Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelorthesis

ArtHook
Callee-side method hook injection on the new Android runtime ART

submitted by
Marvin Wißfeld
on **June 1, 2015**

Advisor:
Philipp von Styp-Rekowsky

Reviewers:
Prof. Michael Backes
Christian Hammer

Declarations

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

(Datum/Date) (Unterschrift/Signature)

Acknowledgement

I would like to thank my advisor Phillip von Styp-Rekowsky for many hours of productive discussion. His feedback has helped me a lot to come up with the results of this thesis.

Thanks goes as well to Prof. Michael Backes and Christian Hammer for reviewing my thesis.

I would also like to thank everyone who bugged me to continue working on my thesis or helped in any other way to finish this thesis on time.

Abstract

Hooking methods on Android's Dalvik runtime has become a common technique for various use cases, including security and privacy extensions. Android 5.0 introduced the new ahead-of-time compiling android runtime ART. The new runtime can not be used with current implementations that relied on the internal structure of Dalvik's just-in-time compiler.

We thus present ArtHook. ArtHook is an approach to hook methods on ART. It will allow to hook the majority of method calls without the need to manipulate the Android operating system and with low performance overhead. We achieve this by rewriting the machine code of the called method in memory.

Contents

Chapter 1

Introduction

The Android platform is the most popular operating system on consumer-devices worldwide. In 2014, roughly the half of all computers, notebooks, tablet and smart phones sold, shipped with Android installed [1], a stunning total of 1.1 billion devices. Private data of millions of users is stored using the Android operating system, but the number of privacy and security mechanisms that are part of Android is outstandingly low. Thus, researchers developed different security and privacy extensions for Android. Some of them, like MockDroid [2], require operating system changes. As for most end-users it is not possible to change the operating system installed on their phones, this solution did not receive wide-spread support.

The other approach to add features to the operating system is to use method hooking. Method hooking is a technique used to intercept the call of a certain method at runtime to change the behavior of the calling application. This is usually done for debugging or to extend the features of an existing library method. AppGuard [3] uses method hooking to embed additional privacy and security features into applications.

ART is a new Java runtime targeting the Android platform. A first technical preview was published with Android 4.4 in 2013 as an alternative to the Dalvik runtime used by then [4]. It already showed the core improvements that were intended to be included with the final release, one of them being the ahead-of-time compilation. Most current Java runtimes use pure byte code interpretation, a just-in-time compiler[1] or a combination of both. The ahead-of-time compiler used in ART creates machine code from the application byte code before it is executed. For Android applications, this step is done during installation, but it is also possible to compile directly before the execution or to provide pre-compiled machine code binaries. With the release of Android 5.0 in 2014, ART became the new default runtime and Dalvik was removed [5]. It was also extended

[1] A system that compiles source code or intermediate byte code into machine code in the moment of its first or repeated usage

to heavily optimize the resulting machine code, leading to performance improvements by up to 2.5 times compared to Dalvik in benchmarks [6].

However, the introduction of ART as default runtime for Android 5.0 makes existing method hooking techniques designed for Dalvik incompatible with the latest Android versions.

1.1 Contribution

We describe a system to hook methods on the ART runtime introduced and published with Android 5.0. This technique does not require any modifications to the operating system and uses callee-site in-memory rewriting to inject the hook. For better understanding, we will also give a comprehensive description of the ART runtime.

To the best of our knowledge, no method hooking system for ART, which does not require modifications of the ART runtime and the operating system, has been described or developed before.

1.2 Outline

In chapter 2, we present the ART Android runtime including several details that will become important for the hooking process.

Chapter 3 covers existing method hooking techniques for different platforms and operating systems and outlines why the related work is not portable to ART. Finally it concludes an idea of how method hooking can be done on ART.

In chapter 4, we describe in detail what our approach of method hooking on ART is and how we implemented it. We also show an example of how to use the ArtHook library and discuss different APIs provided by it.

In chapter 5, we analyze how our implementation behaves on a device. We therefore do a micro benchmark and use a publicly accessible benchmark suite.

Chapter 6 gives a glimpse on another hooking system, which targets the Android ART platform, but uses a completely different approach.

Finally, chapter 7 outlines, which problems are not solved by our hooking approach and ideas how it might be possible to come around them.

Chapter 2

ART Android runtime

The ART runtime is different from previous Java virtual machines in several ways. Thus, in this chapter, we analyze the reasons that resulted in the development of this new runtime and how the process of ahead-of-time compilation works. We well also describe the file format used internally and how method calling is done.

2.1 Introducing ART

The former Java runtime used on Android, Dalvik was originally developed as a full-featured Java runtime for resource constrained devices. This was needed, as by then, smart phones came with a considerably less powerful single-core CPU and down to a 10th of the memory of current smart phones. However, as the market shifted, Dalvik was extended on all edges by adding better support for multi-core CPUs or just-in-time compilation.

Later the developers of Android decided, that their Java runtime should be rethought completely, leading to the development of ART and finally the deprecation of Dalvik. The new Android runtime was build with modern processors, multi-tasking and efficient garbage collection in mind. This results in less CPU usage, longer battery times and performance increases.

2.2 Ahead-of-time compilation

The main feature introduced with ART that distinguishes it as a completely new runtime is the ahead-of-time compilation. During installation an application is compiled to "quick compiled" code, which consists of native processor instructions [5]. This results in performance improvements, as during runtime, no code has to be compiled or interpreted.

Due to the fact that the processor is the main power sink in modern smart phones, performance improvements also result in less power consumption and thus longer battery life. Furthermore there is also measurable less memory pressure[1] due to not using just-in-time compilation. The main drawback of this approach is that the compilation increases application installation time and the compiled code requires more disk space. For this reason, it is possible to compile ART in a mode for devices with limited storage that uses the ahead-of-time compiler only for the system framework (boot classpath). Although applications are only interpreted, this is still a performance improvement compared to Dalvik, as most execution time actually passes in the system framework.

The handling of the boot classpath is optimized to reduce start-up time. This is done by compiling the classes in boot classpath and pre-initializing them. The pre-intializing process includes the creation of constants and the configuration of memory entry points. The result is stored in a file called `boot.art` that is mapped into memory during boot. As pre-initialized classes need more disk space, this does not happen for all classes, but only for those used frequently[2].

The Dalvik executable (DEX) is a special file format that contains machine instructions for the Dalvik virtual machine. ART still handles DEX (Dalvik executable) files. Android applications do not need to be altered to run on ART and thus the first compilation steps of Java source code are the same as before: Java source code is compiled to JVM byte code using the java compiler `javac`, which is then translated to Dalvik byte code using `dx`. These processor intensive and target platform independent tasks happen during application creation, the resulting DEX file is released to users as part of the Android package (APK) file. When the Android package is installed on a system running ART, the ART compiler `dex2oat` compiles Dalvik byte code from DEX files into native, platform-dependent machine code.

2.3 OAT file format

OAT files are special ELF[3] files storing the compiled code for the ART runtime, the original Dalvik byte code, string and class tables as well as ART specific meta data. Although the ELF format is used, tools usually used with this format will not be able to handle OAT files, as ELF sections are not used as usual.

[1] Modern operating system uses virtual memory that may not only represent volatile memory, but also file system entries. As these do not inevitably reside in physical memory, the operating system can assign more virtual memory than physical memory exists and no free physical memory does not necessarily mean that no new memory can be allocated. The term memory pressure is used instead of memory usage to address this.

[2] A list of these classes is provided during the Android build process and is stored inside the resulting image at `/system/etc/preloaded-classes`.

[3] ELF: Executable and Linking Format, a common standard file format storing compiled code [7]

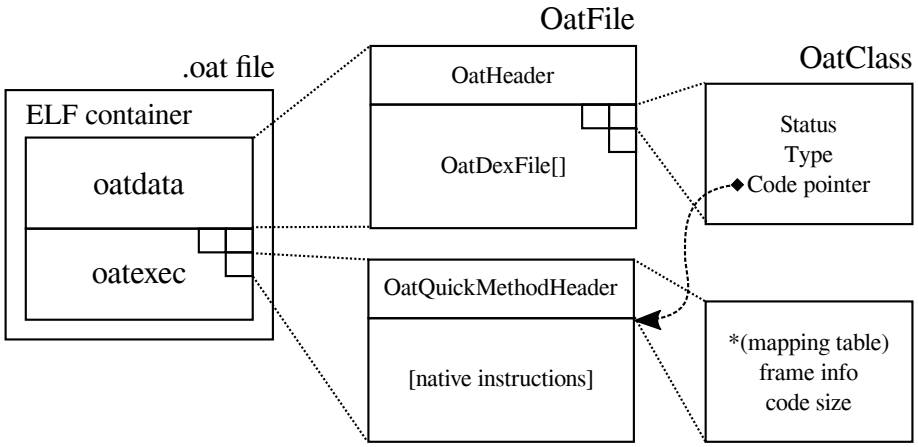

FIGURE 2.1: The OAT file format.
Some details have been collapsed or left out for increased readability.

The native code is stored in an ELF object `oatexec` that is mapped to an executable memory area at runtime, meta data and Dalvik byte code are stored in an object `oatdata`.

The `oatdata` object (see FIGURE 2.1) is a structure build around the original DEX file format to add ART specific meta data. While most OAT files are based on a single DEX file, the file format also supports the inclusion of more than one, which is used by the boot classpath and newer Android packages using the multidex feature, which is natively supported by ART.

The `oatexec` object contains the native code instructions for each method in no particular order. The `OatQuickMethodHeader` is stored directly before the entry point of each method and is only reachable by following the entry point. This header contains information on the method including its size in bytes and the influence the method has on the runtime (e.g. used registers and stack frame size) as well as a pointer to a table that maps the native code to the corresponding Dalvik byte code. This mapping is used to provide full stack traces on Exceptions including line numbers.

2.4 Optimization

Beside the ahead-of-time compilation, heavy optimizations are also an important reason for the performance increases introduced with ART. Some of the optimizations which affect method calling and hooking are also used on other platforms and compilers, others might be new.

Method inlining is an optimization already used by some compilers under certain conditions. Short methods (e.g. single return statements) are inlined into the calling method, reducing the need to jump through memory. Thus reducing memory usage and increasing the effectiveness of CPU caches. While the concept is known to compilers it is usually

```
1  class ArtMethod{
2      Class declaringClass;
3      ArtMethod[] dexCacheResolvedMethods;     // These caches are used when
4      Class[] dexCacheResolvedClasses;         // resolving methods, classes or
5      String[] dexCacheStrings;                // strings from the dex file index
6      long entryPointFromInterpreter;
7      long entryPointFromJni;
8      long entryPointFromQuickCompiledCode;
9      long garbageCollectorMap;
10     int accessFlags;
11     int dexCodeItemOffset;
12     int dexMethodIndex;
13     int methodIndex;
14 }
```

FIGURE 2.2: Fields in the ArtMethod class.

not used with dynamically linked methods if not explicitly expressed by the user (e.g. using the `inline` keyword in C/C++).

An important optimization specific to ART is the usage of direct jumps. Calls of methods inside the framework are replaced by direct jumps to the corresponding position in memory and thus method lookup is no longer needed for this calls. This is possible, as `boot.art` is always mapped to the same location in virtual memory. When `boot.art` changes due to a system update, the jumps in OAT files are updated using the tool `patchoat`. It is not necessary to recompile the complete byte code. If a method is loaded at runtime and therefore the call cannot be converted to a direct jump, a fallback routine is added, that will resolve the method call in the moment of its first occurrence.

Another optimization related to hooking is the method merging used with ART. Methods that share the same native code representation are merged into a single entry point in the `oatexec` object. While this has no side-effects on the intended program flow, it does mean that any modification to the native code of a method can have side-effects to another method, if it shares the same code.

2.5 Method handling

At runtime, methods on ART are represented by an `ArtMethod` (FIGURE 2.2) object. This object keeps references to the declaring class, string tables and the corresponding DEX method as well as pointers to relevant call entry points. When Java Native Interface (JNI) calls, reflection or unoptimized calls from compiled Java code occur, the corresponding `ArtMethod` is used to resolve the relevant entry point. There are three types of entry points, one for each call source. Usually the entry point for compiled Java code is the only one directing to the method itself[4]. The other two entry points are "bridges" between the different application binary interfaces (ABI) of JNI, reflection

[4] In native methods, the JNI entry point is pointing to the real method and compiled code entry point is a bridge.

and ART. If a method is not yet initialized, as it is in the boot classpath but not in the list of pre-initialized classes, the entry points will point to an initializer method. This way, each class is automatically initialized once a method inside it is called.

ART quick compiled code uses its own calling convention derived from the platform specific calling convention. It uses the first register for a pointer to the `ArtMethod` of the called method and an additional register for a pointer to the `Thread` object corresponding to the executing thread, that is used to access the thread-local memory. Everything else is the same as the native calling convention, with a pointer to the `this`-object being the first argument of non-static methods.

Chapter 3

Hooking

The technique of method hooking evolved over years to support different use cases and operating systems. In this chapter, we will present some approaches on method hooking and analyze if and how they can be implemented on ART. At the end we will present a solution that fulfills our requirements.

3.1 Requirements

Before analyzing existing approaches on method hooking and what problems might occur when using them, we define a set of requirements our approach should fulfill.

- **No operating system changes**: As Android does not allow normal users to modify the operating system and we do not want to restrict the advantages that come with method hooking to a limited number of devices and users, the hooking approach should not require the user to modify his operating system in advance.
- **Runtime, on-device hooking**: The hooking itself should happen on the target device and should not require additional computers. User interaction should in no case be required and it should be possible to apply and remove the hook at runtime.
- **Handle direct jumps**: Calls to Android system libraries residing in `boot.art` are done using direct jumps. These represent the majority of method calls, thus direct jumps should be handled correctly.
- **Support reflection API**: According to Felt et al., more than the half of Android applications use Java reflection [8]. For reasons of backward-compatibility, Android developers often use reflection to call methods that are not available on all versions. Furthermore some applications use reflection to use hidden system APIs. To be a portable, the solution should support the reflection API.

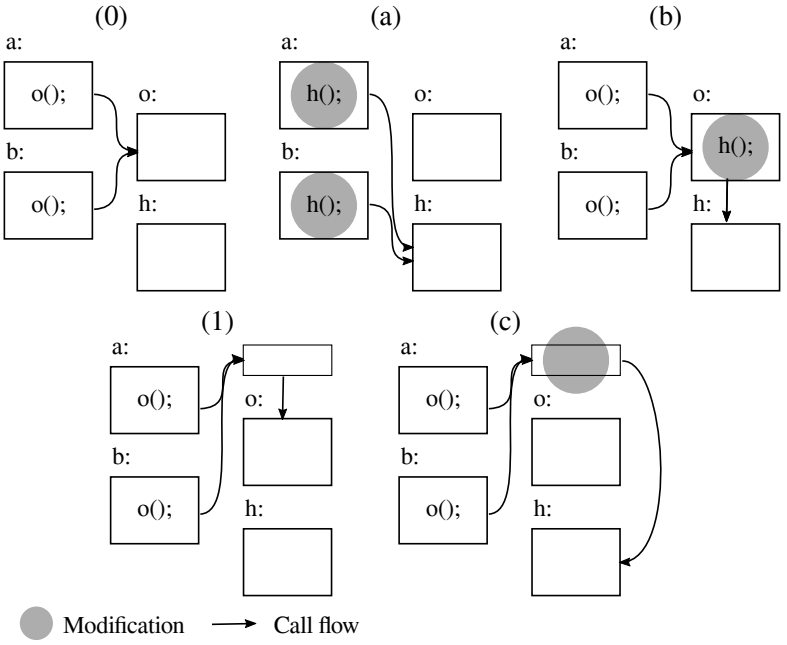

FIGURE 3.1: Hooking approaches
(0) before rewriting (a) caller-side rewriting, (b) callee-side rewriting
(1) before call diversion (c) call diversion

- **Hook inner calls**: The hooking should not be limited to calls to shared libraries, but should also apply to calls to methods inside the application itself.

3.2 Static hooking

The term "static hooking" usually applies to hooking approaches that inject the hook before the application, in which they are applied, is executed. This usually happens by modifying the machine code instructions of the application permanently (on disk) or by changing the environment.

3.2.1 Injecting shared libraries

One static approach is the modification of the runtime environment in a way, that a custom shared library is injected. When shared libraries are linked dynamically, a custom library can provide an implementation for a method usually implemented by the system, e.g. inside the `libc` standard library.

The Linux dynamic linker[1] allows the user to modify the paths in which it searches for dynamic libraries using the `LD_PRELOAD` environment variable. Additionally, the linker

[1] A dynamic linker is a tool used to ensure that dynamically linked libraries (e.g. DLL files) can be accessed from executables that use them.

provides a function `dlsym`, that can be used to retrieve a method with the same name from another library, allowing a hook to call the original function. This approach is often used for debugging or to analyze an application, as it is easy to apply and does not require deep insight in the application.

On Java runtimes, it is often possible to modify the classpath before an application is started and thus load different classes that contain the hooked methods. However, Android does not allow arbitrary classpath modifications, making this approach unusable for ART. Additionally this approach would only work for calls to shared libraries, but we announced the need to hook calls inside the application.

3.2.2 Static caller-side rewriting

The idea of the caller-side rewriting approach is to modify each call of the hooked method (see FIGURE 3.1(a)). This is done by finding all calls of the hooked method and either replacing them with calls to the hook, or surrounding it with additional code. The problem that normally occurs here, is that it is not possible to distinguish between calls and other code in some cases. Caller-side rewriting can also significantly increase code size and thus impact performance, as the same code has to be added multiple times.

When modifying Android DEX files, it is rather easy to find the calls, as one only has to search for a corresponding method invocation instruction[2] in the DEX byte code, which is unique for every method. This approach was implemented by AppGuard [3] to modify applications before they are installed. However, as Backes et al. already mentioned, this approach does not work at runtime, which we stated as one requirement before.

3.2.3 Static callee-side rewriting

It's also possible to apply a hook by modifying the called method itself, which is called callee-side rewriting (see FIGURE 3.1(b)). This is done by moving the normal method code to a new location, and inserting the hook where it was before. While this approach does not suffer from the problems described for the caller-side rewriting, it is not feasible as a general purpose approach. Most operating systems, including Android, do not allow modifying system libraries. This would be a requirement for static callee-side rewriting to be used with calls to system libraries, which usually are of high interest when doing instrumentation.

[2] `invoke-static`, `invoke-virtual` or `invoke-direct` for static, virtual or direct methods respectively

3.3 Dynamic hooking

In contrast to static hooking, the term "dynamic hooking" is used for mechanisms that can dynamically apply a hook at runtime. This is extremely useful if the decision to hook a method is based on the runtime environment. Dynamic hooking happens in volatile memory only.

3.3.1 Call diversion

Call diversion (see FIGURE 3.1(c)) is often used as a dynamic hooking approach. It makes use of the method table used by dynamic linkers in modern operating systems. Calling a method will cause the system to look up the methods position in memory using the method table before jumping to the memory location. On these operating systems, hooks can easily be injected by manipulating the entry in the lookup table corresponding to the hooked method.

Von Styp-Rekowsky et al. described how to use call diversion on Dalvik [9]. The Dalvik virtual machine uses a virtual method table internally, which holds a reference to each method's byte code. By manipulating this reference, each call to a method can be redirected.

On ART, most method calls do not use the virtual method table, although it still exists as a fallback mechanism for methods that could not be resolved during compilation. Instead the majority of methods are invoked by direct jumps, which call diversion can not handle. As we listed the support of direct jumps as a requirement before, call diversion can not be used to fulfill them.

3.3.2 Dynamic callee-side rewriting

Static callee-side rewriting is not feasible for the majority of operating systems because they do not allow to modify system libraries. Dynamic callee-side rewriting is based on the same idea, but circumvents this issue by modifying the in-memory copy of system libraries instead of the originals. As these are loaded into the private address space of the application using them, most operating systems grant the application write access to the in-memory copy (or provide means to do so).

A dynamic callee-side rewriting approach for Windows was described by Hunt et al. in 1999 [10]. Their implementation "Detours" provided one of the first dynamic hooking approaches that caused low performance and memory footprint.

3.4 Approach for ART

For ART, we found a dynamic callee-side rewriting approach to provide the most sophisticated results. By changing the prologue of the method being hooked, we redirect the call to another method leaving most of the original method intact. As the compiled method is the only one being called on ART, whether the source of the call being a direct jump, reflection or JNI, this approach promises good results without high performance impact. However, we are not able to handle the method inlining optimization when using callee-side rewriting. As the original method is no longer called if it was inlined, it is not possible to modify the result of the method call by changing the method in memory. But as method inlining only happens for extremely short methods that only consist of static return values, we found this problem to be negligible.

Chapter 4

Implementation

To verify our findings and for further evaluation, we implemented the approach as a library called "ArtHook". This chapter will cover further details on this implementation.

4.1 Design decisions

We accomplished our approach of callee-site rewriting with focus on portability. Non accessible fields and methods are used as few as possible and JNI is only used to call operating system methods like `mmap`. Our implementation is compatible with 32-bit ARM and Thumb-2[1] instructions, which are the most common instruction sets used on Android devices. To reduce the possibility for side-effects in a future version of ART, all native code injected does not modify the memory. Additionally we also wanted our library to provide an API, that allows developers to easily apply a hook.

4.2 Callee-side code injection

To inject code into the callee, the corresponding native code section needs to be writable. By default it is mapped to the virtual memory readable and executable, but not writable. We bypassed this issue by using the `mprotect` method of the Linux operating system, which is able to remove the protection from the corresponding memory page.

The first bytes of the hooked method contain the prologue which sets up the stack and stores the processor state in it, which is later recovered once the method returned. We backup the prologue machine code and replace it with machine code that causes a jump.

[1] Most ARM processors support two instruction sets: 32-bit ARM instructions and mixed 16/32-bit Thumb-2 instructions.

```
0x0: f000f8df      ldr.w pc, [pc]        Instruction to load the memory content at
                                         the program counter into pc
0x4: b3d9d001                            The jump target address loaded by the
                                         instruction above
```

FIGURE 4.1: Code injected into the prologue of the callee.
This example uses Thumb-2 machine code.

```
0x00: c014f8df      ldr.w ip, [pc, #20]     Load the address of an ArtMethod
                                            object into the ip register
0x04: 4560          cmp r0, ip              Compare the address with the
                                            ArtMethod called
0x06: 8009f040      bne.w 0x1c              If it does not equal, jump to next
0x0a: 4801          ldr r0, [pc, #4]        else load the address of the
                                            ArtMethod of the hook into r0
0x0c: f004f8df      ldr.w pc, [pc, #4]      and jump to corresponding code
0x10: 7458a588                              The address of the ArtMethod object
                                            of the hook
0x14: b3db6cfd                              The address of the code of the hook
0x18: 7458d1f8                              The address of the ArtMethod object
                                            of the hooked method (for comparison)
[...]   Repeat above for merged methods
0x1c: 5c00f5bd                              Execute the original prologue of the
0x20: c000f8dc                              hooked method
0x24: f000f8df      ldr.w pc, [pc]          Jump into the original code after
0x28: 73b40cbd                              the prologue
```

FIGURE 4.2: Code of generated memory pages
This example uses Thumb-2 machine code.

Unfortunately, most instruction sets including ARM and Thumb-2 do only allow to use branch instructions, which are normally used to jump to a different part of the memory. We work around this restriction by building a jump instruction out of two parts (see FIGURE 4.1). The first part contains the instruction to load the memory of the second part into the register that holds the program counter (`pc`). The second part is not an actual instruction, but the address of the jump target. By writing an address into `pc`, we actively manipulate the control flow, without using the branch instructions.

4.3 Custom executable memory page

The target of the jump injected into the callee is a memory page specifically created for each method body. This means that if two methods are merged and thus share the same body (which is a possible optimization), there will be a single memory page for both. This is needed, as the callee-side code injection will apply to all methods sharing the same body.

This memory page contains routines to determine the actual method called from a merged method by analyzing the `ArtMethod` object reference, which is stored in the first register (`r0`) according to the calling convention. By comparing `r0` with the address location of the `ArtMethod` object of the hooked method, we can determine the hook that

should be executed. As the ARM processor does not allow to compare a register with the memory, we use the Intra-Procedure-call scratch register (`ip`) to temporarily store the address for comparison. According to the ARM calling convention, the `ip`-register does not give any guarantees to preserve its value after a method call. We can therefore modify it without causing side-effects.

If this procedure found a method call to be hooked, the corresponding hook is executed. To stay compatible with the calling convention and to allow the runtime to generate valid stack traces, we first update `r0` to point to the `ArtMethod` object of the hook and secondly update the program counter, using the same method as described for the callee-side code injection. As we did not touch the stack, no cleanup is required after the hook exits, which allows us to keep the link register which contains the return address as it was originally. These steps are repeated for every hook applied to a method body, which might be more than one.

If no corresponding hook can be found, we execute the original prologue (which we backed up before) and modify the program counter to continue execution of the original machine code right after the modified prologue. This does not only allow us to handle merged methods correctly, it also provides an easy way to call the original method from everywhere if we want to. By creating a clone of the `ArtMethod` object, we can create a Java reflection `Method` object, which behaves the same, but resides in a different memory address. Due to this difference, the comparison will fail and thus the invocation of the clone will execute the original method.

4.4 Library design and usage

As we wanted our library to be easy to use, we put a lot of the logic code into the background to provide developers with a very small API. As the logic code requires some input from the developer, we use Java reflection and annotations to provide them.

To implement a hook, the developer needs to define the hook as a static method and annotate it with the `Hook` annotation. The method needs to define an object which will become the `this` object as first parameter and the method parameters afterwards. The `Hook` annotation takes a string as a parameter that defines the method that should be hooked. The string is build up from the package name, class name and method name, for example `java.util.ArrayList->add`. If the method is overloaded, the actual implementation will be automatically choosen by the parameter types of the hook. Every hook declared in a class this way can be applied dynamically by calling `ArtHook.hook(Class)` with the hook defining class as a parameter. Alternatively, it is possible to hook Methods directly by calling `ArtHook.hook(Method, Method)` with original and replacement Method object as parameters.

```
1  public class MyApplication extends Application {
2      @Override
3      public void onCreate() {
4          super.onCreate();
5
6          // initialize the hook(s) on Application creation
7          ArtHook.hook(MyApplication.class);
8
9          // Call a hooked method
10         new SipAudioCall(this, null).startAudio();
11     }
12
13     @Hook("android.net.sip.SipAudioCall->startAudio")
14     public static void Sip_startAudio(SipAudioCall call) {
15         Log.d("Debug", "This hook is invoked");
16
17         // Call the original method
18         OriginalMethod.by(new $() {}).invoke(call);
19     }
20 }
```

FIGURE 4.3: Example illustrating the functionality of the hooking library

Another interesting part of the API is calling the original method. We provide multiple different means to do so, as we found that easy solutions might cause problems in some rare cases. These are implemented in a dedicated class `OriginalMethod`, which contains methods to retrieve pointers to original methods as well as the possibility to invoke them. Once an `OriginalMethod` object is accessible, its `invoke` (or `invokeStatic` respectively) method can be used to invoke the original method.

We provide an API to retrieve the original method based on the current stack state. This is done by searching the stack trace for a method that is annotated with `@Hook` and using that as a base to find the the original method. This can be done by calling `OriginalMethod.byStack()`. The caveat of this method is that it is not able to handle method overloading, as Java stack traces do not contain the information required to distinguish two methods with the same name. Another approach that does not require to explicitly name the method inside itself is the use of an anonymous inner class. The reflection API has a reference to the holding method for each anonymous inner classes, making them usable as anchors as well. `OriginalMethod.by(new $() {})` provides exactly this behavior.

Both of these approaches share a common issue: they make heavy use of reflection. Reflection is known to be rather slow in some cases. Additionally this will make it impossible to hook the reflection API itself, as it would cause a recursion loop. We provide an easy way to call the original method that does not have this issue. It requires the method to be annotated using `@BackupIdentifier`, which takes a single unique string as parameter. The original method can than be invoked by calling `OriginalMethod.by(String)` with this unique string as parameter. This is also possible when applying the hook manually using `ArtHook.hook(Method, Method, String)`.

Chapter 5

Evaluation

5.1 Functionality

To prove correctness of our approach as implemented by the ArtHook library, we used it in a sample application. In this application we tested all different kinds of methods to prove that it works correctly.

After it was tested intensively in our test environment, the ArtHook library was inserted into the latest version of AppGuard [3], which uses it to dynamically apply a security policy in monitored applications. The inclusion of ArtHook did not cause greater problems and since AppGuard 2.3, ArtHook is part of a release grade software.

5.2 Performance

Beside functionality, performance plays an undeniable importance when hooking methods, especially when an applications with real-time routines is hooked. We thus also analyzed performance implications caused by ArtHook. We executed these tests on a real device, specifically the second generation "Nexus 7" tablet, which comes with a 1.5GHz quad-core processor.

5.2.1 Micro benchmark

We started our performance measurement by doing a micro benchmark. The idea of this benchmark is to analyze the time needed to call a certain method. In our case, the target method was empty, because our measurement should focus only on the hooking. We measured a test environment overhead of a few nanoseconds per call for the test loop. Depending on the time required for the operation tested, we repeated it between

Testcase	Time
Applying all hooks in a class automatically	~9ms/hook
Applying hooks manually	~7ms/hook
Test environment overhead	~0.000002ms/call
Call an empty method	~0.000009ms/call
Call an empty hook	~0.000010ms/call
Call a hook that uses `preFetchedMethod.invoke()`	0.002ms/call
Call a hook that uses `OriginalMethod.by("ident").invoke()`	0.008ms/call
Call a hook that uses `OriginalMethod.by("ident")`	0.004ms/call
Call a hook that uses `OriginalMethod.by(new $() {}).invoke()`	0.561ms/call
Call a hook that uses `OriginalMethod.by($.class).invoke()`	0.560ms/call
Call a hook that uses `OriginalMethod.byStack().invoke()`	0.831ms/call

FIGURE 5.1: Micro benchmarks of different features of ArtHook

10 thousand and 10 million times to receive a good average and measurable results. The results of the micro benchmark are listed in FIGURE 5.1.

At first, we analyzed the hooking process itself. The memory allocation and modification done during the hooking is native code. It also contains several context switches into the kernel. This resulted in a raw hooking time of about $7ms$ per hooked method. Additional $2ms$ are required to analyze a class and the hooks stored inside if the easy API with `@Hook` is used. Although the overhead of $7ms$ is quiet heavy, it can be considered negligible for most users, given that the number of hooks applied will not exceed a few hundred in most cases.

Secondly, we wanted to know what the performance overhead caused by the redirection instructions, that are added to the method, is. As these are only a few additional machine instructions, we expected it to be very low. The benchmark results confirm this expectation: The additional time required by a hooked method call compared to a normal method call was about one nanosecond.

And finally we analyzed the call of the original method from inside a hook. As described in section 4.4, we provide different means to call the original method. As some of them make more use of the reflection API than others, we expect to see differences in performance. The invocation of a method via reflection takes about 200 times longer than without reflection.

This however first requires to have a reference of the method object. Looking up a method based on the `@BackupIdentifier` annotation and calling it, will take a total $8\mu s$ per method call. Calling a method based on an anonymous inner call takes about a half millisecond and traversing the stack to obtain a reference requires even more time.

Test	Reference	Hooks applied		Hooks applied, original method pre-fetched	
Canvas	58.38	57.87	-0.8%	58.46	+0.1%
Text	58.10	58.44	+0.5%	58.37	+0.4%
Circle	59.33	20.24	-65.9%	56.28	-5.1%
Circle2	31.18	29.61	-5,0%	30.19	-3.2%
Rect	6.44	3.72	-42.2%	6.21	-3.6%
Arc	14.54	9.21	-36.7%	14.00	-3,7%
Image	31.26	13.76	-56.0%	29.83	-4.5%
Average			-41.2%		-4.0%

FIGURE 5.2: Benchmark results of 0xbench 2D. Values in frames per second. The average given does not consider the tests "Canvas" and "Text".

5.2.2 `0xbench` 2D

To analyze how ArtHook performs in larger scale, we used it with `0xbench`. `0xbench` is an open-source Android benchmark suite covering several test-cases like 2D- and 3D-graphics floating-point arithmetic and native JavaScript performance. Most test-suites do not make use of enough Java method calls, to show a significant difference when hooked or not.

`0xbench` 2D uses the Android canvas API to draw screen contents in real-time. By hooking into the canvas API we thus have substantial influence on the test result. Additionally this benchmark covers an API that is used often by applications to draw screen contents and requires a lot of method calls.

For the benchmark, we hooked into 6 drawing entry points in the Android canvas API and additionally in the `View.onDraw` method, which is called whenever the screen is redrawn. Hooking into a method here means that we inject a hook that does nothing but calling the original method. One benchmark run (which includes 7 tests) takes about 330 seconds and involves of about 270.000 calls to the methods we targeted. The results of the benchmark are presented in FIGURE 5.2.

When we started the benchmarking, we noticed that the results of two tests ("Canvas" and "Text") are not useful for us. As Android uses vertical synchronization when drawing screen contents to reduce tearing artifacts, the maximum volume of frames generated is capped at 60 FPS (frames per second). As a result, the number of frames generated with and without hooking does not differ when using these two tests. We therefore did not use them for the following analysis. As the results of the other tests sometimes fluctuated by a few frames (although `0xbench` already averages them), we ran every test 5 times and took the average of them.

At first, we did the benchmark by using an anonymous inner class to call the original method. The result was a performance decrease of about 41% on average. This is an expectable result considering that our micro benchmark showed that the method calls involved in the test will take up about 135 seconds.

We then repeated the benchmark using a hooking method that does not lookup the original method every time, but fetches it one time and uses it afterwards. As seen in the micro benchmark, this reduces the amount of work drastically. In the `0xbench` 2D benchmark, the performance loss was reduced from 41% down to 4% when using cached original methods.

Chapter 6

Related Work

We presented related work in form of hooking approaches in chapter 3. This chapter introduces a different approach on hooking as well as a well-done use case in a single project: The Xposed Framework.

6.1 Xposed Framework

The Xposed Framework [11] is a modification of the Android operating system, that allows modules to inject hooks into any application or system service. This makes it a powerful, extensible and easily-usable modification framework, resulting in high popularity within users that use non-default operating systems on their smart phone.

6.1.1 Xposed on Dalvik

Xposed was originally developed for Dalvik on Android 4.0 and was later extended to work on Android versions up to 4.4. The framework is inserted in the system by modifying `app_process`, the executable called whenever a new Java process is started. It adds a new library to the Java classpath and modifies the start-up, so that the framework can be initialized and modules can be loaded. The actual hooking is done by doing call diversion. Dalvik's `Method`-object that exists for each method is the entry point: It allows to do arbitrary modifications to the method. Xposed modifies the method and marks it as native. Further calls of the method will no longer be directed to its original body but the entry point marked inside the `Method`-object, which is modified as well to point to a global native method. All calls of the method will then be directed to this native handler, which itself forwards the call to the actual hook.

6.1.2 Xposed on ART

At the time of writing, a technical preview of the Xposed Framework for ART and Android 5.0 has been released. As it was for Dalvik, the Xposed version for ART relies on modifications of the operating system. But instead of only extending it, Xposed also modifies it. The most effective change is the modification of the ART compiler `dex2oat`. The compiler is modified to disable the optimizations direct calls and method inlining. This way, all calls will require to lookup the methods entry point in the `ArtMethod` object. At this point, Xposed for ART behaves like its counterpart for Dalvik: The method is marked as native and the corresponding entry points are modified. Due to the strong integration with the operating system and as patching the binary operating system is hardly possible in most cases, Xposed for ART currently lacks portability, reducing the number of users drastically.

Although Xposed provides a hooking mechanism for ART, it works on a basis of modifying the underlying operating system, making it less usable for wide-spread application. The capability to hook other processes however is worth the lack of portability.

Chapter 7

Conclusion

In this thesis we presented a method hooking system for the new Android runtime ART. By using callee-side in-memory method rewriting, we were able to address the special requirements of the runtime environment caused by the ahead-of-time compilation and optimizations introduced with it. We also provided an overview of ARTs behavior and binary files. We developed the ArtHook library, which provides the described hooking mechanism with an easy API to developers for debugging and security purposes.

7.1 Limitations and future work

The hooking approach presented is not able to handle calls of inlined methods. We found that it will be possible to hook these methods by using dynamic caller-side rewriting. This however is likely to have a large performance impact which is not desirable. Dynamic caller-side rewriting can thus only be an extension of an existing hooking approach, but should not be used stand-alone.

During the evaluation we noticed that our easy API has significant performance disadvantages compared to a more direct usage of the hooking methods. This is partly addressable by caching the method resolution inside the hooking library. However, additional API improvements might be needed.

The ArtHook library currently only support 32-bit ARM CPUs. Our implementation also makes use of a few singularities inside the ARM calling convention and instruction set, which might or might not be portable to other CPUs. The latest Android version supports 64-bit ARM processors, 32- and 64-bit x86 processors and 32-bit MIPS processors, with support for 64-bit MIPS processors being planned for a future release. To be portable and future-proof, ArtHook should support all of these processors and the instruction sets provided by them.

The approach used within the Xposed Framework to hook into remote processes is a nice feature. Although it might require permissions normal users do not reach, it also increases the set of possible features. In combination with such a framework, an application like AppGuard would no longer need to ask the user about updates or force him to uninstall, but can dynamically apply its policies.

7.2 Availability

The ArtHook library is freely available. Its latest version including sample code can be found in source form at https://github.com/mar-v-in/ArtHook.

Bibliography

[1] Gartner Inc., "Gartner says tablet sales continue to be slow in 2015." at http://www.gartner.com/newsroom/id/2954317, January 2015.

[2] A. R. 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, (New York, NY, USA), pp. 49–54, ACM, 2011.

[3] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard: Enforcing user requirements on android apps," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, (Berlin, Heidelberg), pp. 543–548, Springer-Verlag, 2013.

[4] Android Open Source Project, "Introducing ART." at http://source.android.com/devices/tech/dalvik/art.html, now available at https://web.archive.org/web/20131104002414/http://source.android.com/devices/tech/dalvik/art.html, November 2013.

[5] Android Open Source Project, "Configuring ART." at https://source.android.com/devices/tech/dalvik/configure.html, March 2015.

[6] B. Carlstrom, A. Ghuloum, and I. Rogers, "The ART runtime." at Google I/O 2014, now available at https://www.google.com/events/io/io14videos/b750c8da-aebe-e311-b297-00155d5066d7, June 2014.

[7] TIS Committee, "Tool interface standard (TIS) executable and linking format (ELF) specification, version 1.2." at http://refspecs.linuxbase.org/elf/elf.pdf, May 1995.

[8] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), pp. 627–638, ACM, 2011.

[9] P. von Styp-Rekowsky, S. Gerling, M. Backes, and C. Hammer, "Idea: Callee-site rewriting of sealed system libraries," in *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ESSoS'13, (Berlin, Heidelberg), pp. 33–41, Springer-Verlag, 2013.

[10] 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, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 1999.

[11] "Xposed module repository." at http://repo.xposed.info.