A-SIT Technology
E-Government Innovation Center

5 July 2021 Study · Mobile Security · Web & Cloud · 23 min read

Android Application Patching Through Runtime Manipulation

Archived. This content is no longer updated. It remains available for reference.

Version 2021-06-23
Source https://technology2.a-sit.at/en/publications/android-application-patching-through-runtime-manipulation/

Application patching is a promising concept for modifying the execution flow of a compiled Android application for the purpose of analyzing its implementation, mitigating flaws or adding new functionality. While traditional static application patching is computationally expensive and thus has to be carried out offline on powerful desktop machines, a dynamic approach that manipulates the data structures of the ART runtime can operate on the device directly and opens the technology to many more usage scenarios.

In this research project, we investigated the feasibility and practicality of this approach. Several existing open-source implementations were identified and analyzed with regards to their strengths and weaknesses. Based on the most reliable core components and an understanding of the implementation details of the ART runtime, we designed a custom solution for dynamically intercepting method calls in arbitrary third-party applications. The resulting software library was integrated into a full on-device application patching pipeline and successfully tested against several of the most popular applications from the Google Play Store.

1. Introduction

While mobile devices like smartphones and tablets are growing their significance as central communication and data hubs in people's lives, they have increasingly drawn interest as the subject of scientific research. Mobile security researchers are seeking to find out how applications running on these new platforms process user data and whether proper protection is implemented for on-device storage and communications to remote servers. As part of these efforts, the past years have shown the discovery of severe flaws in popular programs that can cause data leaks to unsophisticated attacks from malicious third parties [1]. Frequently, these flaws can be traced back to incompetency or ignorance on application developers' part, who for example fail to follow © A-SIT, Partner für Sicherheitsfragen recommendations and best practices for implementing basic cryptography. Many popular applications suffer from these issues to this day, despite numerous efforts from the platform providers [2].

For cases where applications suffering from these data protection flaws still provide unique functionality that users depend on, a third-party solution is needed that is capable of mitigating and detecting these flaws on the fly. Application patching is a particularly promising solution to this problem for the popular Android platform. It involves modifying a compiled application package with the goal of changing its execution flow. This allows the injection of instrumentation fixtures for detailed analysis of a specific application, as well as mitigating flaws in the original application's operation. Since the technique does not require modifications to the operating system of the target device, it can be deployed effortlessly to an unlimited range of different devices.

Most commonly, application patching is carried out statically [3] [4] [5] [6], meaning that modifications are applied in an offline process, before execution of the program. However, since this approach requires parsing the complete program code of the target executable for locating points that need to be patched, it is computationally expensive. This impedes the solution's applicability for scenarios where the whole patching operation needs to happen on the target mobile device. A more suitable approach in that case is dynamic patching, where the execution of a target application is manipulated at runtime, either through an external process or through an instrumentation component that is injected into the target executable offline.

The goal of this project was to identify a method suitable for dynamically patching method invocations of arbitrary compiled Android applications. This required gaining an understanding of the architecture of the Android Runtime (ART) responsible for executing applications on the Android OS, as well as of implementation details regarding the method invocation mechanism and the integrated just-in-time (JIT) compiler. Several existing implementations of Android method hooking were reviewed [1] and their respective strengths and weaknesses compared. We then designed our own custom hooking solution that builds on the most reliable primitives from the evaluated projects. Based on our design, we implemented a software library which was integrated into a complete on-device pipeline for automatic Android application patching that will be published in the near future.

2. Background

The Android ecosystem consists of a widely fragmented landscape of devices differing in CPU architecture, available resources and hardware peripherals. In order to still ensure support for executing third-party applications across the whole range of available hardware, the Android engineers decided to build on the write-once-run-everywhere principle of the Java platform. While the language itself and the standard libraries were mostly carried over from the desktop Java environment, Android added a custom set of APIs for accessing certain system resources and a Java Runtime that differs significantly from the desktop implementation.

As part of the compilation process, the Android build tools compile Java source code into a proprietary bytecode format stored in Dalvik executable (DEX) files. On the target device, it is the responsibility of the Android Runtime (ART) to parse the class hierarchy from the DEX files and execute the contained code. In order to efficiently fulfill this task in the resource-constrained environment of a battery-powered portable device, ART in its current incarnation integrates optimizations such as ahead-of-time (AOT) and just-in-time (JIT) compilation of DEX code into native machine code that can be executed more efficiently. For maximal performance, application developers can also provide parts of their software as shared objects containing native machine code compiled from languages such as C or C++. This native code is then executed directly

on the 1 Since no actively maintained solution from the scientific community exists, these were all open-source projects CPU instead of going through preprocessing or interpretation of the Java runtime, but it can still interact with managed Java code.

Another core responsibility of the Android runtime is memory management. For sparing application developers from the duty of manually allocating and freeing memory space, the Android runtime integrates a garbage collector (GC). This subsystem keeps track of all allocated objects and periodically evicts those that are no longer referenced by any other object.

3. ART Architecture and Implementation

Conceptually, the core components of the ART project can be separated into the compiler (including different compilation backends for the AOT and JIT compilers) and the runtime itself.

3.1. Compiler

The compiler system was designed with an abstract compiler interface, so that the compiler backend ("compiler drivers") can be exchanged transparently to the rest of the system. In recent versions of the Android OS, the default backend is the Optimizing backend, which generates native code that is mostly compatible with its predecessor Quick, which is why many places in the ART source code still carry references to that name. In order to speed up execution of its compiled code, the Optimizing backend integrates several optimization techniques, such as inlining of method calls, as well as inlining of complete method implementations, if they fulfil certain criteria.

Generally, when an application is started for the first time after installation, it is executed purely in interpretation mode, meaning that its methods are not run directly on the CPU but through a software implementation of the load-fetch-execute cycle for the Dalvik bytecode. For every method, a hotness counter is maintained that indicates how frequently it has been accessed during the current application execution. Once the hotness passes a certain threshold, the JIT compiler is invoked in a background thread to produce a native machine code translation of the method's DEX code implementation. For subsequent invocations of the method, this generated machine code function is called in place of the interpreter. Additionally, a dedicated profiler thread monitors the application runtime in order to identify particularly hot methods crucially influencing overall performance. Based on the collected data, the AOT compiler is invoked when the device is idle and connected to a power source. The AOT compiler essentially builds on the same compiler backend as used for JIT compilation, except that the generated machine code is permanently stored to disk in a special OAT file format and is used whenever one of the processed methods is called in subsequent application launches.

3.2. Runtime

The runtime internally organizes its view on the executed application in several data classes that follow the structure of the parsed DEX files. For every loaded class from the DEX file, instances of the ArtMethod and ArtField runtime classes are allocated that reflect the properties of the corresponding methods and fields for quick access by runtime routines. For the purpose of this project, the ArtMethod class is of particular interest, since applying suitable changes to its fields can be used as a simple means for replacing method implementations.

The metadata stored in the fields of the ArtMethod class encode flags originating from the Java source code the DEX file was compiled from, such as method visibility (public, protected, private), scope (static, instance) and kind (native, non-native), as well as runtime-internal bookkeeping flags. The latter for example encode whether a method should be considered for JIT or AOT compilation if it was found sufficiently hot.

Another essential set of fields in the ArtMethod class map how the represented method fits into the bigger context of the whole application and its package. This entails a reference to the class that declared the method (which in turn allows identifying the associated DEX file and used class loader), as well as the offset in the method index of the associated DEX file. Using this information, the interpreter can retrieve the corresponding byte code when the method is to be executed.

Since application and framework methods may be available in various execution formats with varying calling conventions, the ArtMethod structure also holds pointers to several execution entry points the runtime choses from depending on the current execution context (interpreter, quick, JNI). Some of these entry points may actually point to generic stubs that just switch to the correct execution context for the given method, e.g., by translating stack and register arrangement between different calling conventions. An entry point that is of particular significance for the purpose of this research is the entry point from quick code. If the particular method has been compiled by ART (either by the JIT or the AOT compiler), this field points to the memory region that contains the native machine code representation of the method's implementation.

4. Native Hooking

A core technology for hooking execution of application methods within the Android runtime is being able to intercept execution of methods of the ART runtime itself. Since the latter is written in C++ and compiled to native machine code that is executed on the target CPU, we can to some degree take advantage of common method interception techniques known from traditional Linux-based desktop platforms.

While the most basic approaches such as preloaded library shims via the LD_PRELOAD environment variable [7] and simple dlsym lookups do not work due to restrictions in recent versions of the Android OS, we can still benefit from the fact that native code on the Android platform is executed from ELF binaries and uses a common dynamic linker system. By manually mapping the runtime shared object libraries into memory and parsing the ELF headers, we can locate relevant symbols in the process's memory space. By carefully manipulating the first few instructions at the identified address, the execution flow can be diverted to a replacement function. Because the overwritten original instructions are relocated to a different memory location, the replacement function can still call the original implementation. Since the complete native hooking operation requires a complete disassembler for all supported CPU architectures, we took advantage of the relevant functionality of the open source SandHook [2] project for this purpose, which we found most reliable.

5. Basic ART Hooking

The most trivial approach for hooking (i.e., intercepting execution of) an application method in the ART runtime would be simply modifying the quick entry point in the ArtMethod structure. While the ArtMethod instances for application methods are not generally accessible to application code, they can still be located using Java's Reflection API. With the help of some offset calculations depending on the Android version, the contained fields can be identified and modified accordingly.

While this approach generally works in some cases and is implemented e.g., by the YAHFA library [3], it does suffer from major problems that are rooted in the inlining optimizations performed by the ART compiler. Specifically, ART's Optimizing compiler backend occasionally inlines method calls, so that the quick entry point is directly embedded into the generated machine code of a caller method instead of queried during execution. This means that at the time of modifying the quick entry point, its original value might have already been hardcoded into other places of the executable code. Since no data structure in the runtime allows quick access to all callers of a given method, it is impossible to propagate modifications to all these places.

As a mitigation for this problem, a strategy similar to that used for native hooking can be applied, where the first few instructions of the target method implementation are relocated and replaced with a condition-less branch to the replacement method.

2 SandHook: https://github.com/asLody/SandHook 3 YAHFA: https://github.com/PAGalaxyLab/YAHFA The complete process follows these steps:

1. A target method and a replacement method are identified. The latter needs to be a static method that takes the same arguments as the former, except that the instance object (this) is explicitly added as the first argument (it is implicitly passed as the first argument for instance methods). An optional backup method can also be supplied, which will act as a handle to the original method implementation.
2. Since owner classes of static methods are lazily loaded, if the target method is a static method, it is called with a dummy instance object and dummy arguments to ensure that the class has been parsed from the DEX file.
3. The JIT compilation is invoked on the target method to ensure that it is marked as compiled. This can be accomplished by manually parsing the runtime ELF headers in order to locate the address in the process memory space of the compiler interface.
4. The flags of the ArtMethod instance are modified to indicate that the method should not be considered for AOT compilation.
5. Two regions of executable memory are allocated: a. The trampoline region is where the address of the target ArtMethod in register X0 (on ARM) is overwritten and a jump to the replacement method is performed. b. The backup region is where the original first instructions of the target method ("prologue") are copied to. After the prologue, the backup region contains a branch to the remainder of the original method. Thus, pointing the program counter to the backup region will effectively run the original method implementation.
6. The first few instructions of the original method are overwritten with a branch to the trampoline region. The same modification is applied on the backup method, except it will point branch to the backup region.
7. Subsequent invocations of the hooked method will end up executing the replacement method instead. The original method implementation can still be reached through the backup method.

While this procedure does work and is implemented in open-source projects such as SandHook [4] it is still far from the ideal solution. For one, it requires a suitable replacement method with matching signature for every hook, which leads to unnecessary overhead for large patches where multiple replacement methods

perform the same job. If multiple target methods are redirected to the same replacement method, the information about the originally called method is lost. This is particularly problematic for super-calls if a method implementation is hooked in both the parent class and the child class.

6. ART JNI Hooking

In order to address the problems of the approach described in Basic ART Hooking, a slightly different technique was pioneered by the Whale [5] and Frida [6] projects, which takes advantage of the fact that application method implementations may be supplied by application developers in native machine code as part the Java Native Interface (JNI) [7] . In this case, the method needs to be marked as native in the Java source code and a matching native function following a specific naming convention needs to be exposed by a shared object library that is loaded at runtime. Whenever the marked Java method is called, the runtime takes care of switching the execution context and invoking the corresponding native function. Internally, the ART runtime implements this functionality by keeping a pointer to the native function in the ArtMethod structure, where it can be manipulated for the purpose of method interception.

4 SandHook: https://github.com/asLody/SandHook 5 Whale: https://github.com/asLody/whale 6 Frida: https://frida.re 7 Java Native Interface: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html In order to get rid of the need for an existing matching replacement method for every hook, the JNI Hooking approach can utilize the LibFFI [8] library to dynamically generate native replacement methods. LibFFI in general can be used for dynamically invoking methods in any calling convention. Additionally, a feature that is crucial to our use of the library here, it has the ability to generate so-called closure functions, which are (native) functions that take a specified number of arguments in a specified calling convention and pass them as an array to a specified dispatcher function. A userdata pointer can be specified at closure creation, which will be passed to the dispatcher function along with the arguments. This allows us to generate a new native closure function for every patched Java function. By specifying a userdata pointer to a struct that contains the replacement method, we can use the JNI API to call the hook method from the dispatcher function.

This is the complete procedure that needs to be followed:

1. A LibFFI closure function is generated with the user data set to a struct containing references to the replacement method.
2. A stub Java method is identified. It can be pulled from a pool of preallocated empty methods (the signature doesn't have to match) or generated as a proxy method through Java's Dynamic Proxy system.
3. All Properties of the original method's ArtMethod instance are copied to the stub method's ArtMethod instance. From this point on, the stub method will appear like the original method in stack traces.
4. The flags in the ArtMethod instance of the stub method are modified to mark the method as native and the JNI entry point is set to point to the generated LibFFI closure.
5. The quick entry point of the stub ArtMethod is set to point to the generic JNI trampoline, which prepares the stack and registers for transitioning between the quick and JNI calling conventions.
6. The Basic ART Hooking process described above is executed using the stub method as the replacement method.

7. Whenever the target method is called, execution will jump to the replacement method, where the generic JNI trampoline will transition to the native closure function. From there, the replacement method is called through the JNI API.

Because a unique stub method and closure function are used for every hook, the originally called method can be identified through traversing the call stack and maintaining a mapping between target methods and stub methods.

Still, this solution suffers from major reliability problems. The generic JNI trampoline does not correctly set up handling of exceptions that occur during the execution of the native closure function. As a result, the runtime crashes whenever the replacement method or any of its nested calls raises an exception that is to be handled earlier in the call stack than the stub method.

7. ART Proxy Hooking

As a solution to these problems, yet another peculiarity of the Java language and its manifestation in the ART runtime can be exploited. Specifically, Java supports dynamically generating interface implementations through its Dynamic Proxy Class API [9] . To this end, the API provides a method that takes a list of interfaces and an InvocationHandler instance, and returns an instance of a dynamically generated class that implements all specified interfaces. Whenever one of the interface methods is invoked on the instance, the method name and all arguments are passed to the InvocationHandler, which can also control the return value.

For the purpose of this research, we take advantage of the specific implementation of this mechanism in the Android runtime. By using reflection, we can access a private method in the 8 LibFFI: https://github.com/libffi/libffi 9 Dynamic Proxy: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html Dynamic Proxy Class API that allows the generation of a class of a chosen name that implements methods of any other existing class, not just of interfaces.

The methods of the generated class can then be used as the replacement methods for the Basic ART Hooking approach described above. In order to handle invocations of these replacement methods, the native dispatcher function for Proxy method calls in the ART runtime needs to be intercepted. This is necessary because the runtime usually expects the implicit first argument of these calls to be an InvocationHandler instance, which is an invalid assumption for hooked methods, where the first argument (this) may be of any type.

Overall, the procedure for this approach is:

1. The Native Hooking technique described above is used to intercept the ART runtime's Proxy dispatcher function (InvokeProxyInvocationHandler). The replacement implementation checks the type of the first argument for deciding whether the original implementation should be called.
2. Using the private generateProxy() method of the java.reflect.Proxy class, a new class is generated that implements the target method. A reference to the new class needs to be stored in a set, so that it does not get garbage collected. A mapping is maintained between the target method and the replacement method.

3. Basic ART Hooking is applied for redirecting the target method to the generated replacement method.
4. For subsequent invocations of the target method, the hooked Proxy dispatcher function will be called. If the implementation notices that the first argument is not of type InvocationHandler, it consults the map to identify the replacement method and calls it through the JNI API.

Because the transition to the Proxy dispatcher function is performed by the runtime, exception propagation is correctly set up in this solution. It is worth noting that this approach does not work for native methods, but these cases can be solved by selectively applying the JNI Hooking approach described above. Since native methods already are provided with a correct JNI trampoline by the runtime, the exception handling issues of JNI Hooking do not apply to them. Another limitation of this approach is that static methods cannot be hooked, which usually is not a big issue, since static methods do not suffer from the inheritance problems described in Basic ART Hooking.

8. Evaluation

8.1. Compatibility

Since the internal implementation details of the ART runtime and compilers differ between Android versions and OS variations from different manufacturers, a proof-of-concept project like the one developed as part of this research project cannot cover all possible configurations. Instead, we decided to focus on several recent OS releases for the popular Google Pixel series of devices. Specifically, we carried out our research using a Google Pixel 3 running the official Android 10 (Q) and 11 (R) images. Additionally, we confirmed the functionality of our solution on a Xiaomi Mi A1 running an official build of Android 9 (Pie).

We integrated the described ART hooking procedures into a custom automated application patching pipeline that is currently under development, so that we could test the compatibility against popular applications from the Google Play Store. These experiments have shown that the hybrid solution of Proxy Hooking and JNI Hooking works reliably across a broad range of applications. Where we discovered application crashes, they happened as well when utilizing static patching, so were not caused by the runtime manipulations.

8.2. Performance

Inevitably, the procedures described above have some impact on the runtime performance of the target application, simply because of the fact that more instructions have to be executed than in the original method implementation. For Native Hooking and Basic ART Hooking, the overhead is only a few instructions, so should be completely negligible. For JNI Hooking and Proxy Hooking, the overhead is a little higher due to the added cost of transitions between JNI and managed code, but still unnoticeable to end users in our experience.

Another caveat worth mentioning is that AOT compilation is effectively disabled for hooked methods. This means that if a particularly hot method is hooked, it still has to be compiled anew for every application launch.

8.3. Use Cases

The core principles developed as part of this research provide a great starting ground for further research as well as deployment in various real-world scenarios. In order to provide some insights into the practicality of the solution, here are a few examples:

- **Automated On-Device Patching of Arbitrary Android Applications** When the powerful runtime manipulation techniques detailed in this report are integrated into a fully-featured but lightweight pipeline for application patching, a solution can be developed that runs as a background service on the device and automatically applies patches to installed application packages. Through additional hooks in the ART runtime, method hooking can be delayed until the corresponding class is loaded, which reduces the overhead at application startup.
- **Mitigating security vulnerabilities caused by crypto-API misuse** As mentioned in the introduction to this report, Android applications still frequently make trivial mistakes in their use of cryptographic primitives provided by the Android OS. As a result, these applications fail to meet the claimed security premises, risking the leak of sensitive user data to attackers. The dynamic patching techniques detailed above provide an efficient tool for hooking relevant crypto APIs, so that e.g., their incorrect use can be detected and mitigated.
- **Improving Android file system security** Recent versions of the Android OS route all file system operations (even those that originate from the plain Java File API) through the Storage Access Framework (SAF), which implements more fine-grained control over access permissions. By carefully hooking the right APIs, the same improvements could be backported to older Android versions, without having to modify the OS installation itself.
- **System for customizing third-party applications through user-installable modules** A module system similar to the once popular but fading Xposed [10] framework could be devised, that has users install patches to a central patch repository, which are then deployed to all installed applications automatically. Since all modifications are contained within the target applications themselves, no modifications to the OS (such as rooting) would be needed. This crucial difference to the original Xposed framework could lower the entry barrier and open the concept to a lot more users.

9. Conclusion

This report provided an overview of the results of our research project on patching Android applications through manipulating the internal data structures of the ART runtime. We motivated our research with recent findings in the domain of Mobile Security and provided some background on the system structure for execution of third-party applications on the Android OS. After 10 Xposed: https://github.com/rovo89/Xposed introducing the relevant internals of the Android runtime, we presented 4 different hooking techniques, discussed their respective strengths and weaknesses and devised a hybrid scheme that demonstrated the best reliability in our experiments. We continued by evaluating this solution in terms of compatibility and performance, and stressed its versatility for various different applications before concluding the report.

10. Bibliography

[1] M. Egele, D. Brumley, Y. Fratantonio and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, New York, 2013.

[2] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes and S. Fahl, "Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[3] M. Zhang and H. Yin, "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications," in *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14}*, 2014.

[4] S. Ma, D. Lo, L. Teng and R. H. Deng, "CDRep: Automatic Repair of Cryptographic Misuses in Android Applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, Xi'an, China, 2016.

[5] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben and M. Smith, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, Raleigh, North Carolina, USA, 2012.

[6] H. C. Benjamin Davis, "RetroSkeleton: Retrofitting Android Apps," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, Taipei, Taiwan, 2013.

[7] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler and A. Alkhelaifi, "Securing SSL Certificate Verification through Dynamic Linking," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, Scottsdale, Arizona, USA, 2014.