual instructions, and apply these two tools on the resulting apps to observe their behaviors.

Baksmali is an open-source Android disassembler. Its companion project, Smali, is a corresponding assembler. Baksmali and Smali are frequently used to manually repackage apps [39]. Baksmali provides a library that can reliably parse Dalvik execute files [56] and further disassemble them into the mnemonic format. The source code of Baksmali has since been embedded in many other tools (including DIVILAR). For all these seven apps, Baksmali reports an error message of "unknown opcode encountered - nn. Treating as nop" and throws an unexpected top-level exception. Figure 8 shows an encountered exception when we apply Baksmali to one of the guest app. Baksmali and other Android analysis tools expect Dalvik bytecode as inputs. By re-encoding Android apps in virtual instructions, DIVILAR has changed the semantic of opcode

```
W/dalvikvm( 6997): Can't decode unexpected format 0 (op=115)
GLITCH: zero-width instruction at idx=0x0000
W/dalvikvm( 6997): Can't decode unexpected format 0 (op=115)
GLITCH: zero-width instruction at idx=0x0000
```

Figure 9: Error message when applying Dare on a transformed app

seen by these tools and shifted instruction boundaries around. However, these tools still interpreter the guest app according to the Dalvik bytecode format. This usually leads to unknown opcodes, as shown in Figure 8.

To leverage a wide variety of existing static analysis tools for Java bytecode, some analysis tools opt to re-target Android apps into Java bytecode. Dare [51] is one of the most effective tools for this purpose, which is reported to work correctly on more than 99.99% of 262,110 Dalvik classes. Since the very first step of Dare is also to parse the `dex` file, it fails with a similar symptom as Baksmali. Figure 9 shows the error message reported by Dare on the same guest app.

The above experiments show that DIVILAR can immediately render these existing Android disassemblers and analysis tools ineffective because they all depend on parsing the well-formed Dalvik bytecode. These tools include decompiler, slicer, control flow analyzer, data flow and data dependency analyzer [19, 51, 55, 66]. For these tools to function correctly, it is necessary to first recover the original Dalvik bytecode by reverse-engineering the interpreter.

## 5.3 Dynamic Analysis

Dynamic analysis executes the app under investigation in a monitored environment, such as a virtual machine or an emulator, to observe its behaviors. Dynamic analysis has been widely used in malware analysis and program development. In dynamic analysis, the monitor can be placed either inside or out of the run-time environment of the target app. The former has the advantage of direct access to system/app states, but its result may be disturbed or even tampered with by the target app. While the latter gains the tamper-resistance, but faces the "semantic gap" challenge [10, 38], in which the monitor has to *deduce* app states given only externally observable data such as a memory dump [38, 69]. It is challenging to apply this out-of-box approach to analyze Android apps because we need to reconstruct both operating system states and Dalvik VM states due to the two layers of virtualization [69]. DIVILAR makes the problem even more challenging by adding another layer of indirection: the monitor needs to infer states of the operating system, the virtual instruction interpreter, and Dalvik VM.

Our current prototype adopts a pre-execution decoding scheme. An adversary who *continuously* monitors the app's memory might be able to extract the bytecode of the original app. Such powerful attacks can be (partially) addressed by either limiting the amount of restored bytecode (e.g., from method to basic block) and shortening the time window of exposure, or by detecting whether the app is run in a virtual environment (the Android emulator) and refusing to run if so.

## 5.4 Virtualization Specific Analysis

Virtualization-based protection/obfuscation has been long employed by malware to hide its malicious behavior. A number of countermeasures have been proposed to detect or prevent such malware. For example, some systems aim to reverse-engineer the VM and use this information to infer semantics of individual virtual instructions [57, 59]. Coogan et al. propose an inside-out approach [11], which intends to identify instructions that may affect the observable behavior of the wrapped code (i.e., system calls) . Ghosh et al. propose to replace the protecting VM with an attacking VM in order to render the application amicable to analysis and modification [20]. Frequency analysis [68] is often used to break classical ciphers by studying the frequency of letters or groups of letters to disclose the mapping between plaintext and ciphertext. It may potentially be used to infer the mapping between Dalvik bytecode and virtual instructions. In the rest of this sub-section, we discusses DIVILAR's robustness against these virtualization-specific attacks.

**Reverse-engineering VM:** a few approaches are proposed to automatically reverse-engineer the virtual machine, and then use the recovered semantics of virtual instruction to understand the original app [57, 59]. Automatic reverse-engineering a virtual machine, in general, is an unsolvable problem. As such, existing approaches limit the scope by, for example, assuming that the VM works in the loop of fetch-decode-dispatch. In case of DIVILAR, these assumptions do not hold, thus making these approaches ineffective: Dalvik VM has a complex structure with about 104,000 lines of C++ code and 99,000 lines of assembly code. Moreover, Dalvik VM supports many different execution modes, including portable, fast, and JIT modes. These execution modes can also be extended by DIVILAR in many possible ways as discussed earlier.

**Inside-out approach:** a recent work proposes an inside-out approach [11] that complements the VM reverse-engineering approach. Instead of recovering all instructions of the wrapped app, this approach aims at identifying instructions that interact with the operating system and instructions that may affect these instructions. It is assumed that these instructions together approximate the behavior of the original code, while other instructions are considered to be uninteresting. This approach can help malware analysts to gain better understanding of the malware under examination without being overwhelmed by details of all the instructions. However, partial understanding of virtual instructions is not enough to make meaningful modifications to the guest app and repackage it. In particular, any executable additions to the guest app need to be encoded in the same virtual instructions as the app. This is not possible without a complete understanding of the virtual instruction set first.

**VM replacement attack:** VM replacement attack is proposed to subvert virtualization-protected apps by replacing the protecting VM with an attacking VM, within which the app's execution can be monitored and analyzed. However, this attack is not effective against systems where virtual machine is sufficiently anchored to the execution environment [20]. DIVILAR's in-app hooking mechanism allows it to deeply hook into Dalvik VM and merge the execution of virtual and Dalvik instructions. This tight coupling with the underlying execution environment makes VM replacement attacks ineffective against DIVILAR. In addition, DIVILAR can identify whether the underlying VM has been modified or whether an attacking VM (specifically, its code introspection framework) has been loaded into the app's address space. Essentially, VM replacement attack in this case becomes an in-VM monitor and thus subjects to tampering [38].

**Frequency analysis** Frequency analysis is often used to break classic cipher where a plaintext letter is mapped to one ciphertext letter. By studying the frequency of letters or groups of letters, the analyst can deduce the mappings between the plaintext letters and ciphertext letters [68]. Our prototype of DIVILAR also uses a mapping table to convert Dalvik bytecode into virtual instructions. However, our system is not susceptible to frequency analysis. Specifically, by simply replacing one letter with another, a counting tool can easily count the frequency of plain-text and cipher-text in a classic cipher. However, it does not know how

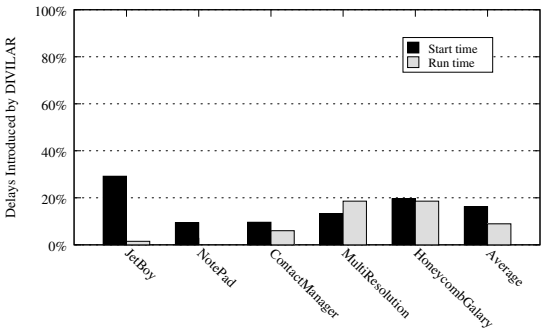

Figure 10: Performance overhead introduced by DIVILAR

to count the instruction frequency for virtual instructions without the knowledge of the virtual instruction set first. In fact, such tools would require a parser to decode instructions, which has been shown to be ineffective against DIVILAR (Section 5.2).

## 5.5 Performance Evaluation

In this section we evaluate the performance overhead introduced by DIVILAR. We measure both start-time and run-time overheads, two overheads that matter most to end users. We choose five sample apps from the Android SDK as our target apps: jet boy, notepad, contact manager, multi-resolution, and honeycomb gallery. We select these apps because their source code are freely available so that we can insert probing points into the apps to measure time required to execute certain operations.

We first measure how DIVILAR affects the app start up, from launching the app to the appearance of the main activity on-screen. To prepare an app for execution, the Android framework performs a series of steps such as loading Dalvik VM. DIVILAR adds to the start-time in order to prepare execution of virtual instructions. Specifically, the guest app is wrapped in a shell app. Android launches the shell first which in turn loads the guest app. To measure the start time, we use the `logcat` command [34] of Android Debug Bridge [33]. The verbose logging level of `logcat` is used so that it shows the elapsed time from app launching to the appearance of the first app view. All the experiments were executed three times with the average reported. Figure 10 shows the percentage of delays introduced by DIVILAR. They range from 9.4% to 29.2% with an average of 16.2%. Considering that all these test apps start in less than half-second even under DIVILAR's protection, the extra delay to start time caused by DIVILAR likely will not be perceived by end users at all.

Next, we measure the overhead of DIVILAR to a running app by adding probe points to log its execution time. Specifically, we first examine their source code to select code ranges where no user interaction is required (so as to exclude the impact of uncontrollable user interaction), and manually insert a call to `android.util.Log` before and after each of these selected code ranges. We then rebuild these apps and use DIVILAR to generated their corresponding protected apps. Figure 10 reports the percentage of overhead caused by DIVILAR to parts of these running apps. They range from 0% to 18.6% with an average of 8.9%. DIVILAR has fixed overhead for each method, i.e., the time to translate the method from virtual instructions to Dalvik bytecode. The run-time of a method, on the other hand, is determined by dynamic program states such as branch conditions and loops. For example, a method could execute only a failed branch in one run and execute hundreds of times in a loop in another run. Overall, our interpreter design is efficient and end users will not perceive any delays or stuttering.

# 6. DISCUSSION

In this section, we discuss implications and possible improvements of DIVILAR and , in particular, our prototype.

In essence, DIVILAR is an obfuscation technology used to protect the Android app from repackaging. Although the technology itself is neutral, obfuscation has long been used by malware authors to evade detection by anti-malware software. Many security researchers and practitioners share the opinion that benign software (Android apps in this case) should not use obfuscation so that its behaviors can be easily analyzed or even formally verified. Unfortunately, it is not an option for anti-repackaging. Most apps are repackaged and distributed in third-party online app stores [74]. These stores often lack regulations against app piracy (some even benefit from it), and many are located in other countries. It is thus difficult for developers to resolve the issue through jurisdiction. Instead, the developers have to enforce their apps for self-defense. App repackagers would try to defeat or bypass these defense mechanisms. It becomes an arms race between developers and app repackagers. For example, Google suggests to use string and other obfuscation techniques to prevent its server-side license verification from being disabled or bypassed [32]. In addition, VM-based protection has been previously proposed to protect applications [21, 46, 61, 62, 65].

Although not an architectural limitation, our prototype has a relatively simple design: a linear mapping of opcode/operand is used to select virtual instructions, and the interpreter implements a pre-execution decoding mode. Our initial experiments and analysis show that this design works reliably against existing static and dynamic countermeasures, including these specific to virtualization-base protection. Nevertheless, the design can be enhanced by additional techniques to thwart analysis. In the future, we will study ways to fortify our current design, particularly, how to formally measure the strength of obfuscation technologies and how to maximize its strength given certain performance overhead.

In its current operation model, developers use DIVILAR to process their apps before releasing them to the online app store. All installations of the app thus are the same across all its users (although different apps will have a different virtual instruction set). In the future, we will study methods to securely perform per-installation obfuscation, for example, to re-encode the app at its first run. In this way, each installation will have a different encoding, possibly tied to the user's device. This would provide a stronger protection because every installation of the app is different.

DIVILAR hooks into Dalvik VM to composite execution of virtual and Dalvik instructions. Like other such software [30], compatibility with different revisions of the host software is a concern. Dalvik VM, the host of DIVILAR , has become relatively stable since Honeycomb (Android 3.0). Our prototype further alleviates the problem by avoiding hard-coding any specific hooking points and locating them through pattern matching instead. Under the unlike circumvents that Dalvik VM is changed significantly and frequently, we could implement a self-contained execution engine (possibly based on the source code of Dalvik VM) and completely bypass Dalvik VM. This will improve the compatibility but may lose the benefit of updates made by Google.

Lastly, DIVILAR implements the VM-based protection for Android apps. From another point of view, VM-base protection can be considered as a generation of instruction set randomization [5]. For example, in addition to randomizing instruction set, VM-base protection can provide a randomized view of memory. Our current prototype focuses on the randomization of Dalvik bytecode. In the

future, we will experiment randomization of other components in VM-based protection as well.

## 7. RELATED WORK

In this section, we present related work in the following category: diversification for security, virtualization-based protection, and Android app protection.

**Diversification for security:** the first category of related work includes researches that apply the principle of diversity to enhance security. For example, address space layout randomization is a defense mechanism to prevent low-level memory-base exploits such as buffer overflow [58]. It randomizes locations of a program's code or data so that an exploit cannot (easily) locate the vulnerability. It has since been ubiquitously deployed in Linux [63], Windows [49], Android [54], and Mac OS X [13]. Instruction-set randomization is a closely related work. It diversifies a program's instruction set to render injected attacking code ineffective (because they are encoded in a different instruction set) [5]. DIVILAR and other virtualization-based protection systems [2] adopt a similar idea to obfuscate a program in order to protect it from reverse-engineering. As software diversity becomes more popular, automated tools have been proposed to improve its performance. For example, Franz et al. use profile-guided optimization to reduce the overhead of software diversity [18]. This line of researches is orthogonal to our study and thus can be leveraged by DIVILAR to further reduce the overhead.

**Virtualization-base protection:** the second category of related work consists of systems that employ virtualization-based protection. Virtualization has long been applied to protect native code from reverse-engineering [21, 61, 62, 65]. Malware authors are also found to use virtualization to thwart detection or analysis. Meanwhile, countermeasures have been proposed to detect or reverse-engineer virtualization-based malware. For example, Rotalume can automatically reverse-engineer virtualization-based malware that employs a virtual machine with a fetch-decode-execute model [59]. To allow malware analyst to focus on the critical behaviors of malware, an inside-out approach is proposed to identify instructions that interact with the underlying operating system and other instructions that may affect those instructions [11]. Moreover, Ghosh et al. propose to use an attacking VM to replace the protecting VM to subject the protected software to analysis. It assumes that the protecting VM is not tightly bound to the execution environment [20]. DIVILAR leverages virtualization to protect Android apps from repackaging. An adversary of DIVILAR could apply these techniques against DIVILAR. In section 5.4, we have analyzed DIVILAR's resilience against these approaches. Virtualization-based protection has also been applied to protect managed code as well (specifically MSIL) [2]. Their interpreters are implemented in managed code, causing performance overhead as high as 50 times to 3500 times. DIVILAR, also targeting managed code, achieves much better performance (Section 5) by combining virtual instruction and Dalvik bytecode execution.

**Android app security:** the last category of related work includes a long stream of research in Android app security, particularly these related to app repackaging. For example, a few systems aim to detect repackaged apps by measuring similarity among a large number of apps [12, 29, 53, 73, 74]. DIVILAR instead enables self-defense of Android apps using virtualize-based protection. It intends to prevent app repackaging at the first place. Recently, Google and other mobile platform providers introduced different flavors of server-side license verification solutions [32, 52]. These solutions and DIVILAR complement each other: DIVILAR can prevent these mechanisms from being circumvented, while they provide license verification. Similarly, Android app watermark [72] and software integrity check [23] can also be combined with DIVILAR to provide a more complete protection. There are also obfuscation technologies for Android [36, 41]. However, they tend to keep high-level semantic information of the app intact. Also, a new app installation mechanism is proposed to perform pre-installation verification and app encryption to protect app from repackaging [71]. However, it requires the app installation mechanism, a fundamental process of the Android platform, to be changed, hampering its adoption. DIVILAR has better compatibility with existing Android systems.

There are also a series of research on other aspects of Android app security. For example, TaintDroid [15] applies information flow tracking to monitor privacy leak in Android apps. Apex [50] and MockDroid [7] intend to address the privacy leak problem caused by invasive Android apps by modifying the Android framework. DroidRanger [76] and RiskRanker [27] propose different approaches to detect malicious Android apps in (third-party) online app stores. Woodpecker [26] and CHEX [47] try to discover capability leak and component hijacking vulnerabilities in Android apps. DIVILAR differs from them by introducing the self-defense capability into Android apps to protect themselves from being repackaged.

## 8. CONCLUSION

App repackaging remains a serious threat to the Android ecosystem and the emerging mobile economy model. To thwart repackaging, we adopt virtualization-based protection to enable the app's self-defense. To this end, we design an effective solution called DIVILAR which hooks into Dalvik VM to efficiently composite the virtual instruction and Dalvik bytecode execution. Our evaluation demonstrates that DIVILAR is robust against existing static and dynamic analysis including these specific to virtualization-based protection or obfuscation. Our prototype incurs a small performance overhead that likely is not perceivable to end users. In conclusion, DIVILAR is an effective and promising technology to enable app self-defense and prevent app repackaging in particular.

## 9. REFERENCES


[1] J. Absar and D. Shekhar. Eliminating Partially-Redundant Array-Bounds Check in the Android Dalvik JIT Compiler. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 121–128, New York, NY, USA, 2011. ACM.

[2] B. Anckaert, M. Jakubowski, and R. Venkatesan. Proteus: Virtualization for Diversified Tamper-Resistance. In *Proceedings of the Sixth ACM Workshop on Digital Rights Management*, pages 47–57, 2006.

[3] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. In *Proceedings of the IEEE, 93(2), 2005. Special Issue on Program Generation, Optimization, and Adaptation*, 2004.

[4] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 177–192, Berkeley, CA, USA, 2008. USENIX Association.

[5] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[6] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[7] A. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, 2011.

[8] A. D. Blog. Android at Google I/O 2013: Keynote Wrapup. http://android-developers.blogspot.com/2013/05/android-at-google-io-2013-keynote-wrapup.html.

[9] S. Brunthaler. Virtual-Machine Abstraction and Optimization Techniques. *Electron. Notes Theor. Comput. Sci.*, 253(5):3–14, Dec. 2009.

[10] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, HOTOS '01, pages 133–, Washington, DC, USA, 2001. IEEE Computer Society.

[11] K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-Obfuscated Software: a Semantics-Based Approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.

[12] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *17th European Symposium on Research in Computer Security*, ESORICS 2012, September 2012.

[13] O. X. Developer. Mac OS X Lion Security Enhancements – Improved ASLR. http://osxdeveloper.wordpress.com/2011/07/28/mac-os-x-lion-security-enhancements-improved-aslr/.

[14] A. Developers. Android NDK. http://developer.android.com/tools/sdk/ndk/index.html. Online; accessed at June 30, 2013.

[15] W. Enck, P. Gilbert, B.-g. 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 the 9th USENIX Symposium on Operating Systems Design and Implementation*, USENIX OSDI '11, 2011.

[16] T. A. Entrepreneur. Android App Piracy Is A Huge Problem! http://theappentrepreneur.com/android-app-piracy-is-huge-problem.

[17] H. Fang, Y. Wu, S. Wang, and Y. Huang. Multi-stage Binary Code Obfuscation Using Improved Virtual Machine. In X. Lai, J. Zhou, and H. Li, editors, *Information Security*, volume 7001 of *Lecture Notes in Computer Science*, pages 168–181. Springer Berlin Heidelberg, 2011.

[18] M. Franz, S. Brunthaler, P. Larsen, A. Homescu, and S. Neisius. Profile-Guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[19] J. Freke. Smali - An Assembler/Disassembler for Android's dex Format. http://code.google.com/p/smali/. Online; accessed at Jun 30, 2013.

[20] S. Ghosh, J. Hiser, and J. W. Davidson. Replacement Attacks against VM-Protected Applications. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 203–214, New York, NY, USA, 2012. ACM.

[21] S. Ghosh, J. D. Hiser, and J. W. Davidson. A Secure and Robust Approach to Software Tamper Resistance. In *Proceedings of the 12th international conference on Information hiding*, IH'10, pages 33–47, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proceedings of 11th International Conference on Mobile Systems, Applications and Services*, 2013.

[23] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening Software Self-checksumming via Self-Modifying Code. In *The 21st Annual Computer Security Applications Conference*, pages 23–32. IEEE Computer Society, 2005.

[24] GitHub. Smali 2 Java Conversion Tool. https://github.com/demitsuri/smali2java. Online; accessed at Jun 30, 2013.

[25] R. Goodwin. WWDC 2013: Apple's App Store Numbers are Insane - 50 Billion App Downloads, $10 Billion Paid to Devs. http://www.knowyourmobile.com/apple/20409/wwdc-2013-apples-app-store-numbers-are-insane-50-billion-app-downloads.

[26] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[27] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[28] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 98–115, Berlin, Heidelberg, 2008. Springer-Verlag.

[29] S. Hanna, L. Huang, S. Li, C. Chen, and D. Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA 2012, July 2012.

[30] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[31] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, WINSYM'99, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[32] G. Inc. Android Application Licensing. http://developer.android.com/guide/google/play/licensing/index.html. Online; accessed at Jun 30, 2013.

[33] G. Inc. Android Debug Bridge. http://developer.android.com/tools/help/adb.html. Online; accessed at Jun 30, 2013.

[34] G. Inc. Android Logcat. http://developer.android.com/tools/help/logcat.html. Online; accessed at Jun 30, 2013.

[35] G. Inc. The AndroidManifest.xml File. http://developer.android.com/guide/topics/manifest/manifest-intro.html. Online; accessed at Jun 30, 2013.

[36] S. Inc. A Specialized Optimizer and Obfuscator for Android. http://www.saikoa.com/dexguard. Online; accessed at Jun 30, 2013.

[37] M. Jacob, M. H. Jakubowski, and R. Venkatesan. Towards Integral Binary Execution: Implementing Oblivious Hashing Using Overlapped Instruction Encodings. In *Proceedings of the 9th workshop on Multimedia & security*, pages 129–140, New York, NY, USA, 2007. ACM.

[38] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 128–138, New York, NY, USA, 2007. ACM.

[39] S. Joo and C. Hwang. Mobile Banking Vulnerability: Android Repackaging Threat. Virus Bulletin, May 2012.

[40] A. Krall. Efficient javavm just-in-time compilation. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 205–212, 1998.

[41] E. Lafortune. ProGuard. http://proguard.sourceforge.net/. Online; accessed at Jun 30, 2013.

[42] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[43] Lohan+. AntiLVL - Android License Verification Library Subversion. http://androidcracking.blogspot.com/p/antilvl_01.html. Online; accessed at Jun 30, 2013.

[44] Lohan+. Cracking Amazon DRM. http://androidcracking.blogspot.com/2011/04/cracking-amazon-drm.html. Online; accessed at Jun 30, 2013.

[45] Lohan+. Cracking Verizon's V Cast Apps DRM. http://androidcracking.blogspot.com/2011/06/cracking-verizons-v-cast-apps-drm.html. Online; accessed at Jun 30, 2013.

[46] S. T. Ltd. StarForce - Reliable Copy Protection for Software, Content, CD/DVD/USB. http://www.star-force.com/.

[47] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 229–240, New York, NY, USA, 2012. ACM.

[48] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441, 2007.

[49] MSDN. Address Space Layout Randomization for Windows Vista. `http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/608315.aspx`. Online; accessed at Jun 24th, 2013.

[50] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, 2010.

[51] D. Octeau, S. Jha, and P. McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, New York, NY, USA, 2012. ACM.

[52] A. Police. [Updated: Amazon Provides Clarifications] Amazon App Store's DRM To Be More Restrictive Than Google's? `http://www.androidpolice.com/2011/03/07/amazon-app-stores-drm-to-be-more-restrictive-than-googles/`. Online; accessed at Jun 30, 2013.

[53] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques. In *Proceedings of the 4th International Conference on Engineering Secure Software and Systems*, ESSoS'12, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[54] A. O. S. Project. Android Security Overview. `http://source.android.com/tech/security/index.html#memory-management-security-enhancements`. Online; accessed at Jun 24th, 2013.

[55] G. C. Project. Android-Apktool - Tool for Reengineering Android apk Files. `http://code.google.com/p/android-apktool/`. Online; accessed at Jun 30, 2013.

[56] T. A. O. S. Project. Bytecode for the Dalvik VM. `http://source.android.com/tech/dalvik/dalvik-bytecode.html`. Online; accessed at Jun. 23, 2013.

[57] R. Rolles. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[58] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[59] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109, 2009.

[60] Slashgear. 95% Android Game Piracy Experience Highlights App Theft Challenge. `http://www.slashgear.com/95-android-game-piracy-experience-highlights-app-theft-challenge-15282064/`.

[61] A. Software. ASProtect - Software Protection Tools for Software Developers. `http://www.aspack.com/asprotect.html`.

[62] V. Software. VMProtect Software Protection. `http://vmpsoft.com/`.

[63] P. Team. Pax Document for Linux ASLR. `http://pax.grsecurity.net/docs/aslr.txt`. Online; accessed at Jun 24th, 2013.

[64] A. Technologies. State of Security in the App Economy: Mobile Apps Under Attack. `http://www.arxan.com/assets/1/7/state-of-security-app-economy.pdf`, 2012.

[65] O. Technologies. Code Virtualizer Overview. `http://www.oreans.com/codevirtualizer.php`.

[66] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[67] C.-S. Wang, G. Perez, Y.-C. Chung, W.-C. Hsu, W.-K. Shih, and H.-R. Hsu. A Method-based Ahead-of-Time Compiler for Android Applications. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 15–24, New York, NY, USA, 2011. ACM.

[68] WikiPedia. Frequency analysis. `http://en.wikipedia.org/wiki/Frequency_analysis`. Online; accessed at Jun 24th, 2013.

[69] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[70] M. Zheng, P. P. Lee, and J. C. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems . In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[71] M. Zheng, M. Sun, and J. Lui. Reboot the Android Third-party Markets:The Way to Defend Against Cracking and Malware Repackaging on Android Application. In *Internet Security Forum*, 2012.

[72] W. Zhou, X. Zhang, and X. Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ASIA CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[73] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, Scalable Detection of Piggybacked Mobile Applications. In *Proceedings of the 3nd ACM Conference on Data and Application Security and Privacy*, CODASPY '13, February 2013.

[74] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY '12, February 2012.

[75] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, Oakland 2012, May 2012.

[76] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.