IMMoRTALS Challenge Problem 3

3rd Party Dependency Upgrade

# Introduction

Modern software is built upon many layers of 3rd party libraries and operating system features. This has helped to drive down the development cost of software, allowing more capability to be developed for the same time and effort. In fact, the *raison d’etre* for operating systems is to provide common functionality such as keyboard drivers, networking stacks, and other I/O support so that each application doesn’t have to “reinvent the wheel”.

The cost of using 3rd party libraries and operating system features is that the development trajectory of those dependencies is outside the control of a given application developer. There are a variety of ways incompatible 3rd party code might be forced upon an application developer. From the mobile phone ecosystem, you can have situations where the OS vendor stops supporting the old version of the OS on newer hardware, for technical or economic reasons. Or it can be as simple as a third-party library that has security fixes in a new version of the library that the vendor chooses not to backport to the version of the library that our application was using.

# Challenge Problem Description

There are an incredible variety of ways that 3rd party libraries might evolve. We will focus on a few specific kinds of evolution for this challenge problem motivated by the following two scenarios

### Android Upgrades

Android applications like the client part of our tactical SA application is critically dependent on the Android Operating system. The Android Operating System is evolving rapidly, and services in the next version may differ from the current version syntactically (e.g., changes in function signature) as well as semantically (i.e., changes in the assumptions and intended effects). Consequently, a given application can be impacted by an upgrade in a number of ways ranging from failing to compile to runtime failures. One recent example that impacted the ATAK application was upgrading to Android OS version 6, when the permission model in the Android Operating System was changed. Prior to version 6, Android applications made compile-time static declarations of ‘permissions’ that the application required. These ‘permissions’ governed access to various subsystems of the operating system. Example permissions included use of the network, read and write from disk, use of Bluetooth, use of fine or course grained location, ability to place phone calls, etc. Android 6 kept the static declarations, but designated some subset of permissions as also requiring run-time confirmation to the user (i.e., to allow users to deny specific permissions). Strictly speaking the API that applications used didn’t change, so old applications would still compile, but they would fail when they ended up trying to use APIs that were affected by the run-time confirmable permission.

There are multiple aspects of this change that are challenging:

* Even though this particular change appears purely semantic i.e., the application will still compile—that is a false sense of success, the new version actually forces a specific way to use the permissions package, without which the application fails at runtime. This will force the DAS to include techniques to detect these situations and develop techniques to adapt the application code in response to changes in 3rd party library (our second motivating case will highlight a different approach).
* There is a scoping challenge- all the code that depends on the specific subset of permissions need to be changed. This will require the DAS’s program analysis to identify all such call sites.
* There is a usage mode/idiom challenge- the new version turned a synchronous API into an asynchronous one (because there was a potential to have to pop up a UI and wait for user input). This will require new ways to capture what has changed and use that information in the DAS’s reasoning.

We propose to start with this motivating example as a representative case of library evolution because of a couple of reasons:

* This case represents a non-trivial semantic change as compared to purely syntactic changes such as moving a function from one package to another, or rearranging the arguments of a function. The techniques we developed in Phase 1 can already address some of these.
* The underlying techniques to address this change can also be used to handle other non-trivial changes such as adding or dropping a new argument, adding or dropping an exception, deprecating a function that can be realized as a combination of other functions. These cases require additional metadata (either modeled or inferred) about the change (e.g., semantics/usage pattern of the added function, argument or exception) just like the semantics and usage pattern of the permission model.

### Security Upgrades in 3rd Party Libraries

There are many cases where a library has security fixes, but because the development team for the 3rd party library is small or underfunded, the fixes are only applied to the most recent version of the library and not to all prior vulnerable versions. It is also fairly common that applications stick to older versions of the library because the newer versions included API breaking changes. This combination leads to a situation where application maintainers cannot continue to use the old library because of the security vulnerability, and they cannot easily upgrade to the new library because of more substantial changes to the library API (i.e., there was a reason they stuck to the older version).

We propose to develop a usable solution to this practical challenge by investigating two potential approaches:

* If the security problem is isolated to a part of the library that isn’t in use by the application, then it might be possible to just cut the vulnerable code out of the existing (old) library, thereby reducing the attack surface of the library and the application. This will involve enhancements to our mutation testing and bytecode analysis.
* Another possible solution involves automated backporting of the security patch. This approach requires access to both the source code of the library and the series of individual patches that take the 3rd party library from its ‘old’ version (in use by our application) and the ‘new’ version. This will involve our enhancements to our program analysis and bytecode rewriting techniques.

In contrast to the 1st motivating case, this case involves adapting the library, specifically, the older in-use version of the library, as opposed to the application itself.

# Adaptation scenarios

The Test Adapter will support scenario generation in several ways. The DAS requires not just a notification of change, but oftentimes some metadata about the change as well. For instance, in the Android example above, the recipe for how to migrate code is provided by the vendor (e.g. <https://developer.android.com/training/permissions/requesting.html> ). We want to provide a way for a variety of scenarios to be developed, but at the same time allow appropriate metadata and/or concrete implementations to be emitted along with the high-level requirements.

A *Change Request* is what the DAS needs as input, and will include information about what library is affected, which functions within that library are affected, and what the mitigations are. Mitigations could include wrapping code with a recipe similar to the Android example, or “upgrade to library version X”). Our Test Adapter and TA4 interface will provide a way for the Test Harness to pick combinations of changes, and will then “fill out” the rest of the *Change Request* by adding appropriate metadata and potentially generating a new version of the library.

If the Test Harness specifies that the mitigation is a wrapper, then the Android description above gives us a concrete example: the test harness might flag any combination of Android permissions used by the application as requiring additional checking. (There are 24 “dangerous” permissions and 34 “normal” permissions that Android defines, Android 6 added additional requirements on the dangerous permissions.) The support tools (part of the Test Adapter) will generate an appropriate *Change Request* that specifies the affected permissions and the recipe for fixing the code. Finally, the Test Adapter will generate appropriate *Intent Tests* for the specified permissions.

IMMoRTALS will analyze the application to find out where the flagged permissions are used, and add wrapper code.

If the Test Harness specifies that mitigation is to upgrade to a newer version of the library, our Test Adapter will have to generate both the actual binary library with the specified changes, and the associated metadata that describes the changes. We will only consider changes that ‘break’ the application in some way, because library upgrades that are drop-in replacements don’t require any of the techniques we’re developing. The DAS will then try to use both the techniques described in the *Partial Library Upgrade* section to see if a new version of the application can be synthesized.

# CP3 Test Parameters

TBD

# Intent Specification and Evaluation Metrics

The intent specification comes in two categories. The first is the baseline functionality: a client sending position reports to the server, and the server sending other client’s position reports back. Testing that baseline intent is straightforward, and we tested this in Phase 1. The other intent is scenario specific change that was requested. Based on the mitigation a different set of tests might be required. If the mitigation is “upgrade to the new version”, then a battery of tests validating the baseline intent is sufficient (the API-breaking changes in the new library must have been resolved if the application compiles and runs with the new library). If the mitigation specified was code-wrapping, those scenarios will require intent tests to be generated on the fly. Specifically, we may modify the *recipe* specified in the Change Request mitigation to include some log messages that will allow us to verify that the code was modified.

# Test Procedure

The test harness will provide mission requirements by selecting change drivers as described in the “Intent specification” section. Tests will execute much as they did in our Phase-1 challenge problems: After the Test Harness provides the parameters, the TA and DAS will produce compliant versions of the client (ATAK) and server applications, then execute intent tests.

# Interface to the Test Harness (API)

[TODO]

