École Polytechnique Fédérale de Lausanne

The Story Of The Oxidizing Droid: An analysis of Rust integration and interfacing within the Android platform

by Niels Lachat

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Sánchez Marín Andrés
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 9, 2023

Abstract

This project aims at developing tools and methods to better understand how the Rust programming language has been integrated inside the Android operating system, and how it interfaces with existing code written in C and C++. This work will prove useful for further investigations of memory safety at the boundary between memory-safe and memory-unsafe code in the Android platform.

Because of the relatively recent introduction of Rust inside Android (2021), little work outside of Google itself has been done to understand the security implications of interfacing with unsafe code inside of Android, which is why we propose to clear the way for future analysis by our work.

At a high-level, we develop tools and methods to perform static and dynamic analysis of how Rust components interact with C and C++ at a high-level, and we propose ways to compute useful metrics about these interactions.

Contents

Chapter 1

Introduction

Since Android 12 (released in stable version on October 4, 2021 [1]), the Rust programming is supported to develop platform code on Android, alongside C and C++ [9]. This is particularly interesting, because Rust is a memory-safe language by default, contrary to C and C++. The goal with this change is to decrease memory vulnerabilities in Android, as is noted in the conclusion of a Google blog post discussing this change:

> " As Android migrates away from C/C++ to Java/Kotlin/Rust, we expect the number of memory safety vulnerabilities to continue to fall. Here's to a future where memory corruption bugs on Android are rare! "[9]

Indeed, as is reported in this same post, "Android 13 is the first Android release where a majority of new code added to the release is in a memory safe language.". Again based on this post, this increase in memory-safe languages usage has been correlated with a drop in memory safety vulnerabilities year after year.

This development seems very promising. However, as was noted in the post, memory-safe languages often need to interact with inherently unsafe components (IO, interfacing with existing unsafe code, inter-process communication, ...). This is one of the main ways in which memory-safe languages can introduce vulnerabilities.

In this project, we aim to understand how Rust interfaces with the existing Android platform code and in which proportions Rust is used in Android 13. For this, we develop tools and methods to perform both static and dynamic analysis of Android.

In essence, this project aims to perform a global, high-level analysis of the integration and interfacing of Rust in Android 13.

We will show using simple static analysis that Rust represents about 6.26% of lines of code amongst system oriented languages (Rust/C/C++), which indicates it is already well-integrated. More interestingly, we will show that on a representative execution of platform code (using the Android Compatibility Test Suite, CTS), interface code calls represent about 3.85% of total calls, with about 2.88% being calls from C or C++ to Rust, 0.97% from Rust to C or C++, 0.01% from Rust to C++[1].

This will allow subsequent work to understand how efforts should best be invested in order to find vulnerabilities in Android. The techniques developed can also be used on other projects starting to integrate Rust into their code bases.

[1]There are some caveats to these results, that we will explain in more detail in chapter 5.

Chapter 2

Background

In this chapter, we will explain all the necessary tools and concepts needed in order to understand how Rust code is integrated in the Android platform. We will start by explaining how Rust-C/C++ interop works outside of the Android platform. We will then explain the structure and build system of the Android platform, as well as explain how Rust is integrated into the build system. We will then illustrate with examples from the Android source code how all this works together to achieve the goal of Rust integration.

2.1 Rust-C/C++ interop

In order to understand how Rust is used inside Android, we must first understand how Rust code can call C/C++ and vice versa. The main mechanism that allows interoperation is the Foreign Function Interface (FFI), which enables function calls between languages through the C Application Binary Interface (ABI). When this common interface is respected, compiled code of the different languages can be linked together and the function calls between the languages are done in the same way as a call within C would be done. In this section, we will explain how this is happens in practice.

2.1.1 Calling C code from Rust

In order to call C functions from Rust, we need to instruct the Rust compiler to produce code that respects the C ABI when performing a call to a C function. This can be done quite simply by using the `extern` keyword. Let us look at an example from the Rust documentation[1]:

```
use libc::size_t;
```

[1]https://doc.rust-lang.org/nomicon/ffi.html#calling-foreign-functions

```
2 #[link(name = "snappy")]
3 extern {
4         fn snappy_max_compressed_length(source_length: size_t) -> size_t;
5 }
6 fn main() {
7         let x = unsafe { snappy_max_compressed_length(100) };
8         println!("max␣compressed␣length␣of␣a␣100␣byte␣buffer:␣{}", x);
9 }
```

As we can see on line 3, the `extern` keyword is used to indicate that the functions inside it are external function that need to be called with the C ABI. By default, `extern` means `extern "C"`, but other ABIs are also available.

Note also on line 7 that the call to the external function needs to be wrapped in an `unsafe` block.

This method works, but it would be very tedious to manually declare all external functions in Rust and to keep this up-to-date when new functions are added to the C library, or when functions get deprecated.

This issue is addressed by Rust bindgen, which allows to automatically generate Rust bindings for C and C++ libraries[2]. Let us see with an example how this works[3]. If we give the following C header file to bindgen:

```
typedef struct CoolStruct {
        int x;
        int y;
} CoolStruct;
void cool_function(int i, char c, CoolStruct* cs);
```

It will generate the following Rust FFI code:

```
/* automatically generated by rust-bindgen 0.99.9 */
#![allow(unused)]
fn main() {
        #[repr(C)]
        pub struct CoolStruct {
                pub x: ::std::os::raw::c_int,
                pub y: ::std::os::raw::c_int,
        }
        extern "C" {
```

[2]Note that currently only a subset of C++ features are supported by bindgen, see `https://rust-lang.github.io/rust-bindgen/cpp.html`

[3]Taken from `https://rust-lang.github.io/rust-bindgen/`

```
                    pub fn cool_function(i: ::std::os::raw::c_int,
                    c: ::std::os::raw::c_char,
                    cs: *mut CoolStruct);
        }
}
```

What is commonly done in order wrap many header files and produce bindings all at once is to make a single `wrapper.h` file where we `#include` all the C header files for which we need Rust bindings.

We then need to write a `build.rs` file that instructs cargo (the Rust build tool) where to find the wrapper header and where to write the generated bindings file[4]. Finally, to include the generated bindings, we use the `include!` macro: `include!(concat!(env!("OUT_DIR"), "/bindings.rs"));`. We mention this, because we will see that it is different in the Android build system.

2.1.2 Calling Rust code from C/C++

To call Rust functions from C, the following changes are needed in the source code [5]:

1. From the Rust side, we need to add two annotations to the function we want to ’export’: `#[no_mangle]` and `extern "C"`. The first annotation instructs the compiler to refrain from performing name mangling on the function name to allow linking. The second instructs the compiler to generate code for the function that respects the C ABI.
2. From the C side: declare the function signature by adding an `extern` annotation to the signature.

2.1.3 CXX: Support for C++ specific interop

Rust bingen can currently handle only a subset of C++ features[6]. This is problematic because a majority of the Android platform code is written in C++. Thankfully, the CXX library[7] enables interop between Rust and C++ for features missing from bindgen. This library knows some use in the Android platform, although the vast majority of interop is done through the C ABI method, as we

[4]For more details, see `https://rust-lang.github.io/rust-bindgen/tutorial-3.html`

[5]For more details, see `https://doc.rust-lang.org/nomicon/ffi.html#calling-rust-code-from-c`.

[6]“bindgen can handle some C++ features, but not all of them.”: `https://rust-lang.github.io/rust-bindgen/cpp.html`

[7]`https://cxx.rs/index.html`

will see. CXX augments the possible ABIs that can be declared after an `extern` keyword with `extern "Rust"` and `extern "C++"`[8] within a so-called bridge module declared with `#[cxx::bridge]`.

2.2 Android build systems

Now that we know how Rust interoperates with C and C++ outside of Android, we want to know how this works in Android. For this, we need some more background about the Android build system.

2.2.1 Past and Future: Make and Bazel

Before Android 7.0, Android was built with GNU Make[9], but this caused scaling issues. This is why they migrated to a custom build system called Soong that we will describe more precisely below. Soong is planned to be replaced with the Bazel build system in the coming years[10], but this change is not in effect yet, so we will focus our presentation on Soong.

2.2.2 Present: Soong

The Soong build system uses files called `Android.bp` to define build targets. A given `Android.bp` file typically defines multiple build targets, each with its own module type. We will be mostly interested in the module types that build Rust targets, so this is what we will look at next.

2.2.3 Rust module types

The following module types are available to build Rust targets in Soong: `rust_binary`, `rust_library`, `rust_ffi`, `rust_proc_macro`, `rust_test`, `rust_fuzz`, `rust_protobuf`, `rust_bindgen`. We will only explain in detail module types related to interop, namely `rust_bindgen` and `rust_ffi`.[11].

First, let us explain the `rust_bindgen` module type. `rust_bindgen` “Generates source and produces a Rust library containing Rust bindings to C libraries.”[11]. This corresponds to use case explained in subsection 2.1.1. The header files for which need to generate bindings are put in a wrapper header, and Rust bindgen is invoked to generate the bindings. The major difference is that

[8]see `https://cxx.rs/extern-rust.html` and `https://cxx.rs/extern-c++.html`

[9]see `https://source.android.com/docs/setup/build`

[10]`https://source.android.com/docs/setup/build/bazel/introduction`

[11]For details about other module types, see `https://source.android.com/docs/setup/build/rust/building-rust-modules/android-rust-modules`

there is no need to write a `build.rs` file to build the bindings, since this is handled directly by Soong (see [4], section “No nested build systems”).

Next, let us explain the `rust_ffi` module type. According to the documentation (see footnote 11), `rust_ffi` “Produces a Rust C library usable by cc modules, and provides both static and shared variants.”. This module type corresponds to the interfacing explained in subsection 2.1.2. Rust functions that we want to make available to C or C++ are marked as `extern "C"`.

We will show concrete examples of how these module types are used below.

Note that there is no `rust_cxx` module type that would be specific to the use of CXX. Because cxx is a normal Rust library, modules that need CXX simply add it as a dependency in the target config like so: `rustlibs: ["libcxx"]`[12]

Here is a table that recaps what we learned about Rust interop in Android:

Called language -> Caller language:	C/C++ (partial support for C++)	C++	Rust
C/C++			rust_ffi: Produces a Rust C library usable by cc modules
C++			extern "Rust": declares Rust types and signatures to be made available to C++ (CXX)
Rust	rust_bindgen: Generates source and produces a Rust library containing Rust bindings to C libraries.	extern "C++": declares C++ types and signatures to be made available to Rust (CXX)	

2.3 Examples

To illustrate what we explained in the previous section, we will now show some concrete examples of how interop works in Android.

[12]e.g. `https://cs.android.com/android/platform/superproject/+/android-13.0.0_r42:system/security/keystore2/src/vintf/Android.bp;l=28;drc=ec7872aaf6a2d47217014da555066403731bb1ee`

2.3.1 Calling Rust from C/C++: DoH module of DnsResolver

DNS-over-HTTPS (or HTTP/3) increases user privacy by encrypting DNS requests and making them look like regular HTTPS traffic. This protocol was implemented in Android as a module written in Rust, specifically because of Rust's memory safety:

> "The DNS resolver processes input that could potentially be controlled by an attacker, both from the network and from apps on the device. To reduce the risk of security vulnerabilities, we chose to use a memory safe language for the implementation." [5]

The DoH module is part of the wider `DnsResolver` module which handles the DNS requests on Android. At runtime this module lives as a daemon called `netd` (more on that later).

The DoH module is defined as a `rust_ffi_static` module type (a static library that exports an FFI for C/C++) in the `Android.bp` build file of the `DnsResolver` module and is built under the name `libdoh_ffi`[13]. Let us look at the function that allows to perform DoH queries : `doh_query`. This function is defined and implemented as expected in the `ffi.rs` file [14]:

```
#[no_mangle]
pub unsafe extern "C" fn doh_query(...) -> ssize_t {...}
```

It is also defined in a `doh.h` header file, so that it can be included from C/C++[15].

It can then be called from C++, for example in `PrivateDnsConfiguration::dohQuery` in the `DnsResolver` module (call on line 6)[16]:

```
1 #include "doh.h"
2 // ...
3 ssize_t PrivateDnsConfiguration::dohQuery(unsigned netId,
4         const Slice query, const Slice answer, uint64_t timeoutMs) {
5         // ...
6         return doh_query(mDohDispatcher, netId, query.base(), query.size(),
7         answer.base(), answer.size(), timeoutMs);
8 }
```

We will come back to this example in the following chapters.

[13] `https://cs.android.com/android/platform/superproject/+/android-13.0.0_r42:packages/modules/DnsResolver/Android.bp;l=341;drc=297f4afe58effdf61cf6b4840f9664b10877be1d`

[14] `https://cs.android.com/android/platform/superproject/+/android-13.0.0_r42:packages/modules/DnsResolver/doh/ffi.rs;l=255;drc=882eeb91d914fb540cc9dbff745ebea37b820b45`

[15] `https://cs.android.com/android/platform/superproject/+/android-13.0.0_r42:packages/modules/DnsResolver/doh.h;l=104;drc=882eeb91d914fb540cc9dbff745ebea37b820b45`

[16] `https://cs.android.com/android/platform/superproject/+/android-13.0.0_r42:packages/modules/DnsResolver/PrivateDnsConfiguration.cpp;l=543;drc=882eeb91d914fb540cc9dbff745ebea37b820b45`

2.3.2 Calling C from Rust: `Virtualization/microdroid_manager`

Microdroid is a light-weight VM, which can run on Android thanks to the Android Virtualization Framework (AVF). "Android Virtualization Framework (AVF) provides secure and private execution environments for executing code."[17] The `microdroid_manager` module was written in Rust, but requires access to a C library, `libcap.h`. So they wrote a safe wrapper around the bindings generated by bindgen. For example, `drop_inheritable_caps` is a safe wrapper around the functionnality provided by `libcap.h`.

The module configuration in `Android.bp` for `cap_bindgen` that defines how the build system should call bindgen to generate bindings looks as follows:

```
rust_bindgen {
        name: "libcap_bindgen",
        edition: "2021",
        wrapper_src: "bindgen/libcap.h",
        crate_name: "cap_bindgen",
        source_stem: "bindings",
        shared_libs: ["libcap"],
        bindgen_flags: [
        "--default-enum-style␣rust",
        ],
        visibility: [
        "//packages/modules/Virtualization:__subpackages__",
        ],
}
```

18

Note in particular the `wrapper_src` and `crate_name` attributes. The first attribute indicates for which headers to generate bindings, and the second indicates which crate name will be given to the bindings (needed to import the crate).

We now see how the generated bindings can be used from Rust:

```
use cap_bindgen::{
        cap_clear_flag, cap_drop_bound, cap_flag_t, cap_free, cap_get_proc,
        cap_set_proc, cap_value_t,      CAP_LAST_CAP,
};
//...
```

[17]https://source.android.com/docs/core/virtualization
[18]https://cs.android.com/android/platform/superproject/+/master:packages/modules/Virtualization/libs/capabilities/Android.bp;l=5;drc=3452ee2bb0a2a941153f017dd45addaf03cea1ee

```
6  pub fn drop_inheritable_caps() -> Result<()> {
7          unsafe {
8                  // SAFETY: we do not manipulate memory handled by libcap.
9                  let caps = cap_get_proc();
10                 scopeguard::defer! {
11                         cap_free(caps as *mut std::os::raw::c_void);
12                 }
13                 if cap_clear_flag(caps, cap_flag_t::CAP_INHERITABLE) < 0 {
14                         let e = Errno::last();
15                         bail!("cap_clear_flag␣failed:␣{:?}", e)
16                 }
17                 if cap_set_proc(caps) < 0 {
18                         let e = Errno::last();
19                         bail!("cap_set_proc␣failed:␣{:?}", e)
20                 }
21         }
22         Ok(())
23 }
```

[19] On lines 1-4 we see the generated bindings being imported from `cap_bindgen` (which corresponds to `crate_name` attribute in the target definition). On lines 7-21, we see an `unsafe` block from which the functions get called.

2.3.3 Rust-C++ interop with CXX: Bluetooth module

One example of the use of interop between Rust and C++ can be found in the `ffi.rs` file in the part of the Bluetooth module written in Rust [20].

2.4 Android testing infrastructure & tooling

To assist us in our dynamic analysis, we will leverage existing tools of the Android ecosystem. We present them briefly here, but we will examine their use in detail in later chapters.

[19] `https://cs.android.com/android/platform/superproject/+/master:packages/modules/Virtualization/libs/capabilities/src/caps.rs;l=28;drc=3452ee2bb0a2a941153f017dd45addaf03cea1ee`

[20] `https://cs.android.com/android/platform/superproject/+/master:packages/modules/Bluetooth/system/rust/src/core/ffi.rs;l=34;drc=6ca00da40441915e01b044826f5bac19ee4401d7`

2.4.1 Compatibility Test Suite (CTS)

> "[...]. The CTS is a set of unit tests designed to be integrated into the daily workflow (such as via a continuous build system) of the engineers building a device. Its intent is to reveal incompatibilities early on, and ensure that the software remains compatible throughout the development process."[2]

The CTS consists in approximately 200'000 individual tests (201'913 on the `android-13.0.0_r42` branch we will be using). Currently, these tests are either unit tests (testing single classes or functions) or functional tests (which use a combination of APIs to test more high-level behavior). We are interested in using the CTS because it will allow us to perform large-scale testing of every aspect of the Android platform code, without needing to write tests ourselves (which would probably be impossible).

2.4.2 Cuttlefish Virtual Device (CVD)

> "Cuttlefish is a configurable virtual Android device that can run both remotely (using third-party cloud offerings such as Google Cloud Engine) and locally (on Linux x86 machines)."[3]

The main goal of cuttlefish is to simplify testing of Android platform code, for example for continuous integration. Our use case for cuttlefish is to enable us to run the CTS for our large-scale dynamic analysis (explained in later chapters).

2.4.3 Perfetto

> "Perfetto is a production-grade open-source stack for performance instrumentation and trace analysis. It offers services and libraries for recording system-level and app-level traces, native + java heap profiling, a library for analyzing traces using SQL and a web-based UI to visualize and explore multi-GB traces."[6]

Our main use for perfetto will be to perform callstack sampling of native processes on Android, while we execute tests that trigger functions that we are interested in. We will explain in detail how we use perfetto in the implementation chapter.

Chapter 3

Design

Towards the aim of better understanding how Rust has been integrated in -and interfaces within- the Android platform, we pursued two main paths: static and dynamic analysis. The former in order to understand at a high-level how much of Android is currently written in Rust, and the latter to understand how much of this code actually gets used at runtime, which is the more useful metric of the two (only code that gets executed can have an impact on security). In the dynamic analysis, we will be most interested in how interface functions are called, and in which proportions.

3.1 Static analysis

3.1.1 Module types

First, we wanted to understand which Rust module types are most used in order to understand at a high level in which direction Rust $\Longleftrightarrow$ C/C++ interactions are most likely to occur. For example, if there are 10 times more `rust_ffi` modules being declared than `rust_bindgen`, it would suggest that it is more frequent to have C/C++ calling Rust than the other way around. Of course this would need to be confirmed by dynamic analysis.

The design of this experiment is relatively simple. The high-level idea is to go over the whole Android source code and to for each `Android.bp` build file, count how many times each module type appears (for module types starting with the `rust_` prefix).

To get a better idea of how module types are distributed in the source code, we also wanted to count module types per top-level directory of the Android source tree. This allows us to get a better view of where most Rust modules live, and which type of modules are most used in which domain of Android. For simplicity, we keep the coarse-grained approach of only counting module types per top-level directory, but it would be possible to extend the approach to go down to the package level. However, this would loose vision of the higher-level distribution, which is why we didn't choose to implement this fine-grained approach.

3.1.2 Lines of code

Next, we wanted to have an idea of how much code Rust represents among system oriented languages in Android. For this we use the `cloc`[1] utility to go over the source tree and count lines per language. This utility takes into account lines of comment and blank lines, which allows us only compare lines of actual code.

3.2 Dynamic Analysis

Static analysis can only give a rough idea of how Rust interacts with other languages in Android. In particular, we wanted to understand how components interact by seeing which components (packages, modules) use which components, what type of interactions are most frequent (Rust→C/C++ or C/C++→Rust); in other words, understand how Rust interacts with the rest of the system at runtime.

To attain this goal, we decided to use callstack sampling, as it would allow us capture examples of how FFIs are used at runtime, and also to build flamegraphs to understand which proportion of calls use the FFIs.

We chose to use callstack sampling instead of other profiling techniques, because this allows us to get an understanding of which functions are called (in particular, which interface functions in which we are interested) and in which proportion (i.e. by building a flamegraph for example). The major limitation of this technique is that it does not allow us to capture the absolute duration of function calls, nor the memory used by code in each function call. This is acceptable in our use case, because we are more interested in proportions rather than in absolute values of execution duration for function calls. But it would be interesting for future work to try different techniques to get an even better understanding of the impact of Rust code on Android.

We use the Perfetto tool to enable us to do the callstack sampling, because it is very flexible (with many configuration options, in particular to sample individual processes) and has been developped with Android in mind, which means it is easy to setup.

We first wanted to setup a small-scale experiment focused on a single component written in Rust to understand the challenges of doing callstack sampling on Android. After having done the small-scale experiment, we hoped we would be able to scale-up and perform runtime analysis on

[1] `https://github.com/AlDanial/cloc`

the whole codebase, by leveraging the Compatibility Test Suite.

3.2.1 Small-scale analysis: DoH

We needed a component that would be relatively easy to trigger with API calls and which didn't require interacting with the external world (e.g. Bluetooth, Ultra-Wide Band, ...). Thus we ended up choosing the DNS-over-HTTPS (presented in subsection 2.3.1) component which was apparently already well-integrated and quite easy to trigger.

The idea is to implement an Android app that makes many DNS requests, with the device configured to use DoH for DNS queries. While the app does this, we use Perfetto to collect callstack samples for the `netd` process. Then, we can look for callstacks that contain the `doh_query` function, and understand the context of the call (functions before and after in the callstack), and get an idea of how much of the execution this represents by generating a flamegraph thanks to Perfetto.

3.2.2 Large-scale analysis: CTS on CVD

Having gained confidence in the general approach with our small-scale test, we decided that scaling up the approach would allow more interesting and more global analysis of all the components written in Rust, rather than just testing component by component. This method would also allow the approach to remain useful with the addition of new Rust components, and also maybe be applicable to other systems that integrate Rust components.

What we want to know with this analysis is: which direction of interfacing is most frequent during execution of the CTS, Rust→C/C++ or C/C++→ Rust?

The general approach for our analysis consists of 3 phases:

1. **Pre-processing**: In this phase, we collect all function names that we can identify as being interface functions. We define 4 types of interface functions: C→Rust, C++→Rust, Rust→C and Rust→C++. To distinguish between these types, we use clues left by the build system (bindgen headers) and function annotations. More details on this in the implementation chapter.
2. **Execution**: In this phase, we execute the CTS on a CVD. In parallel to this, we perform callstack sampling *on all processes* running on the CVD, at a high-enough frequency (the higher the better, but above 100Hz, this starts getting very heavy on the CVD side). The fact that we perform the sampling for all processes may seem inefficient, but it would be quite hard to know, for each test, which process to sample, because a lot of the platform code we are interested in runs inside daemons across multiple separate processes. In the small-scale

analysis, we could find out that `netd` was the process hosting the DoH module by examining the source code, but this is of course not feasible (and not scalable) for the large-scale analysis.

3. **Post-processing**: We now have the two pieces of data we are interested in: which functions are interface functions (and of which type), and which functions appear in callstack samples collected during execution of the CTS. Given this, we can perform the inner join, and obtain the list of interface functions called during the CTS run, with associated type. In the end, we can produce statistics about which type of interaction is most frequent.

The process as we have described has one severe limitation: function names are not globally unique. This means that it is very likely that we collect a function name in the pre-processing phase and classify it as a Rust→C interface function for example, but then at runtime, a different function within a different namespace but with the same name is called. This will cause us to falsely count a Rust→C interaction, when in fact this was not an interface call. It would probably require much more involved analysis and instrumentation to address this limitation, which is why we set aside this consideration for what remains. We should however keep this in mind when examining the results produced by our analysis.

Chapter 4

Implementation

4.1 Static analysis

The static analysis was relatively simple from an implementation point of view. We will however briefly describe some aspects of the implementation of the analysis of rust module types. For the counting of lines of code, there is nothing to describe as we rely entirely on `cloc`.

4.1.1 Module types

The analysis is done in two steps (although this is could be simplified): First, we use `grep` to identify all build files containing definitions for Rust modules. The command used is as follows:

```
grep -r -E '^rust_.+␣{' --after-context=1 --include=Android.bp AOSP-source
```

[1] This produces a list of build files looking as follows (short extract):

```
[...]
/packages/modules/DnsResolver/Android.bp:rust_ffi_static {
/packages/modules/DnsResolver/Android.bp-    name: "libdoh_ffi",
--
/packages/modules/DnsResolver/Android.bp:rust_test {
/packages/modules/DnsResolver/Android.bp-    name: "doh_unit_test",
[...]
```

[1]The `-after-context=1` flag is not necessary for this analysis, but it allows us to see the name of the module corresponding to the module type, which was useful many times in other contexts.

We then use a Python script that parses this output and produces, for each top-level directory of the source-tree, a count of the module types.

4.2 Dynamic Analysis

4.2.1 Small-scale analysis: DoH

Setup and Android config

The app that performs the triggering of the DoH module is written in Kotlin. The UI is extremely simple (see figure 4.1), with only buttons that allow to trigger the desired component (originally, we explored the possibility of making multiple small-scale tests, but we then decided to change direction and explore large-scale testing instead; this is why there are buttons to trigger `Keystore2` and `Bluetooth`). The emulator used for testing is one that can be installed from Android Studio's

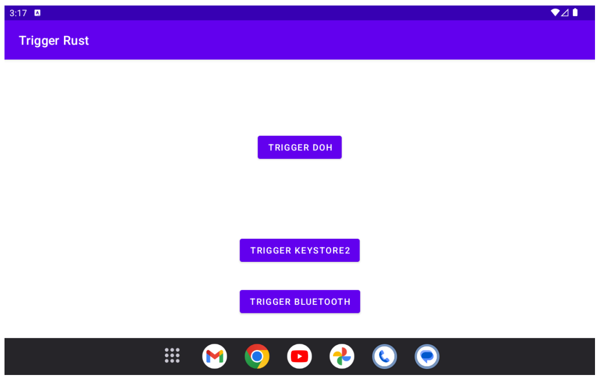
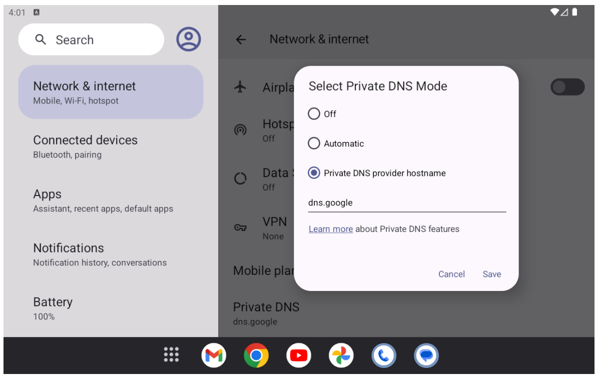


Figure 4.1: UI of the app to trigger the DoH module

virtual device manager (we used the latest available image, namely `Android API UpsideDownCake x86_64`).

Before performing the DNS requests, we must make sure that Android is configured to use DoH. This can be done from the system settings, as seen in figure 4.2. One quirk we discovered while trying to make this work is that currently, the list of available DoH servers is *hard-coded* in a header file[2]. There are 3 supported servers for production use: `dns.google`, `dns64.dns.google` and `cloudflare-dns.com` and 2 for testing (`example.com` and `dns.androidtesting.org`). This is probably temporary, but it is still useful to note. We chose to use `dns.google` for our testing.

[2]see: `https://cs.android.com/android/platform/superproject/+/master:packages/modules/DnsResolver/PrivateDnsConfiguration.h;l=250;drc=4c80818efc71774b662aa3167d03cf49b1266909`



Select Private DNS Mode

- Off
- Automatic
- [✓ Private DNS provider hostname]

dns.google

Figure 4.2: Adjusting the Private DNS settings

Triggering DoH through the API

The ultimate goal is to trigger at least the `doh_query` from the DoH module. We guessed that this could be achieved by calling a high-level Android API which would ultimately result in this function being triggered. The API we call is a method of the `DnsResolver` class called `query` (unsurprisingly)[3]. The queries are started in rapid succession so that they run in parallel and so that we maximize our chances of catching a callstack that contains the call to `doh_query` we are interested in.

Perfetto config

We use Perfetto's `cpu_profile` script to perform the callstack sampling[4]. We configure the sampling frequency to be 500Hz, and the `ring_buffer_read_period_ms` to be 1, so that we don't risk losing samples[5]. For the processes to sample, we choose the app process and the `netd` process (`scope` setting in `callstack_sampling`).

A complete execution

Given all this, we now briefly describe how the whole setup is used to perform the callstack sampling:

1. Start Perfetto's `cpu_profile` script, with the right config.
2. With the app opened on the emulator, click on the "TRIGGER DOH" button to start sending DNS requests.
3. When the app is finished with the requests, stop the profiler.

[3]see: `https://developer.android.com/reference/kotlin/android/net/DnsResolver#query`
[4]see: `https://perfetto.dev/docs/quickstart/callstack-sampling`
[5]see: `https://perfetto.dev/docs/reference/trace-config-proto#PerfEventConfig`

4.2.2 Large-scale analysis: CTS on CVD

The large-scale analysis was more interesting from an implementation point of view. We will explain some of the details for each phase of the analysis.

Pre-processing

The pre-processing is done with a Python script. In the pre-processing, the most interesting part is how we distinguish the types of interface functions. For each Rust source file we examine, we look at the first line of the file. If it starts with `"/* automatically generated by rust-bindgen"`, we know that we are in a file containing bindings for use by Rust generated by bindgen. Thus we know that functions that are marked as `extern "C"` are functions that allow Rust→C/C++ interaction. The limitation here is that we can't distinguish between C and C++, thus we arbitrarily choose to consider it as a Rust→C function (we will need to keep this in mind when interpreting the results). If we are *not* in a bindgen generated file, we use the `extern` annotation to find the direction of interaction:

- `extern "C"` indicates a C→Rust function (the Rust function is *exported* with the C ABI). It could of course be the case that this function is used by C++, but like before, we have no way of knowing this, so we make this arbitrary choice.
- `extern "C++"` indicates a Rust→C++ function (CXX declaration). We chose to consider only functions declared directly in the Rust file, and not try to follow `include!` macro inclusions of headers[6], because it seemed quite time-consuming to implement correctly. This means that only a small fraction of Rust→C++ interface functions are actually detected as such, which is again a limitation we need to keep in mind when interpreting the results.
- `extern "Rust"` indicates a C++→Rust function (again a CXX-specific declaration[7]). This detection is unambiguous and complete, contrary to the others.

Of course, we need to perform some parsing to extract function names, and we also need to take into account the separate cases of blocks and single function declarations with the `extern` annotation. At the end of the process, we save the collected function names grouped by type into a JSON file.

[6]see: `https://cxx.rs/extern-c++.html`
[7]see: `https://cxx.rs/extern-rust.html`

Execution

The execution phase requires some preliminary setup and configuration to be able to run. We need to build the CVD and the CTS[8]. We decided to build from the `android-13.0.0_r42` branch because this was the most up-to-date branch for Android 13 (also, the CTS is not yet ready for Android 14).

We decided to write a bash script to automate the setup, running and cleanup of the tests. We briefly describe this:

1. **Setup**: First, the environment is setup[9]. Then the CVD is started: `launch_cvd --num_instances=1 --daemon --memory_mb=24000 --cpus=8`. We allocate more than enough RAM (24'000 RAM, probably also works with 8000 MB) and configure it to have 8 CPUs. Starting the CVD with too little RAM (e.g. 2000 MB) causes frequent crashes when running the CTS while profiling at high frequency. When the CVD has booted, we enable the Wi-Fi (needed by some tests).

2. **Running the CTS**: First the CTS is started[10] in the background. Then, we start the `cpu_profile` to start collecting callstack samples. We use a sampling `frequency` of 100 Hz, and a `ring_buffer_read_period_ms` of 200. By trying with multiple configurations, this seems to be the configuration that allows to collect the most samples while not crashing the CVD too often. Increasing the buffer read period seems to be the main way we can make the CVD more stable. One detail to think about is that the profiling stops each time the CVD crashes and reboots, so we have to keep restarting the profiling until the whole CTS has finished executing.

3. **Cleanup**: The CVD is shutdown using the `stop_cvd` command.

Post-processing

As explained before, we need to combine the functions we collected during the preprocessing with the callstacks we acquired during execution to extract the metric we want, which is the relative frequency of the types of interactions. Let us show some samples of the data we are working with, to illustrate the whole process more clearly. First we load the list of collected function names, with their associated interface type:

```
name                             interface_type
crosvm_client_stop_vm               C_CALLS_RUST
crosvm_client_suspend_vm            C_CALLS_RUST
crosvm_client_resume_vm             C_CALLS_RUST
```

[8]The process is described here: `https://source.android.com/docs/setup/create/cuttlefish-cts`

[9]`source build/envsetup.sh; lunch aosp_cf_x86_64_phone-userdebug`

[10]As detailed here: `https://source.android.com/docs/setup/create/cuttlefish-cts#run-cts`

```
crosvm_client_balloon_vms       C_CALLS_RUST
crosvm_client_usb_list          C_CALLS_RUST
...                                      ...
tombstoned_notify_completion  RUST_CALLS_CPP
get_hal_names                 RUST_CALLS_CPP
get_hal_names_and_versions    RUST_CALLS_CPP
get_hidl_instances            RUST_CALLS_CPP
get_aidl_instances            RUST_CALLS_CPP

[9817 rows x 1 columns]
```

As a sanity check, we can check that for example `crosvm_client_stop_vm` is indeed a function marked as `extern "C"`, which is confirmed by a search through the source code [11]. Similarly for `tombstoned_notify_completion` which is indeed in an `extern "C++"` block[12].

Next, we load the list of all the functions collected from the callstack sampled during the execution of the CTS:

```
                                                    name
0                                             __libc_init
1                                                    main
2                                           _Z9adbd_maini
3                       _ZN21fdevent_context_epoll4LoopEv
4                   _ZN15fdevent_context13FlushRunQueueEv
...                                                   ...
141217  _ZN7android4base12StderrLoggerENS0_5LogIdENS0_...
141218  _ZNSt3__112basic_stringIcNS_11char_traitsIcEEN...
141219  _ZN5scudo9AllocatorINS_13AndroidConfigEXadL_Z2...
141220  _ZN5scudo28SizeClassAllocatorLocalCacheINS_20S...
141221  _Z15StartSubprocessNSt3__112basic_stringIcNS_1...

[141222 rows x 1 columns]
```

This is enabled by the convenient Python API offered by Perfetto[13], in particular we use the Batch Trace Processor[14] because we may have multiple traces (because of the CVD rebooting and the profiling restarting).

[11]See: https://cs.android.com/android/platform/superproject/+/master:external/crosvm/crosvm_control/src/lib.rs;l=60;drc=0e1c85f2932fa6c4581af60995b2cf392de7e834

[12]See: https://cs.android.com/android/platform/superproject/+/master:system/core/debuggerd/rust/tombstoned_client/src/lib.rs;l=124;drc=3b7591248d6e643d95196ca9b8a90b47413504e8

[13]https://perfetto.dev/docs/quickstart/trace-analysis#python-api

[14]https://perfetto.dev/docs/design-docs/batch-trace-processor

We use the following code to query the function names:

```python
# ...
with BatchTraceProcessor(traces) as btp:
        called_fns = btp.query_and_flatten(
                'select name from stack_profile_frame')
        # ...
```

Where `stack_profile_frame` is the table containing the names of functions in the collected call-stacks[15].

We then perform the inner join of both tables on the name column, and obtain the following:

```
                    name interface_type
1                   main   C_CALLS_RUST
185                 main   C_CALLS_RUST
230                 main   C_CALLS_RUST
240                 main   C_CALLS_RUST
934                 main   C_CALLS_RUST
...                  ...            ...
134484              dup3   C_CALLS_RUST
136489  EVP_DigestVerify   RUST_CALLS_C
136491      ECDSA_verify   RUST_CALLS_C
136492   ECDSA_do_verify   RUST_CALLS_C
139561           strncat   C_CALLS_RUST

[5443 rows x 2 columns]
```

We can see here that there are some functions that really should not be considered interface functions (e.g. main) and even functions that have the wrong interface type (e.g. strncat, which is a function from the C standard library, so it should be marked as `RUST_CALLS_C`). Another limitation is that we actually don't know if the functions with type `RUST_CALLS_C` are really the result of calls from Rust (and not from C). Because of lack of time, we cannot investigate further into these problems in the current report, but there may be some ways to improve the accuracy of the reporting of interface type calls. For example, it may be interesting to try to use the `mapping` column of the `stack_profile_frame` table to filter out calls from libraries not written in Rust.

Even with these problems, we believe that the general steps of the process are correct, and it would be possible to obtain accurate results with some more work on the problems mentioned.

[15] See docs for more details: `https://perfetto.dev/docs/analysis/sql-tables#stack_profile_frame`

Chapter 5

Evaluation

We will now present the results of our various analyses.

5.1 Static Analysis

5.1.1 Module types

We first report the total counts of Rust module types for the whole project:

Module type	Nb occurences
rust_test	424
rust_library	319
rust_defaults	94
rust_library_host	40
rust_proc_macro	37
rust_test_host	35
rust_binary	33
rust_bindgen	29
rust_library_rlib	25
rust_fuzz	18
rust_ffi_static	16
rust_binary_host	15
rust_toolchain_library_rlib	15
rust_protobuf	7
rust_ffi_shared	3
rust_toolchain_library	3
rust_ffi	2
rust_ffi_host_static	2
rust_bindgen_host	1

Module type	Nb occurences
rust_ffi_host	1
rust_library_host_rlib	1
rust_stdlib_prebuilt_host	1

What we first observe here is that a clear majority of Rust modules (other than `rust_test` which only produces test artifacts) consists in library modules (319 `rust_library` modules). We will see later that this mostly (257/319 ≈ 80%) corresponds to external dependencies in the `/external` top-level directory. Moreover, we observe that there are 29 bindgen (`rust_bindgen`), and 24 FFI (`rust_ffi_*`) modules. This result gives very little insight because there is no clear difference between the two types of modules. It may indicate that we could expect more Rust→C/C++ interactions, but we must remain cautious not to over-interpret these results.

In appendix A, we report the count of module types per top-level directory. What we first notice is that the external dependencies contribute for the most part to the count of module types, with 759 out of 1121 total rust modules (≈ 68%).

The next biggest contributor is the `/packages` directory. This represents 152/1121 ≈ 14% of Rust modules. The packages directory contains stand-alone modules such as the `DnsResolver` (which contains the Rust DoH implementation we examined) or the Bluetooth stack (with the new Gabeldorsche stack partly written in Rust). Here we see that there are twice as many FFI than bindgen modules (14 vs 7), so we could expect more instances of C/C++→Rust interactions than the opposite.

The last top-level directory we will discuss is `/system`. This represents 133/1121 ≈ 12% of rust modules. This directory is interesting, because there are no FFI modules, only 13 bindgen modules. Thus we can suppose that there are no C/C++→Rust interactions in this directory, but there is some need for Rust→C/C++ interactions for C/C++ code in this directory.

5.1.2 Lines of code

We now present the results obtained from the `cloc` tool (only for the systems languages).

Language	**Nb. Files**	**Code only [lines]**	**Code only [%(TOTAL)]**
C++	71099	26′202′230	64.32
C	38120	11′980′062	29.41
Rust	8189	2′551′895	**6.26**
TOTAL	117408	40′734′187	100

We see that among systems languages, Rust currently consists of 6.26% of lines of code. This is already quite impressive, considering its recent addition to the usable systems languages in Android.

5.2 Dynamic Analysis

5.2.1 Small-scale analysis

First, we show a flamegraph containing the call to `doh_query` in the `netd` process, produced from sampling during 50 DoH queries:

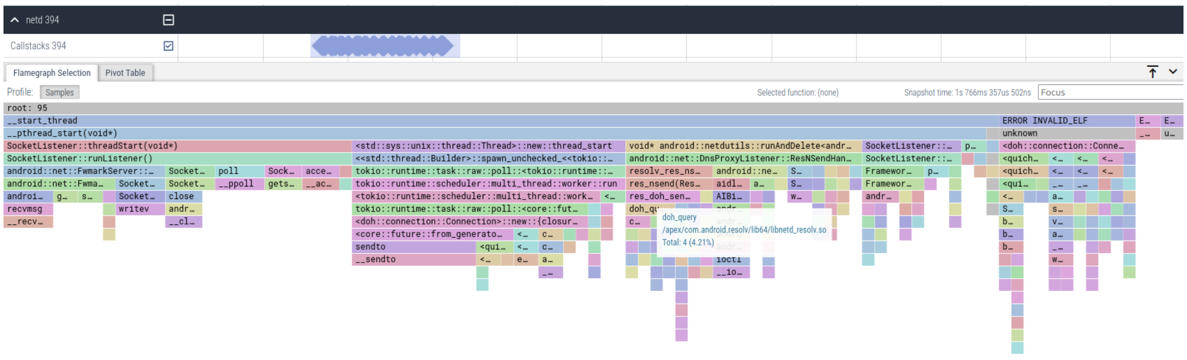
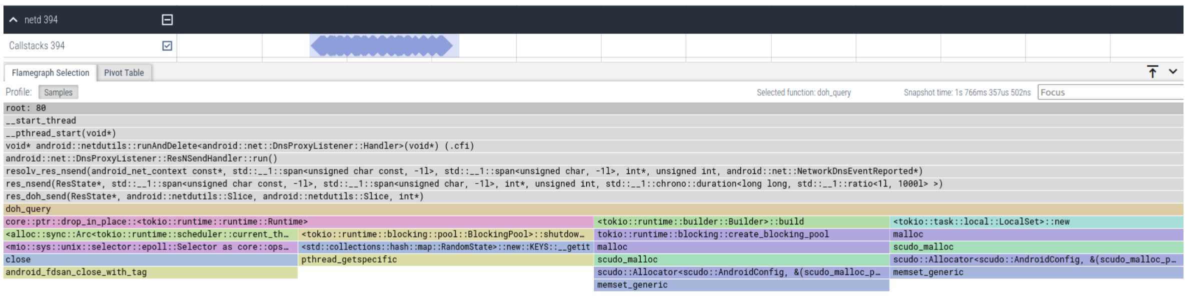

netd 394
Callstacks 394
Flamegraph Selection
Pivot Table
Profile: Samples
Selected function: (none)
Snapshot time: 1s 766ms 357us 502ns
Focus
root: 95
__start_thread
__pthread_start(void*)
SocketListener::threadStart(void*)
<std::sys::unix::thread::Thread>::new::thread_start
void* android::netdutils::runAndDelete<andr…
ERROR INVALID_ELF
unknown
doh_query
/apex/com.android.resolv/lib64/libnetd_resolv.so
Total: 4 (4.21%)

Figure 5.1: Flamegraph showing the call to `doh_query` within the `netd` process

We also show the callstack to and from `doh_query` to gain a better understanding of the context of the call:



```
root: 80
__start_thread
__pthread_start(void*)
void* android::netdutils::runAndDelete<android::net::DnsProxyListener::Handler>(void*) (.cfi)
android::net::DnsProxyListener::ResNSendHandler::run()
resolv_res_nsend(android_net_context const*, std::__1::span<unsigned char const, -1l>, std::__1::span<unsigned char, -1l>, int*, unsigned int, android::net::NetworkDnsEventReported*)
res_nsend(ResState*, std::__1::span<unsigned char const, -1l>, std::__1::span<unsigned char, -1l>, int*, unsigned int, std::__1::chrono::duration<long long, std::__1::ratio<1l, 1000l> >)
res_doh_send(ResState*, android::netdutils::Slice, android::netdutils::Slice, int*)
doh_query
core::ptr::drop_in_place::<tokio::runtime::runtime::Runtime>                                         <tokio::runtime::builder::Builder>::build        <tokio::task::local::LocalSet>::new
<alloc::sync::Arc<tokio::runtime::scheduler::current_th…  <tokio::runtime::blocking::pool::BlockingPool>::shutdow…  tokio::runtime::blocking::create_blocking_pool  malloc
<mio::sys::unix::selector::epoll::Selector as core::ops…  <std::collections::hash::map::RandomState>::new::KEYS::__getit  malloc  scudo_malloc
close                                                     pthread_getspecific  scudo_malloc  scudo::Allocator<scudo::AndroidConfig, &(scudo_malloc_p…
android_fdsan_close_with_tag                              scudo::Allocator<scudo::AndroidConfig, &(scudo_malloc_p…  memset_generic
                                                          memset_generic
```

Figure 5.2: Flamegraph showing the call to `doh_query` in context

What is interesting with these results, is that it allows us to understand very well the context in which the Rust function is called (from a C++ function called `res_doh_send`[1]), and also, which functions get called after it in the callstack (e.g. `malloc`). Having this information can be quite

[1] https://cs.android.com/android/platform/superproject/+/master:packages/modules/DnsResolver/res_send.cpp;l=1379;drc=ac98b9d4efaac218fb64a9d1581ef2f875be55da

valuable when trying to understand where security problems might appear when interfacing with Rust. The method we used can very well be reused in other scenarios.

5.2.2 Large-scale analysis

We now show the results of our large-scale analysis.

First, we show a table summarizing the counts of function calls for interface functions:

Interface type	Call count	Call percentage
C_CALLS_RUST	4067	2.879
RUST_CALLS_C	1368	0.968
RUST_CALLS_CPP	8	0.005
(Non-interface calls)	135779	96.145
TOTAL	141222	100.0

Now we show the data only for interface calls as a bar plot:

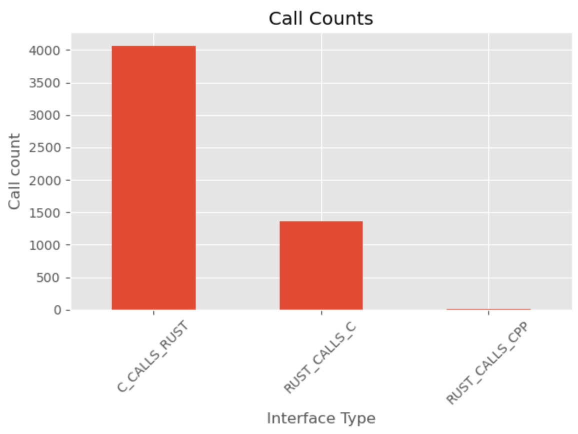


Figure 5.3: Bar plot showing the call counts per interface type

As explained before, we must observe these results with abundant caution, because of the many limitations we examined. Nonetheless, we could interpret these results as showing that calls from C/C++→Rust are more frequent than the opposite, which makes sense because of the wider context (Rust being integrated into the existing Android platform code gets called more often by this code than the opposite). Also, we can note that there is no C++→Rust interactions happening, which individually is meaningful, because this was the only type of interface type that we could detect in a complete and unambiguous way (see section 4.2.2, Pre-processing).

Overall, we can see that the results of the large-scale dynamic analysis are a bit disappointing because they cannot be confidently interpreted because of the limitations we observed. However, we believe that, more than the results, the method presented in the Design chapter and detailed in the Implementation chapter will be interesting for future work focusing on understanding Rust interfacing in the Android platform. Also, we believe that the limitations can be addressed with more work, which sadly we couldn't undertake because of time limitations.

Chapter 6

Related Work

As Rust integration into Android is relatively recent (since Android 12, released in October 2021), there is little academic literature that we could find looking specifically into the integration mechanisms or the problems that could arise because of it.

In somewhat related work, we can cite NativeGuard [7] which aimed to make native libraries more secure by isolating them into a non-privileged application. This was more targeted at 3rd party native libraries than at examining Android's native platform code, but it could probably also be applicable in this (and also for the case of Rust libraries). The limitation being that isolating native platform libraries would probably incur too much overhead. Our work is more focused on the specific case of Rust interfacing inside Android platform code.

Outside the academic literature, we can also cite the blog posts from `https://security.googleblog.com` made by Google engineers that we already referenced in previous chapters : [4, 5, 8, 9]. These posts are very interesting because they give an inside explanation of why some decisions were made, and even provide some analysis closely related to our work. In particular, [9] is very interesting because it correlates the decrease of new memory unsafe code (increase in Rust code) and the decrease in memory safety vulnerabilities. This was actually our starting point for this project: we wanted to look at the interfaces between memory safe Rust, and memory unsafe C/C++, because as noted in this same blog post, it is there that Rust code must use `unsafe` blocks, and only there that memory safety issues can still occur.

Chapter 7

Conclusion

As we saw, we developed useful static and dynamic methods and tools for understanding how Rust interfaces with C and C++ in the Android platform. We found some indication that matches our intuition that calls from the existing C/C++ code to Rust is more frequent than the opposite in a representative execution of the Compatibility Test Suite (CTS). However these results must be taken with extreme care because of the various limitations we explained (false detection of interface calls, function names collision, ...).

While the concrete results achieved by our evaluations are quite limited, we believe that the main contribution from this project comes from the methods and tools developed in order to understand how Rust interfaces with the existing codebase. In particular for the dynamic analysis, the small-scale proof of concept with the DoH module allows to understand very precisely how this component works and could be applied to other Rust components of interest; for the large-scale analysis, we are confident that some improvements to the implementation would allow interesting insights, also by producing other statistics with different methods than callstack sampling.

Bibliography

[1] Igor Bonifacic. *Android 12 has been released to the Android Open Source Project.* Engadget. Oct. 4, 2021. URL: https://www.engadget.com/android-12-android-open-source-project-173339442.html (visited on 06/03/2023).

[2] *Compatibility Test Suite.* Android Open Source Project. Sept. 13, 2022. URL: https://source.android.com/docs/compatibility/cts (visited on 06/08/2023).

[3] *Cuttlefish Virtual Android Devices.* Android Open Source Project. Dec. 16, 2022. URL: https://source.android.com/docs/setup/create/cuttlefish (visited on 06/08/2023).

[4] Ivan Lozano. *Integrating Rust Into the Android Open Source Project.* Google Online Security Blog. May 11, 2021. URL: https://security.googleblog.com/2021/05/integrating-rust-into-android-open.html (visited on 06/03/2023).

[5] Matthew Maurer and Mike Yu. *DNS-over-HTTP/3 in Android.* Google Online Security Blog. July 19, 2022. URL: https://security.googleblog.com/2022/07/dns-over-http3-in-android.html (visited on 06/05/2023).

[6] *Perfetto - System profiling, app tracing and trace analysis - Perfetto Tracing Docs.* Perfetto. URL: https://perfetto.dev/docs/ (visited on 06/08/2023).

[7] Mengtao Sun and Gang Tan. “NativeGuard: protecting android applications from third-party native libraries”. In: *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks.* WiSec’14: 7th ACM Conference on Security & Privacy in Wireless and Mobile Networks. Oxford United Kingdom: ACM, July 23, 2014, pp. 165–176. ISBN: 978-1-4503-2972-9. DOI: 10.1145/2627393.2627396. URL: https://dl.acm.org/doi/10.1145/2627393.2627396 (visited on 06/09/2023).

[8] Jeff Vander Stoep and Stephen Hines. *Rust in the Android platform.* Google Online Security Blog. Apr. 6, 2021. URL: https://security.googleblog.com/2021/04/rust-in-android-platform.html (visited on 06/09/2023).

[9] Jeffrey Vander Stoep. *Memory Safe Languages in Android 13.* Google Online Security Blog. Dec. 1, 2022. URL: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html (visited on 06/03/2023).

Appendix A

Rust module types count per top-level directory

A.1 /development : 1 rust modules

Module type	Nb occurences
rust_binary_host	1

A.2 /device : 8 rust modules

Module type	Nb occurences
rust_binary	2
rust_test_host	2
rust_binary_host	1
rust_bindgen_host	1
rust_library_host	1
rust_ffi_host	1

A.3 /external : 759 rust modules

Module type	Nb occurences
rust_test	344
rust_library	257
rust_defaults	39
rust_proc_macro	34
rust_library_host	31
rust_library_rlib	18
rust_test_host	14
rust_bindgen	5
rust_binary	5
rust_binary_host	5
rust_ffi_static	2
rust_protobuf	2
rust_ffi_shared	1
rust_ffi	1
rust_fuzz	1

A.4 /frameworks : 21 rust modules

Module type	Nb occurences
rust_library	8
rust_test	5
rust_bindgen	4
rust_ffi_static	2
rust_fuzz	2

A.5 /hardware : 4 rust modules

Module type	Nb occurences
rust_test	2
rust_binary	1

Module type	Nb occurences
rust_defaults	1

A.6 /packages : 152 rust modules

Module type	Nb occurences
rust_defaults	30
rust_test	28
rust_library	25
rust_binary	16
rust_test_host	13
rust_ffi_static	11
rust_bindgen	7
rust_library_rlib	6
rust_binary_host	4
rust_protobuf	3
rust_proc_macro	2
rust_library_host	2
rust_ffi_shared	2
rust_library_host_rlib	1
rust_ffi	1
rust_fuzz	1

A.7 /platform_testing : 1 rust modules

Module type	Nb occurences
rust_test	1

A.8 /prebuilts : 20 rust modules

Module type	Nb occurences
rust_toolchain_library_rlib	15
rust_toolchain_library	3
rust_stdlib_prebuilt_host	1
rust_defaults	1

A.9 /system : 133 rust modules

Module type	Nb occurences
rust_test	41
rust_library	28
rust_defaults	21
rust_bindgen	13
rust_binary	8
rust_test_host	6
rust_library_host	4
rust_binary_host	4
rust_fuzz	4
rust_protobuf	2
rust_proc_macro	1
rust_library_rlib	1

A.10 /tools : 22 rust modules

Module type	Nb occurences
rust_fuzz	10
rust_test	3
rust_ffi_host_static	2
rust_library_host	2
rust_defaults	2
rust_library	1

Module type	Nb occurences
rust_binary	1
rust_ffi_static	1