A2P2 - An Android Application Patching Pipeline Based On Generic Changesets

Florian Draschbacher
florian.draschbacher@iaik.tugraz.at
Graz University of Technology and Secure Information Technology Center Austria
Graz, Austria

ABSTRACT

Inspecting and manipulating runtime behavior of Android applications is a common need in mobile security research. However, existing tools lack a holistic application-agnostic approach. They either require changes to be manually adapted to each target application, or they focus exclusively on executable code parts, neglecting the key role the application manifest and resources play in the Android ecosystem. This limits their use for research purposes, where a specific series of modifications on various app components frequently has to be applied to a whole body of applications.

In this paper, we present A2P2, a flexible patching pipeline for compiled Android applications. Our system encompasses a custom declarative patch format for specifying complex manipulations on all parts of an application package. Patch projects are developed inside the Android Studio IDE and compiled into patch packages. These may then be applied to an arbitrary number of application package (APK) files through our flexible patching pipeline implementation. Existing pipeline stages may be freely arranged and augmented with user-supplied custom stages so that entirely new sophisticated transformations may be implemented from a range of core primitives. For manipulating Dalvik bytecode, we provide two different rewriting backends and an abstraction that enables addition of new rewriting technologies transparently to patch projects.

We demonstrate A2P2's efficiency and efficacy by providing estimates for deployment speed and effects on compatibility, application size, and runtime performance for typical use cases. Lastly, we implement A2P2 patches that reproduce previous research and facilitate common security analysis tasks.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; • **Software and its engineering → Application specific development environments**.

ACM Reference Format:
Florian Draschbacher. 2023. A2P2 - An Android Application Patching Pipeline Based On Generic Changesets. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29–September 01, 2023, Benevento, Italy.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3600160.3600172

This work is licensed under a Creative Commons Attribution International 4.0 License.

ARES 2023, August 29–September 01, 2023, Benevento, Italy
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0772-8/23/08.
https://doi.org/10.1145/3600160.3600172

1 INTRODUCTION

As part of their work, mobile security researchers commonly face the need to inspect and manipulate the runtime behavior of closed-source Android applications. This may be necessary for analyzing a suspectedly malicious program, reverse-engineering a proprietary protocol, assessing the security of an unknown piece of software, or detecting and mitigating a particular class of vulnerabilities.

Over the past decade, a number of solutions for inspecting and manipulating application runtime behavior have been proposed. From these, three different technical approaches can be derived, each with its own advantages and disadvantages.

(1) *System Modifications.* Root privileges can be utilized on the Android platform for either replacing parts of the system components that form the application runtime [7], or for attaching to an application process during execution [5]. However, gaining root privileges requires modifying the OS installation, which impacts the entire software stack on the device and not just individual applications. Additionally, application patching tools proposed as part of research are often designed for aiding inexperienced developers or advanced users in retrofitting security improvements into compiled apps. In these scenarios, the resulting apps are installed on real-world devices, where making root permissions available to the user must be considered a security risk.
(2) *Virtual Containers.* Some solutions execute target applications within the context of a container application, e.g. [3]. From the perspective of the OS, the target application lives in the process of the container application, which grants it arbitrary capabilities for runtime manipulation without requiring root permissions. However, this setup also considerably limits the OS integration of the target application. Additionally, it requires the container application to statically pre-request all permissions any contained application may need later, which qualifies as an extreme case of over-permissioning.
(3) *Repackaging.* Applying changes directly onto the application package (APK) file [13, 15] allows modifying any aspect of the contained files, such as the executable code or the application manifest. All changes only affect the target application and can be conveniently redistributed for installation on unmodified Android systems. However, modifying the APK file requires resigning it, which may lead to issues. Still, in research scenarios where root privileges are not available, repackaging is the most reliable way for inspecting and manipulating Android application runtime behavior. We therefore concentrate on this technology in the following.

Although multiple repackaging tools are readily available for applying user-specified changes to a given APK file, their use for research purposes is limited. Many research use cases involve applying a given change to a whole body of diverse applications. However, available tools either require changes to be manually adapted to each individual target APK file or only allow modifying the executable parts of the APK, entirely neglecting the central role of the Android manifest as the contract between the application and OS. Due to this lack of existing tooling, researchers commonly had to implement their own purpose-built tools [4, 8, 11, 16], wasting resources that they may instead have invested into their core contributions.

We introduce A2P2 to address this evident gap in research tooling. To the best of our knowledge, our patching pipeline is the first solution for developing and deploying holistic application-agnostic patches to compiled Android applications.

Our key contributions are:

- We propose an application-agnostic patch format that allows declaratively specifying complex changesets to Dalvik bytecode, Android manifest and application resources, as well as addition of arbitrary files into a compiled Android application package. The format entails a developer-facing source variant (*A2P2 patch project*) as well as a compiled binary variant (*A2P2 patch package*) for feeding into the deployment pipeline.
- We present a suite of tools for developing and compiling changesets, as well as for applying them to concrete compiled Android application packages (APK files). The deployment components are organised in a fully customizable patch deployment pipeline. We provide full implementations of all pipeline stages needed for deploying our patch format. Additionally, our solution allows users to supply custom pipeline stages that may specify complementary changes procedurally, utilizing our low-level manipulation primitives for various Android-specific file formats.
- We provide performance and compatibility metrics for evaluating A2P2's efficacy and showcase a set of example patches that demonstrate how our patching pipeline facilitates mobile security research.
- We make our entire toolchain (including development tools and deployment pipeline) available to the research community under an open-source license. [1]

The remainder of this paper is organized as follows: Section 2 lays out the necessary background knowledge. Section 3 then introduces the overall architecture of the A2P2 system. Subsequent Sections 4 and 5 describe our patch format design and pipeline implementation, which are then evaluated in Section 6. We discuss limitations of our concept and plans for future work in Section 7, highlight related publications in Section 8 and conclude this paper in Section 9.

2 BACKGROUND

In this section, we summarize background information on Android application development, the APK format, and the ART runtime.

[1] Source code is available at https://extgit.iaik.tugraz.at/fdraschbacher/a2p2

2.1 Android Application Development Process

Google provides an official Android Studio IDE for developing Android applications. Key components of an application project are:

- *Android manifest*: An XML file declaring the permissions required for operation and the functionality (Activities, Services, ContentProviders) exposed to the rest of system. If any such externally exposed characteristic is to be added to or removed from the application, the change needs to be reflected in the manifest for the system to pick it up.
- *Program code*: Google recommends implementing program logic in Java or Kotlin, although integrating native code is possible for performance-critical program parts or incorporation of existing components. The Java Native Interface (JNI) enables interaction between native (C/C++) and managed (Java/Kotlin) code.
- *Resources*: These include files that define UI layout structures, strings for localisations, images and many more.
- *Other assets*: Files that are copied into the application package in verbatim during build, i.e. in an unaltered form.

During build, various human-readable parts of an application project are transformed into representations more suitable for execution and packed into an Android Application Package (APK) file. The Gradle build system and an Android-specific plugin are used for coordinating and configuring the tasks involved in the build process. Java and Kotlin code is compiled into Java bytecode using the javac compiler, then further transformed into Android-specific Dalvik bytecode by the d8 build tool. The Android manifest and XML resources are compressed into an Android-specific binary XML representation. A resource index is assembled that assigns each resource an identifier that can be used for cross-referencing.

2.2 Android Package Format

The APK file format is used for deploying a compiled Android application to a device for installation. At the outermost layer, an APK file is a ZIP archive whose contents follow a well-defined structure. The core components of an APK file are stored in the top level of the ZIP container. These comprise the Android manifest in the AXML binary XML format and one or multiple Dalvik Executable (DEX) files that hold the Dalvik bytecode for the Java and Kotlin classes that implement the program logic. The ARSC file plays a key role in organizing the resources within an APK file. It contains a detailed resource index that not only allows unique identification of resources but also is structured in a way that enables dynamically selecting resource values at runtime depending on device characteristics such as screen size or configured locale.

Before an APK file can be installed onto an Android device, it needs to be signed with a self-signed developer certificate. This ensures that application updates can only be installed if they stem from the same developer as the original installation.

For efficiency reasons, modern Android applications may be composed of multiple APK files that carry the same package name and are signed with the same developer certificate. All individual APK files need to be installed onto the device for the application to be operational.

2.3 Android Runtime

The Dalvik bytecode stored in DEX files encodes program functionality in the instruction set of a register-based virtual machine. This ensures a compiled application is compatible with any Android device, no matter its native CPU architecture. The Android Runtime (ART) is the system component responsible for implementing the virtual machine instruction set. It executes Dalvik bytecode through a combination of interpretation, ahead-of-time (AOT) compilation, and just-in-time (JIT) compilation. Instead of initializing a fresh runtime instance at every application launch, application processes are forked from a special process called Zygote that already pre-initialized the most frequently used framework classes. A list of these classes may be obtained from the publicly-readable file at `/system/etc/preloaded-classes` on a device.

3 A2P2 OVERVIEW

In this section, we describe the core goals and concepts of the A2P2 system from a high-level perspective.

Most fundamentally, A2P2 seeks to keep the entry barrier for implementing common patch use cases low while still offering the maximum degree of flexibility for advanced patching needs. For maintaining a low entry barrier, we propose a custom declarative patch format and all tooling required for developing and deploying patches to compiled APK files. We accomplish flexibility by structuring the deployment functionality as a fully extensible patching pipeline that offers the framework and primitives for efficiently implementing arbitrarily complex APK transformations. Figure 1 illustrates how a patch in the A2P2 format can be deployed by configuring and executing a suitable A2P2 pipeline instance.

3.1 Declarative Patch Format

Our declarative patch format is designed for specifying changes in a generic form once and deploying them to an arbitrary number of application packages later. We accomplish this by focusing supported changes on the interface between the platform and application. Fundamentally, this entails the Android manifest, where an app registers its key components to the system and declares the permissions it requires for its operation. Our custom XML patch format supports specifying arbitrary modifications to manifest contents, including addition, deletion, and modification of element or attribute nodes. Second, the A2P2 patch format also allows addressing the APIs that form the functional interface between the OS and an application. An annotation-based domain-specific language enables precisely intercepting method invocations to system framework or third-party library APIs. Replacement methods are implemented in Java, may manipulate or inspect passed parameters and invoke the original method implementation. Furthermore, patches may specify resources for addition into target APK files' resource index in a way that does not collide with existing resource identifiers while still allowing to reference them in the patch project's code or application manifest. Lastly, our patch format also allows provisioning assets or native libraries for addition into target APKs. Before a patch project can be used in the deployment pipeline, it is transformed into a patch package, which contains machine-readable representations of all changes that already resemble the respective target structures inside APK files.

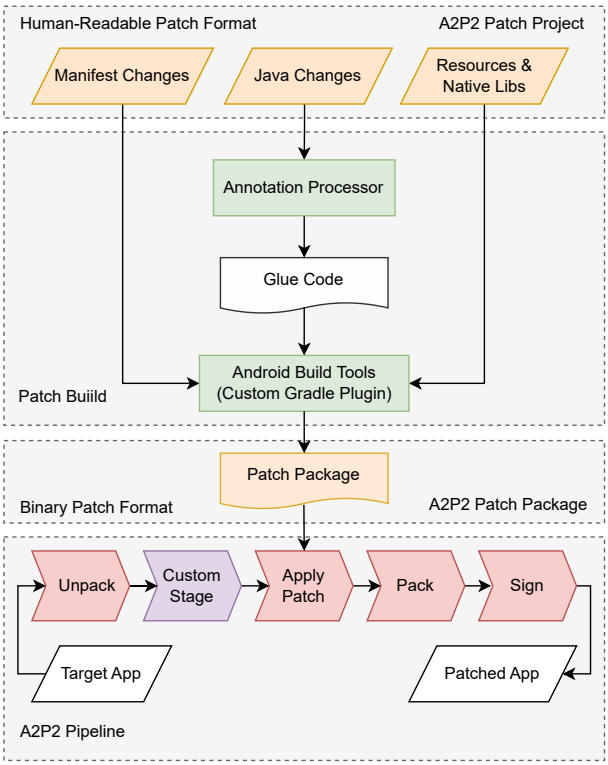

Figure 1: Core components of the A2P2 system: Patches in the A2P2 patch format can be deployed by configuring and executing a pipeline instance that uses the Apply Patch stage.

3.2 Development Tooling & Patching Pipeline

A2P2 includes all tooling required for building patch packages, as well as for deploying them into application package (APK) files.

For lowering the entry barrier to A2P2, our *development tooling* integrates into the Android Studio IDE so that patch projects can be constructed in an environment Android researchers and practitioners are already familiar with. To this end, A2P2 provides a custom Gradle plugin that can be used for organizing the structure and build of a patch project. Transforming a patch project into a patch package takes advantage of the official Android build tools normally used for app development. A custom annotation processor integrates into the build process to generate glue code that later facilitates patch deployment.

Deployment tooling consists of a flexible pipeline design. As part of this design, we specify a chainable pipeline stage interface. Every stage implements a transformation on the currently processed target application package, optionally incorporating additional configuration or input files. An arbitrary number of pipeline stages and respective configuration or input files can be arranged into a sequence, forming a pipeline instance. A body of application packages (APK files) can then be filled into the pipeline and pass through the stages one by one. After undergoing the sequence of transformations implemented by the configured pipeline stages, each input package exits the last stage as an APK file again.

Our pipeline design and interface were kept generic so that users can implement and supply custom stages for integration into pipeline instances together with readily made stages shipped

with our A2P2 reference implementation. Among these pre-built stages, we provide essential functionality for unpacking, packing, and signing application packages, as well as an Apply Patch stage for applying patch packages in our declarative patch format.

The Apply Patch pipeline stage takes advantage of primitives for parsing and manipulating AXML, ARSC, and DEX data structures, as well as our respective patch formats. For deploying code changes, A2P2 supports two rewriting backends that differ in compatibility characteristics.

4 PATCH FORMAT

This section describes our declarative patch format, which comprises the human-readable source variant we term *A2P2 Patch Project*, as well as *A2P2 Patch Package*, a binary variant that facilitates deployment.

The A2P2 patch format was designed to be application-agnostic yet easy to use. To this end, our patch project and patch package structures as closely as possible follow those of application projects and application packages, respectively. Changes to a given application component (e.g., the application manifest, code, or resources) are specified in the same general file format as their respective target. For example, changes to the application manifest XML file are themselves specified inside an XML file. In the following, we describe how patch projects can specify changes to the different components of an Android application package (APK file).

4.1 Application Manifest Changes

Changes to the application manifest are specified in the `AndroidManifest.xml` file of the patch project, which we term manifest patch file. While the manifest patch file would pass validation as an application manifest, it contains custom XML elements that specify a sequence of transformations to be applied to a target APK's application manifest file during patch deployment. The available XML elements are loosely based on RFC5261 [14] but adapted for the Android-specific AXML format and enhanced by XPath placeholders in added or replaced values.

Every transformation element encodes a combination of a selector and an operation. The selector is an XPath expression that specifies the precise subject or subjects (one or multiple element, attribute or text nodes) of the modification. Available operations are addition, replacement, or insertion of elements or attributes. Listing 1 showcases an example manifest patch file taking advantage of all supported transformation elements.

4.1.1 Add Element. Addition of new elements or attributes can be accomplished using the `<add sel="..." pos="..."/>` element in the manifest patch file. The `sel` attribute contains the selector expression, while the optional `pos` attribute encodes where exactly in relation to the selected node or its child nodes the addition should be applied. Possible values are `prepend`, `before`, and `after`. All child elements of the `add` node in the patch manifest file are added as children to target nodes. If no child elements are provided, all attributes of the `add` node are copied to selected target nodes.

4.1.2 Replace Element. The `<replace sel="..."/>` element allows replacing elements or attribute values. If its selector in the `sel` attribute targets an element, the latter will be replaced with the first child node of the `replace` element, including all attributes and children. If the XPath selector targets attributes, the replacement value is to be provided as the text content of the `replace` element.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.
↪ com/apk/res/android"
↪ xmlns:patch="http://schemas.android.
↪ com/apk/res-auto"
    package="com.a2p2.sample.patch">

  <patch:add sel="manifest">
    <metadata android:name="patched"
    ↪ android:value="true" />
  </patch:add>

  <patch:replace
  ↪ sel="manifest/application/activity/@name">
    $${globalize(xpath("/manifest/@package"),
    ↪ xpath("."))}
  </patch:replace>

  <patch:remove
  ↪ sel="manifest/application/@testOnly"/>
</manifest>
```

Listing 1: Example manifest patch file

4.1.3 Remove Element. Lastly, elements or attributes may be removed by utilizing the `<remove sel="..."/>` element.

4.1.4 Placeholder Expressions. Added values may include placeholder expressions. These are string tokens prefixed with `$$` and surrounded by curly braces. The `$${xpath(...)}` placeholder allows including results of XPath expressions evaluated in relation to the replaced element. The `$${globalize(...,...)}` placeholder expression combines a relative class name (first parameter) and its package name (second parameter) into a fully-qualified class name. `$${appendeach(...,...)}` appends the string passed as its second parameter to every element in the semicolon-delimited list of strings in the first parameter. Placeholder expressions may be nested, enabling very complex combined expressions.

4.2 Java Execution Flow Changes

Fundamentally, A2P2's execution flow manipulation operates by intercepting method calls. Since this scheme is particularly suitable for manipulating the interaction between an application and Android framework APIs, it caters to our philosophy of application-agnostic patching. Only static methods, instance methods, and constructors that are publicly visible may be intercepted.

For every target method intercepted, patch developers implement an interception handler. Inside a patched application, this interception handler will be called in place of the target method. In a patch project, interception handlers are implemented as static Java methods that accept the same arguments, return the same type and throw the same exceptions as the target method. Every

interception handler needs to be marked with an annotation that encodes the target class and method it wishes to intercept. For convenience, multiple interception handlers for methods of the same target class may be combined into one Java class, in which case the target class annotation may be used at class level and is inherited by all contained interception handlers. Listing 2 takes advantage of this feature and demonstrates all possible interception annotations.

In addition to interception handler implementations, patches may contain arbitrary Java or Kotlin code. This is necessary for use cases where patches need to augment existing app functionality, e.g. by adding core components such as activities or services. These can be injected into the application manifest through the manifest patch file and backed by an implementation in the patch's code. All code is compiled into DEX files during patch build, which are to be copied into target application packages during patch deployment.

Whenever an application wishes to call the original implementation of an intercepted method, it can take advantage of the `OriginalMethods` class generated by A2P2's build tools. This class exposes a function for every intercepted method. How exactly these exposed functions implement the invocation of original method implementations depends on the employed rewriting backend (see Section 5.3). This abstraction enables adding arbitrary rewriting backends without modifying patch projects.

4.2.1 Intercepting Static Methods. For intercepting a static method, the corresponding interception handler needs to be marked with the `@PatchStaticMethod` annotation. The name of the target method is either assumed to be identical of the interception handler or passed explicitly as an argument to the annotation. Interception handlers for static methods may call the original implementation of the target method through the `OriginalMethods` abstraction.

4.2.2 Intercepting Constructors. The `@PatchConstructor` annotation encodes interception handlers for constructor invocations. Since Java identifies constructors solely by their signature, no additional name parameter or naming convention is necessary.

Due to specifics of the Dalvik bytecode format and ART runtime which are further discussed in Section 5.3.1, constructor interception handlers do not have a means to explicitly call the original constructor implementation.

4.2.3 Intercepting Instance Methods. Interception handlers for instance methods may be marked with the `@PatchInstanceMethod` annotation. The same name rules and original method invocation mechanism apply as for static method interception.

A particularity of instance methods in the Dalvik bytecode format is that they carry the instance object as an implicit first argument. In the Java and Kotlin programming languages, this first argument is accessible by using the `this` keyword. Since that keyword may only be used in instance methods, static interception handlers for instance methods take the instance object as their explicit first argument.Code inside interception handlers may use this instance object for accessing member variables, or pass it to the `OriginalMethods` abstraction for invoking original method implementations.

4.2.4 Patch Inference. An important detail about the semantics of instance method interception requires further discussion. In many Android framework APIs, only the abstract type of objects passed from the system to the application is known. This affects arguments that the framework passes when invoking an application's lifecycle or callback methods, as well as results returned from framework methods.

For example, consider the `Activity.getContext()` method. Its return value is only guaranteed to extend from the abstract `Context` class. The exact concrete type of the returned object is not known before runtime and may in fact differ between applications. This potentially leads to problems if a patch wishes to intercept an instance method of the `Context` class. In most cases where the exact instantiation type of abstract framework types and interfaces is not known, it is desirable to intercept calls to all corresponding implementations in subclasses. Consider a patch that wishes to intercept calls to the abstract `Context.getSystemService()` method. In the absence of knowledge about the possible concrete implementations of this class (in the system framework or in the code of target applications), the most sensible approach for the patch is to intercept all implementations in all subclasses of the `Context` class.

This consideration plays an important role in the semantics of our execution flow patches. For maintaining application-agnosticity, we require all instance method patches to target those classes in the inheritance tree that first define the target method. Our build and deployment tools take care of enforcing this requirement and inferring patches to all possible subclasses both in framework and application code. These semantics do not have any practical consequence on the expressiveness of our Java patch format. Interception handlers that wish to only target specific subclass implementations may simply check the runtime type of the instance object and use the `OriginalMethods` interface for invoking the original implementation for all types they are not interested in.

4.3 Adding Resources

Resources in the patch project will be copied into any target application package the compiled patch package is applied to. Different resource types and configurations are fully supported. Cross-references between resources, as well as resource references in patch code (interception handler or otherwise) or the manifest patch file, may be used in patch projects and remain functional through patch deployment.

4.3.1 Avoiding resource identifier conflicts. As described in Section 2, an index of all resources in an Android application project is created during compilation and stored in the ARSC file inside the APK package. As part of the indexing process, the aapt2 build tool assigns a unique integer identifier to each resource. Resources of the same type receive consecutive identifiers. For cross-referencing, Android's build tools automatically generate a Java class that maps between a human-readable resource file name and its integer identifier. The mapping is implemented by exposing a final static integer field for each resource, named after its file name and hardcoded to its integer identifier. Developers may thus conveniently reference resource identifiers through their corresponding fields in the generated Java class. During compilation, all final static integer references are inlined, which means that the convenience added by this approach does not have any runtime costs. Since inlined resource identifiers inside Dalvik bytecode can not reliably be distinguished from arbitrary hardcoded integer values, this scheme

```java
@PatchClass("java.net.Socket")
public class SocketSamplePatch {
  // Intercept Socket creation
  @PatchConstructor
  public static void init(Socket thiz, InetAddress
  ↪  address, int port) throws IOException {
    try {
      thiz.setSendBufferSize(4096);
    } catch (SocketException ignored) {}
  }

  // Intercept calls to Socket.setSocketImplFactory
  @PatchStaticMethod("setSocketImplFactory")
  public static void
  ↪  interceptionHandler(SocketImplFactory fac)
  ↪  throws IOException {
  }

  // Intercept calls to instance method of same name
  @PatchInstanceMethod
  public static boolean isConnected(Socket thiz) {
    return OriginalMethods.java_net_Socket.」
    ↪  isConnected(thiz) || true;
  }
}
```

Listing 2: Example execution flow patch

impedes changing resource identifiers in compiled application packages. However, given the predictable nature of resource identifiers and the fact that the ARSC index for patch packages is assembled by the standard aapt2 tool, identifier conflicts between resources in the application package and the patch package are to be expected.

To avoid this problem, the A2P2 patch package format takes advantage of a namespacing mechanism Android normally uses to distinguish between application and system resources . We alter the ARSC data structure inside patch packages to use a package identifier (`0x8f`) that differs from that used in applications (`0x7f`). Since the package identifier constitutes the higher-most byte of every resource identifier, this effectively prevents collisions.

4.4 Additional assets and native libraries

Patch packages may contain additional asset files or native libraries. During patch deployment, these are copied to locations in the target application package that correspond to their locations in the patch package. It is worth noting that files in the patch package overwrite equally named files in the application package. If replacement is not desired, patch developers must take care to choose names and locations that are unlikely to occur in application packages.

5 A2P2 DEVELOPMENT AND DEPLOYMENT TOOLS

In this section, we discuss the tools we implemented to support our A2P2 patch design. The A2P2 distribution comprises all software required for developing patch projects and compiling them into patch packages, as well as for applying patch packages to application packages.

5.1 Development Tools

A2P2 patch projects are developed inside the Android Studio IDE. The software components required for adapting the IDE to our purposes are implemented in a custom plugin for the Gradle build system and an annotation processor.

5.1.1 A2P2 Gradle Plugin. Since the A2P2 patch project and patch package formats closely resemble the respective Android application formats, our Gradle plugin extends the standard Android Gradle plugin. It integrates our annotation processor into the build process, configures the non-standard aapt2 resource namespace, and exposes new build tasks for the patch package. Furthermore, it automatically adds dependencies to our patch base libraries, which contain definitions for patch annotations, as well as parts of the rewriting backend logic.

5.1.2 Annotation Processor. The A2P2 annotation processor (AP) holds a key role in the build process of an A2P2 patch project. It validates execution flow patches, implements parts of the inference logic for execution flow patches, and generates glue code for different rewriting backends.

For validating execution flow patches, the AP parses the classpath dependencies of the patch project, in particular the `android.jar` file included in the Android SDK distribution. This file contains stub implementations of all classes and interfaces in public Android APIs. The AP looks up the target methods of interception handlers in the classpath and compares the declared parameters, return types, and thrown exceptions. If a specified target method could not be found or a mismatch in the signature is detected, the AP aborts the build. It thus helps prevent bugs in patches that would otherwise later manifest in obscure runtime issues after patch deployment.

Inference of execution flow patches in the AP works by constructing a class inheritance hierarchy from the patch project's classpath dependencies. Using this information, the AP generates additional annotations for consumption during patch deployment.

The glue code generated by the AP is needed for abstracting away the implementation details of different rewriting backends, as well as for facilitating later patch deployment. Generated code parts constitute wrappers around interception handlers, as well as implementations of the `OriginalMethods` class for invoking original method implementations.

5.2 Deployment Tools

A2P2's deployment functionality is organized in a flexible transformation pipeline. Our implementation in the Java programming language may be used as a standalone command-line tool or integrated into other projects as a Java library. The logic for applying our declarative patch format is implemented in a pipeline stage. Other provided stages implement functionality for unpacking, packing, and signing application packages, or adding asset files. All stages follow a generic pipeline interface that facilitates the development of custom stages as well as extending existing stages.

The general procedure including pipeline instantiation and execution is depicted in Figure 2. We call the entirety of pre-installed

```
$java -jar a2p2.jar file1.apk ./folder_of_APKs !
↪  unpack ! apply patch_static.zip static !
↪  custom_stage ! pack ! sign ! ./output
```

Listing 3: Example A2P2 pipeline invocation

and custom pipeline stages the stage repository. A pipeline instance is assembled as a parameterized sequence of stages from this stage repository. The pipeline instance is then executed on a concrete set of APK files. Every stage operates on a pipeline context, which represents the output of the previous stage in the execution. The result of a successful execution of the pipeline instance is a set of transformed output APK files.

5.2.1 Pipeline Implementation. The command line interface to our pipeline implementation allows configuring pipeline instances and executing them on a specific body of applications in one invocation. An example invocation is shown in Listing 3. Individual stages are specified by their name and separated using an exclamation mark character. Additional space-delimited arguments may be passed to each stage. A list of all known stages (the stage repository) and their supported arguments may be obtained by invoking the command line interface without any parameters. Custom stages may be added to the stage repository as jar files installed to a specific subdirectory of the A2P2 pipeline distribution.

5.2.2 Apply Patch Stage. The logic required for applying a patch package to an application package is implemented in the Apply Patch stage. Besides the mandatory path to a patch package, the user may optionally specify the rewriting backend. By default, static rewriting is employed.

5.2.3 Custom Stages. Pipeline instances may include custom stages implemented by third parties. Every stage must extend from our abstract `Stage` class, which provisions functionality for querying stage metadata (such as its name), configuring a concrete stage instance, and executing its transformation on a pipeline context. Custom stages may take advantage of A2P2's utility classes for replacing or inserting instructions in method implementations inside Dalvik bytecode, manipulating binary AXML files, or altering ARSC files.

5.3 Rewriting Backends

Our build and deployment infrastructure supports two different rewriting backends for applying execution flow patches to DEX files. Static rewriting bakes all changes directly into DEX files, which means that the runtime overhead of interception is minimal, and the resulting patched application package is fully compatible with any device that supported the original APK. Dynamic rewriting injects a native library into application packages that applies code manipulations at runtime. In contrast to static rewriting, this approach supports intercepting invocations through Reflection or JNI but relies on implementation details of the ART runtime that differ between devices, CPU architectures and ART versions.

A2P2 patch projects may be developed independently of the rewriting backend that is later used for deployment. Still, compiled patch packages in our current implementation are specific to a rewriting backend. During patch deployment, the user has to

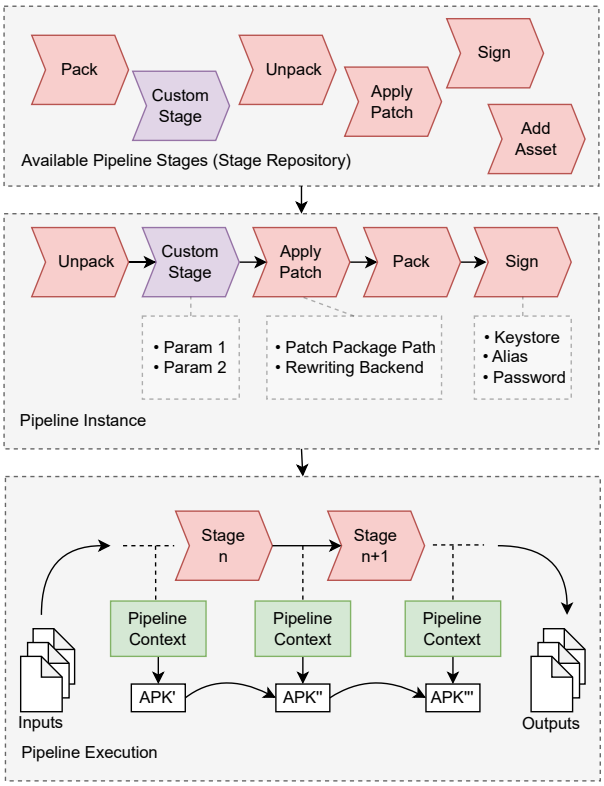

Figure 2: Pipeline instantiation and execution

ensure the supplied patch package is coherent with the specified rewriting backend. The A2P2 build tools and deployment pipeline were designed so that new rewriting backends may be incorporated without changing any code in patch projects.

5.3.1 Static Rewriting. The static rewriting implementation scans Dalvik bytecode for invocation instructions, replacing references to methods targeted by execution flow patches with references to the corresponding interception handler.

For constructor interception, the call to the interception handler is injected as an additional instruction that does not replace the call to the constructor but immediately follows it. This is necessary because the Dalvik bytecode specification does not permit passing an allocated object to another method before its constructor has been called. Since the added interception handler invocation changes the addresses of subsequent instructions, our rewriting backend updates all offsets and ranges in affected try/catch or switch/case blocks, goto or jump instructions, and debug items.

Patch inference for classes implemented in application code works by scanning the app's DEX files for subclasses of all classes targeted by interception handlers. It is worth pointing out that super calls (invoking the super class's implementation of the currently executed method) are ignored in static rewriting. This is because the corresponding `invoke-super` instruction may only be used inside of class context, which means that the originally called method would not have been accessible to interception handlers. In practice, since our patch format is designed to focus on the interface between application and framework code, this limitation doesn't have any

negative consequences on patch developers. Our static rewriting implementation builds upon low-level DEX manipulation primitives of the open-source dexlib2[2] library.

5.3.2 Dynamic Rewriting. Dynamic rewriting applies execution flow changes at runtime. Through a high-priority `ContentProvider` injected into target application packages, we can manipulate the ART runtime before any third-party code is executed. The `ContentProvider` parses annotation data from the patch DEX file to assemble a list of methods that need to be intercepted. It then iterates over all classes preloaded by the Zygote process to intercept all targeted methods they implement. Lastly, we instrument the ART runtime's class loading mechanism to intercept targeted methods for all classes as they are loaded. This also provides an opportunity for inferring patches onto subclasses.

Method interception is implemented by manipulating the JIT-compiled in-memory representation of target methods. Their preamble is replaced by a trampoline to the interception handler. The original preamble is copied to a freshly allocated executable memory region, followed by a jump to the end of its original location, and executed whenever the original method implementation is invoked through the `OriginalMethods` interface.

Because of patch inference, an interception handler for an instance method may be invoked for various different concrete overridden implementations of its target method. This leads to potential issues in the `OriginalMethods` implementations for dynamic rewriting, where the exact originally referenced method needs to be known for being able to call it. Due to potentially involved super calls (which cannot be distinguished from normal method calls), the object type alone does not suffice for identifying the originally called method. As a solution to this problem, our dynamic rewriting backend takes advantage of internal mechanisms of the `java.lang.reflect.Proxy` infrastructure for dynamically generating a distinct trampoline method for every infered method interception. These trampoline methods act as breadcrumbs in the call stack, which we combine with a lookup table for determining the originally called method in `OriginalMethods` implementations.

Our dynamic rewriting implementation integrates modified low-level primitives of the open-source SandHook[3] library.

6 EVALUATION

In this section, we provide patch case studies that demonstrate the efficacy of A2P2 for facilitating mobile security research, as well as performance and overhead metrics for evaluating the efficiency of our solution.

6.1 Patch Case Studies

The patches we showcase here were chosen to demonstrate the capabilities of our patch and pipeline design and reflect typical use cases in mobile security research.

6.1.1 Grab'n'Run. We used A2P2 to reimplement the purpose-built application-rewriting tool proposed by Falsina et al. [8] for injecting their novel verification protocol for dynamic code loading into compiled applications. Falsina et al.'s implementation involves almost 1000 lines of Python code for manually manipulating the SMALI IR representation of decompiled Dalvik bytecode and adding permissions to the application manifest. In contrast, the A2P2 patch can be fully implemented using our declarative patch format, which means that all changes are specified in high-level Java and XML. The key part of the Java execution flow patch can be found in Listing 4 in the Appendix. It intercepts class loading through the `DexClassLoader` class to verify the code that is about to be loaded. Note how the patch intercepts the constructor to create shadow objects that are later looked up in the interception handler for the instance method. This pattern effectively allows replacing object types. The `loadClass()` interception handler also displays how changes to the execution flow may be applied selectively based on the concrete instance type. While our patch implementation uses the default A2P2 pipeline command line interface for deployment, a more convenient custom Java program could easily be written that integrates the pipeline as a library.

6.1.2 Cloning applications. In some scenarios, an app needs to be installed on a device twice, e.g. when it comes preinstalled to the device and thus cannot simply be uninstalled for replacement with a patched version. The Android OS requires the application package name and certain app components to be unique on the system, so that the same APK file cannot be installed twice. While for simple applications, changing the package name in the application manifest is enough to create an APK file that can be installed alongside the original version, more modifications are needed in general.

We implemented an A2P2 patch project that changes the package name, custom permissions, and `ContentProvider` authorities in the application manifest. A part of the patch manifest of this project can be found in Listing 5 in the Appendix. It shows how A2P2's XML patch format and XPath expressions may be used for changing the package name and various related values inside an app's manifest. Note how the actual package name change is the last entry in the patch manifest. Since patch manifest entries are applied sequentially, this allows earlier entries to reference the original package name. This is e.g. used for globalising class name references that were constructed relative to the package name in the original manifest. The Java part of the patch project intercepts various framework API calls so that the original package name is spoofed to all application-facing code while system- and world-facing code is aware of the manipulated package name. This even entails package names sent to backend servers in HTTP request headers. We complemented our patch project with a custom pipeline stage that also replaces the account type for app-specific authenticators, which are required to be system-unique as well.

6.1.3 Injecting Flipper debugger. Flipper is an open-source mobile application debugging platform maintained by Facebook. Developers may integrate plugins into their Android application project that communicate with the Flipper desktop companion program. Flipper offers plugins for inspecting files in app-private storage, databases, UI layouts, or network communication. While originally intended for debugging during application development, the platform may also serve as a security analysis tool.

We designed an A2P2 patch that injects the base Flipper library and several plugins into target applications. A `ContentProvider`

[2] dexlib2: https://github.com/JesusFreke/smali/tree/master/dexlib2
[3] SandHook Android ART Hook: https://github.com/asLody/SandHook

added through the manifest patch initializes plugins at runtime. Several execution flow patches are used for integrating the networking plugin into the OkHttp stack within target applications.

6.2 Performance and Overheads

Quantifiable performance and overhead metrics include the patch deployment duration, as well as the size and runtime overhead incurred by applying a patch to an application package. It is worth noting that all metrics shown reflect a pipeline configuration for applying a patch package, utilizing the built-in Unpack, Apply Patch, Pack and Sign stages.

6.2.1 Patch Deployment Speed. Patch deployment speed largely depends on the sizes of patch and application packages. As a realistic estimate, we provide measurements for applying our relatively large Flipper patch (see 6.1.3) to the APK file of the Wikipedia Android app (version 2.7.50431) on a 2.3 GHz Intel i7 quad-core CPU.

For dynamic rewriting, patch deployment consists of unzipping the APK file, adding a `ContentProvider` to the application manifest, merging patch resources, extracting the rest of the patch package (including native libraries and DEX files) into the target application package, zipping the resulting APK file and signing it again. For our test setup, this process takes about 12 seconds on average. Static rewriting additionally involves parsing interception handler targets from the DEX file in the patch package and iterating over all instructions in the target application's DEX files. In our test setup, these additional steps resulted in an overall patch deployment time of 21 seconds for static rewriting.

6.2.2 Application Size Overhead. The file size overhead of application packages caused by our patching very closely follows the size of the applied patch package. All patch packages contain support files that amount to about 950 KB for dynamic rewriting and 10 KB for static rewriting. Since the size of patch packages otherwise depends on the contents of the corresponding patch projects, we cannot provide any more specific metrics here.

6.2.3 Application Runtime Overhead. As an indicator for the runtime impact of execution flow patches, we timed the overhead per call to an intercepted method. Additionally, we timed the ART manipulations carried out for every intercepted method at app launch when dynamic rewriting is used.

For all measurements, we used a simple patch that intercepts a constructor, static, and instance method. The interception handlers simply invoke the corresponding original method implementations. We then applied the patch on an application that times the execution of intercepted methods over 100000 runs. Per-call overheads were taken as the average over these runs. Per-method (launch time) overheads were taken as the average time it took to carry out the runtime manipulations required for intercepting one method (or constructor) over 10 app launches.

The per-call overhead induced by static rewriting is almost nonexistent (77 ns at maximum). Dynamic rewriting generates a higher overhead (we measured 43499 ns in the worst case) due to the involved Java Native Interface calls and original implementation lookup. Still, at less than a tenth of a millisecond, this overhead can still be considered negligible in most practical scenarios. Complete measurement data can be found in Table 1.

	Constructor	Static M.	Instance M.
Static Rewriting	77 ns	23 ns	20 ns
Dynamic Rewriting	25733 ns	1779 ns	43499 ns

Table 1: Per-call runtime overhead for different rewriting backends and method types.

	Constructor	Static M.	Instance M.
Dynamic Rewriting	1.46 ms	0.46 ms	0.86 ms

Table 2: Per-method runtime overhead for dynamic rewriting at app launch (no launch overhead for static rewriting).

Static rewriting does not have any per-method overhead at app launch time. For dynamic rewriting, we measured a worst-case overhead (for constructors) of 1.45 milliseconds. Exact data can be found in Table 2.

6.2.4 Application Compatibility. As a measure for the general compatibility of our patching process, we applied the Cloning patch described in Section 6.1.2 to the 132 most popular free applications (4 per category) from Google Play and confirmed that the patched APK could still be successfully installed and launched on a Pixel 3 running Android 11 (Build RQ3A.211001.001). Our results show that for static rewriting, 92 % of applications still launched, while the success rate of dynamic rewriting was 91 %. Patching generated an installable APK file for all applications. Dynamic rewriting caused one app to hang during launch. Other incompatibilities with individual applications that lead to runtime issues across rewriting backends could be attributed to signature checks (3 apps) and side effects of the package name manipulations of the specific patch implementation (9 apps).

7 DISCUSSION & FUTURE WORK

This section discusses known limitations in A2P2's design, as well as possible improvements to the current implementation.

7.1 Resigning

Some applications integrate signature checks designed for preventing malicious repackaging attacks. Unfortunately, these checks also impact application patching for security research purposes. Still, Ibrahim et al. [10] have recently shown that less than 0.1 % of apps employ out-of-process app attestation. All other in-process signature check schemes may be bypassed through application patching, and we argue this is ethically viable for security research. We note that A2P2's execution flow patching may be used for bypassing the most common signature check implementations.

7.2 Malicious Patching

It is worth noting that although we designed A2P2 as a security tool, it may be misused for applying malicious modifications to compiled application packages, i.e. for mounting repackaging attacks. These attacks may e.g. use A2P2 for creating a version of a paid application that bypasses license checks and/or adds code for collecting sensitive user information. However, these types of attacks have been possible before, so we argue that A2P2's publication does not lead to any additional harm on end users. We hope that the availability of Android application patching tools in general encourages more developers of sensitive applications to

adopt out-of-process app attestation, which is the only effective mitigation for repackaging.

7.3 Obfuscation

In principle, A2P2 is capable of not only intercepting framework methods but also those exposed by third-party libraries. To this end, our annotation processor can look up method definitions from any class in the classpath of the patch project. However, implementations of these libraries are embedded in application code and may thus be subjected to obfuscation, i.e. replacing class and method names with undescriptive short strings during build. This technique is commonly used by developers to thwart reverse engineering and optimize APK size. Since our execution flow patches rely on method and class names for identifying target methods, obfuscation may render them ineffective. Future work could investigate how target methods can be identified via obfuscation-resistant selectors.

7.4 Native Code

Our current patch design does not support manipulating the execution flow of native code. However, dynamic rewriting does cover calls from native code to Java through the Java Native Interface.

7.5 Device and OS compatibility

As mentioned in Section 5.3.2, dynamic rewriting relies on implementation details of the ART runtime that differ between devices, OS, and ART versions. We developed and tested our dynamic rewriting backend on a Google Pixel 3 running Android 9, 10, and 11. We anticipate slight adjustments for expanding compatibility to additional device configurations. For devices not yet supported by dynamic rewriting, patch developers may use static rewriting.

7.6 Performance

It is worth noting that our implementation can be further optimized, particularly for patch deployment performance. Possibilities for enhancements would be parallelizing the processing of individual DEX files inside the Apply Patch stage or implementing DEX, ARSC, and AXML manipulation primitives in native code.

8 RELATED WORK

In this section, we highlight previous publications that are concerned with modifying some aspects of compiled Android application packages.

8.1 Modifying APK Files

While some solutions exist for manipulating arbitrary parts of a compiled APK file, none of them share A2P2's application-agnosticity. The most established tool for manipulating APK files both in academia and in industry is Apktool[4]. It is capable of decompiling resources and application manifests back into human-readable XML format and disassemling Dalvik bytecode into the SMALI intermediate representation. Although Apktool supports recompiling APK files after modifications to any of its parts, changes have to be manually applied and adapted to each target application.

[4]Apktool - A tool for reverse engineering Android apk files: https://ibotpeaches.github.io/Apktool/

8.2 Manipulating Dalvik bytecode

A number of publications have proposed generic solutions for modifying *the execution flow* of compiled Android applications. Some of them seek to offer a similar application-agnostic approach as A2P2's patch format.

The existing work that shares the most similarities with A2P2 with respect to execution flow patching is RetroSkeleton by Davis and Chen [6]. It allows intercepting arbitrary methods using transformation policies written in Closure. RetroSkeleton also follows a similar notion of patch inference but only supports bytecode instrumentation, which operates similarly to our static rewriting. The implementation was never made available to the public. Reptor by Ki et al. [12] focuses on API virtualization, for which they go to great lengths to accomplish method implementation replacement instead of call replacement. However, their solution creates boilerplate class hierarchies that lead to considerable APK size overhead and was never made available to the public.

Some tools decompile Dalvik bytecode to Java so they can take advantage of established manipulation tools and techniques for the desktop Java platform. However, this approach is prone to introduce issues since the transformation from Java to Dalvik bytecode during app compilation is not fully reversible in general. SIF by Hao et al. [9] takes advantage of this approach for implementing their instrumentation framework that allows procedural configuration of interception targets. While their system supports complex analysis of the control flow graph, it does not have any notion of patch inference. Arzt et al. [2] and Ali-Gombe et al. [1] apply aspect-oriented programming principles from the desktop Java world to Android applications.

Beside solutions proposed by academia, Frida[5] is a popular choice for instrumenting Android applications both on rooted or unrooted devices. It injects the V8 engine into target applications, so that runtime manipulations can be expressed in JavaScript code. However, Frida operates by manipulating runtime structures. Similar to our dynamic rewriting backend, it is prone to break whenever ART implementation details change.

9 CONCLUSION

Inspecting and manipulating runtime behavior of Android applications is a common need in mobile security research. However, existing solutions for application patching either lack a holistic view of the application package or do not support application-agnostic operation. In this paper, we introduced the A2P2 patch format and deployment pipeline for remediating these limitations in tooling. Our patch format allows declaratively specifying changes to the application manifest and Dalvik bytecode, as well as addition of resources, native libraries, and assets. We discussed the patch semantics for different APK components, the functionality we provide for building application-agnostic patch projects, and our extensible tooling for applying them to compiled Android application packages. For demonstrating the efficiency of our design and implementation, we showed various performance characteristics of our solution. Lastly, we evaluated the efficacy of A2P2 through compatibility tests and by showcasing patches that reproduce previous research and facilitate security analysis.

[5]Frida - Dynamic instrumentation toolkit: https://frida.re

REFERENCES

[1] Aisha I. Ali-Gombe, Irfan Ahmed, I. I. I. Golden G. Richard, and Vassil Roussev. 2016. AspectDroid: Android App Analysis System. In *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy (CODASPY)*.

[2] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Instrumenting Android and Java Applications as Easy as abc. In *Runtime Verification - 4th International Conference (RV)*.

[3] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium*.

[4] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard - Enforcing User Requirements on Android Apps. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference (TACAS)*.

[5] Valerio Costamagna and Cong Zheng. 2016. ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime. In *Proceedings of the 1st International Workshop on Innovations in Mobile Privacy and Security (IMPS)*.

[6] Benjamin Davis and Hao Chen. 2013. RetroSkeleton: retrofitting android apps. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[7] Lukas Dresel, Mykolai Protsenko, and Tilo Müller. 2016. ARTIST: The Android Runtime Instrumentation Toolkit. In *11th International Conference on Availability, Reliability and Security, ARES 2016*.

[8] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. 2015. Grab 'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*.

[9] Hao Hao, Vicky Singh, and Wenliang Du. 2013. On the effectiveness of API-level access control using bytecode rewriting in Android. In *8th ACM Symposium on Information, Computer and Communications Security (ASIA CCS)*.

[10] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. 2021. SafetyNOT: on the usage of the SafetyNet attestation API in Android. In *MobiSys '21: The 19th Annual International Conference on Mobile Systems, Applications, and Services*.

[11] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd D. Millstein. 2012. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*.

[12] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y. Ko, and Lukasz Ziarek. 2017. Reptor: Enabling API Virtualization on Android for Platform Openness. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*.

[13] Jierui Liu, Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2017. InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

[14] Jari Urpalainen. 2008. An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors. RFC 5261.

[15] Philipp Von Styp-Rekowsky, Sebastian Gerling, Michael Backes, and Christian Hammer. 2013. Idea: Callee-Site Rewriting of Sealed System Libraries. In *Engineering Secure Software and Systems - 5th International Symposium (ESSoS)*.

[16] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. 2015. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*.

A PATCH CASE STUDY SNIPPETS

```
public class DexClassLoaderPatch {
  static Map<String, SecureDexClassLoader> loaders = new
  ↪  HashMap<>();

  @PatchClass("dalvik.system.DexClassLoader")
  @PatchConstructor
  public static void init(DexClassLoader thiz, String dexPath,
  ↪  String dir, String path, ClassLoader parent) {
    loaders.put(thiz.toString(),
    ↪  RepackHandler.generateSecureDexClassLoader(dexPath, dir,
    ↪  path, parent));
  }

  @PatchClass("java.lang.ClassLoader")
  @PatchInstanceMethod
  public static Class<?> loadClass(ClassLoader thiz, String name)
  ↪  throws ClassNotFoundException {
    if (thiz instanceof DexClassLoader) return
    ↪  OriginalMethods.java_lang_ClassLoader.loadClass(thiz,
    ↪  name);
    SecureDexClassLoader loader = loaders.get(thiz.toString());
    if (loader == null) return
    ↪  OriginalMethods.java_lang_ClassLoader.loadClass(thiz,
    ↪  name);
    Class result =
    ↪  OriginalMethods.java_lang_ClassLoader.loadClass(loader,
    ↪  name);
    if (result == null) RepackHandler.raiseSecurityException();
    return result;
  }
}
```

Listing 4: Grab'n'Run Patch: Securing class loading by intercepting `DexClassLoader` methods

```
  <!-- Permission names must be unique -->
  <patch:replace sel="manifest/permission/@name">
    $${xpath(".")}.patched</patch:replace>

  <!-- Also adjust permission uses -->
  <patch:replace
  ↪  sel="manifest/uses-permission[not(starts-with(@name,
  ↪  'android.permission')) and not(starts-with(@name,
  ↪  'android.gms.permission')) and not(starts-with(@name, 'com.
  ↪  google.android.calendar'))]/@name">$${xpath(".")}.patched
  </patch:replace>

  <!-- Change class reference to use full original package name -->
  <patch:replace sel="manifest/application/@name">
    $${globalize(xpath("/manifest/@package"), xpath("."))}
  </patch:replace>

  <!-- Make sure the patched app can query the original -->
  <patch:add sel="manifest/queries">
    <package
    ↪  android:name="$${xpath(&amp;quot;/manifest/@package&amp;quot;)}"/>
  </patch:add>

  <!-- Provider authorities must be unique -->
  <patch:replace sel="manifest/application/provider/@authorities">
    $${appendeach(xpath("."), ".patched")}</patch:replace>

  <!-- Change package name by appending .patched -->
  <patch:replace sel="manifest/@package">
    $${xpath(".")}.patched</patch:replace>
```

Listing 5: App Cloning Patch: Simplified manifest patch for making a patched app installable alongside the original