Technical Report

UCAM-CL-TR-991
ISSN 1476-2986

Number 991

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Porting a mix network client to mobile

Jacky W. E. Kung

December 2023

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 2023 by the author for the degree of Bachelor of Arts (Computer Science Tripos) to the University of Cambridge, St Edmund's College.

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-991*

Abstract

This project set out to investigate the feasibility of mix network clients on the mobile ecosystem. It considers the Android operating system, and *Nym*, a production-grade mix network infrastructure based on the abstract *Loopix* architecture first presented in 2017. The goal of the project was to produce a minimal working prototype, and present an evaluation of the trade-offs necessary for an efficient implementation in the Android ecosystem. Nym's client codebase written in Rust has been successfully ported over to Android after adjusting parts of the code and constructing the compilation toolchain. An exploration of the performance effects of compilation parameters and mixnet parameters is presented. Two extension tasks were completed: a semi-automated compilation pipeline, and further evaluation using measurements taken using the custom hardware provided by my supervisor. The repository also contains, as a side-product, a Rust library that provides a friendly interface between code that runs across the Rust and Kotlin languages.

Acknowledgements

I would like to extend my heartfelt gratitude to the following people who have supported me in this journey:

- **Daniel Hugenroth**, my supervisor, for having me as a project supervisee, and for providing invaluable guidance and constant encouragement throughout the course of this project. Without his direction, this work would not have been possible.
- **Zachery Liu**, for the insightful discussions regarding privacy in the modern world.
- **Chun Wei Yang**, for kindly proofreading this dissertation.
- **Luana Bulat**, by Director of Studies, for her kind support and supervision over my academic progress for the past three years.
- My friends and family, who have supported me in my undergraduate studies thus far.

Contents

Chapter 1

Introduction

This dissertation demonstrates the feasibility of implementing Nym clients for mobile devices, and presents development and performance insights useful for developers of similar systems. We begin with the motivations driving work of this nature.

1.1 Motivation

Privacy is the right to keep information about one's personal life secret [1]. Historically, it is declared as a fundamental human right under the Universal Declaration of Human Rights [2]. In the digital world of today, data protection and privacy laws are legislated in a majority of countries worldwide [3]: the 2018 General Data Protection Regulation (GDPR) is one such example.

Edward Snowden's revelations of prevalent global surveillance has sparked deep discourse [4]. Rogaway cautioned on its perils to society, and argues that privacy empowers individuals to express themselves without excessive scrutiny [5]. Nobody should live in constant fear that their private remarks may trigger punitive actions from higher powers; whistleblowers should not be discouraged by fear of retaliation from speaking up against injustice.

In digital communications, privacy takes the form of *anonymity*, which Pfitzmann defines as being unidentifiable within a group called the *anonymity set* [6]. While cryptographic encryption protects the *confidentiality* of data, the Internet infrastructure was not designed for anonymity: metadata such as IP addresses are sent as plaintext in IP headers, and can be used to identify communicating parties. To mitigate this, several anonymity-preserving network architectures have been invented and deployed.

This work centres around the mix network ("mixnet") architecture. However, current implementations do not support mobile devices, which present unique challenges such as providing offline support, working with a more restrictive OS API, and minimising power consumption. At the same time, computer ownership is declining and usage shifting towards mobile: worldwide, the average person spends 56.9% of their total Internet time on mobile devices [7]. Smartphones contain a trove of personal identifiable information, which are often shared via the Internet. This work hopes to spur further development of anonymity networks in the mobile ecosystem for the benefit of mobile users.

1.2 Anonymity Networks

Consider this common scenario: a user U wishes to communicate with a service S while remaining anonymous, that is, without any third-party or even S knowing that U was the origin of the communication.

1.2.1 Virtual Private Networks

A Virtual Private Network (VPN) architecture can be modelled as a node V on the Internet that tunnels U's traffic through itself:

$$\begin{aligned} U &\to V : \mathrm{Enc}_k(S, m) \\ V &\to S : m \end{aligned}$$

where k is a private key known to both U and V. I use the notation "$A \to B : m$" to represent an IP packet sent from source IP address A to destination IP address B, with payload m (typically assumed to be encrypted using HTTPS for *confidentiality*). U is anonymous to S, as S receives traffic that appears to originate from V. The fact that U is conversing with S is also hidden from U's local Internet Service Provider (ISP), who only sees that they are communicating with V. However, the VPN V is a trusted party who can identify both endpoints engaged in a conversation, as in Figure 1.1.

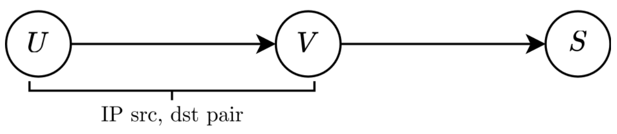


Figure 1.1: The general VPN network architecture.

1.2.2 Onion Routing

Onion routing is similar to VPNs, but no onion router has the full end-to-end picture [8]. Like VPNs, it is an overlay logical network over the existing physical Internet infrastructure. Prior to communicating with S, U chooses three onion routers O_1, O_2, O_3 from a public directory.

$$\begin{aligned} U &\to O_1 : \mathrm{Enc}_{s_1}(O_2, \mathrm{Enc}_{s_2}(O_3, \mathrm{Enc}_{s_3}(S, m))) \\ O_1 &\to O_2 : \mathrm{Enc}_{s_2}(O_3, \mathrm{Enc}_{s_3}(S, m)) \\ O_2 &\to O_3 : \mathrm{Enc}_{s_3}(S, m) \\ O_3 &\to S : m \end{aligned}$$

Using public-key exchange protocols, U establishes shared symmetric keys s_1, s_2, s_3 with the respective onion routers, such that the only parties with access to the private-key encryption and decryption functions $\mathrm{Enc}_{s_i}(\cdot)$ and $\mathrm{Dec}_{s_i}(\cdot)$ are U and O_i. The traffic flow is illustrated in Figure 1.2. Similarly to VPNs, U is anonymous to all nodes from O_2 onwards.

1.2.3 Traffic Analysis

In our threat model, we consider *global passive adversaries (GPAs)* who can inspect network traffic at the global scale. Their existence in the real world has been exemplified by Edward Snowden's document disclosures in 2013.

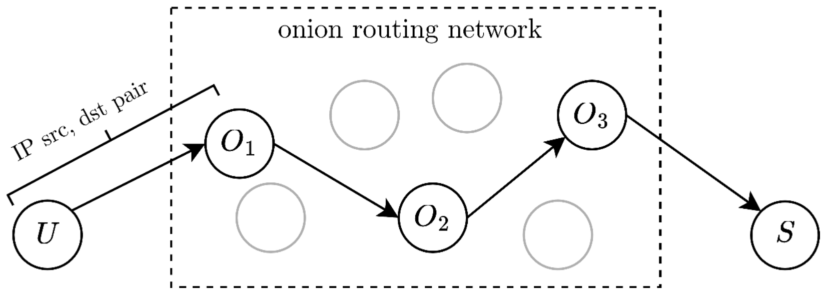


Figure 1.2: The general onion routing network architecture.

Because VPNs and onion routing do not obfuscate inter-packet ordering and timings, they are vulnerable to traffic correlation attacks, and do not provide anonymity in the presence of GPAs. In the VPN architecture, a GPA can correlate traffic flowing into and out of V to determine pairs (U, S) of communicating parties. In the onion routing architecture, a GPA can still, with more effort, correlate end-to-end traffic flowing out of any U and into any S to determine pairs (U, S) of communicating parties. Efficient attacks have been demonstrated [9]. We next introduce a anonymity-preserving architecture that has resistance against GPAs.

1.3 Mixnets

Formulated in 1979 by David Chaum, mixnets are similar to onion routing networks, but employ strategies to foil traffic analysis [10]. Instead of forwarding messages in first-in-first-out order (FIFO), each mixnet node maintains a buffer of incoming messages, and only forwards a reshuffled full buffer. This foils attacks that rely on FIFO forwarding order. By adding cover messages and artificial delay before forwarding messages, timing analysis is also hindered (§2.1.2–2.1.3). Additionally, every packet is independently routed through the mixnet via randomly chosen paths. Analysis by GPAs is thus harder in mixnets, in contrast to onion routing wherein packets from the same session take the same route. Figure 1.3 depicts an overview of a mixnet. We explore more details in the next chapter.

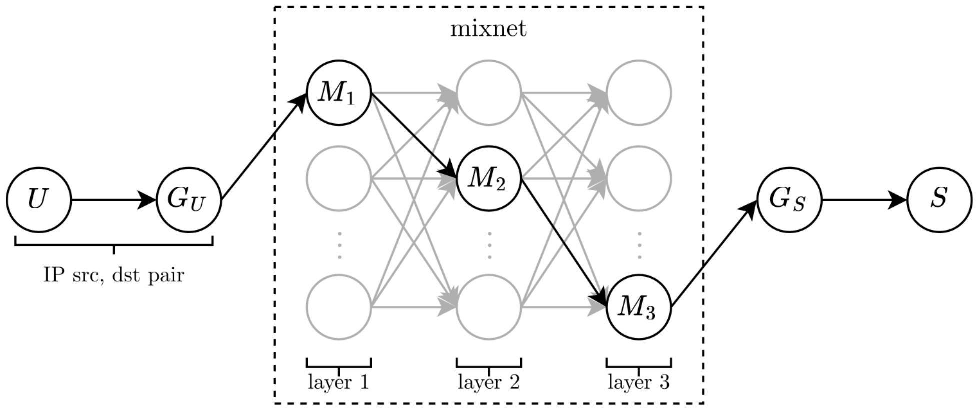


Figure 1.3: The general mixnet architecture (Nym). A message from U to S travels through a randomly chosen path (example in black). G_U and G_S are gateways of U and S respectively, and serve as their entry points to the mixnet.

However, U must be permanently online to receive messages. The wait for batches of messages at mix nodes also significantly increases end-to-end latency. These made the original mixnet designs unattractive for practical use.

1.3.1 Mixnets are increasingly necessary and practical

The existence of real-world GPA-like entities provide impetus for more widespread adoption of mixnets. The publication of *Loopix* in 2017 demonstrated an abstract mixnet architecture that reduces latency while strengthening privacy guarantees [11]. It also supports offline clients for when they have poor connectivity, which is mobile-friendly.

Nym is the first large-scale implementation of the Loopix architecture, with plans to partner with popular services such as Signal, Google and Brave and to support millions of users by 2024 [12]. However, as of October 2022, Nym clients are only available on the desktop environment. While Nym's roadmap for 2023 aims to encourage adoption for everyday use by further reducing latency, there are no publicly announced plans for a mobile Nym client.

1.4 Contributions

The main contributions of this work are:

1. A prototype implementation of the Nym mixnet for Android devices, demonstrating the practicality of such designs for mobile users.
2. An assessment of privacy properties of the implementation, and design trade-offs (in particular, message frequency) necessitated by the mobile platform.
3. `jvm_kotlin_typing`, a Rust library for interfacing with Kotlin data types. It allows Rust code to interact with a Java Virtual Machine (JVM) running Kotlin code while utilising Rust's unique language features. There is currently no suitable public library. This library is general for use in other projects.

1.5 Starting Point

I had no prior experience with mixnets or the Rust programming language. Thus, I read the Loopix paper over summer to gain basic understanding of the principles behind mixnets. I also studied the Rust e-book [13], and have implemented a web-application backend in Rust. I briefly experimented with Rust's built-in cross-compilation facility, but only to make Rust binaries compiled on my laptop (*x86-64*) run on a Raspberry Pi (*ARMv7*).

I have prior experience with Android programming. The most recent major project is writing a Kotlin application for my own personal expenses management in 2020. However, new Jetpack frameworks have arisen since then, which I use in this work. Furthermore, since it was pure Kotlin, I had not used Java Native Interface (JNI) or non-Kotlin code.

As for my supervisor, he has worked in an Android industry job before, has one year of experience coding in Rust, and has worked with the Nym codebase before.

Chapter 2

Preparation

In this chapter, I describe Loopix formally and outline how Nym differs from it. I then present the technologies used for implementation, and conclude with a requirements analysis.

2.1 Loopix Mixnet

The Loopix network consists of end users (e.g. sender U and receiver S), a global mixnet of mix nodes arranged in a stratified topology with a fixed number of layers [14], and "providers", a special type of mix node that mediate end users' interaction with the mixnet ("gateways" in Figure 1.3). Providers are densely connected with the first and final layers of the mixnet, and are responsible for storing messages for end users in case they are offline. Providers and mix nodes are run by different entities, and in Nym, they are rewarded with tokens to encourage good quality of service. The behaviour of participants in a Loopix network is specified by a set of parameters shown in Table 2.1.

Symbol	**Units**	**Meaning**	**Usage in Nym Config Files**
λ_P	s^{-1}	Mean rate of *real/drop cover (real)* packets sent by U	`average_packet_delay`
λ_L	s^{-1}	Mean rate of *loop cover* packets sent by U	`loop_cover_traffic_average_delay`
λ_D	s^{-1}	Mean rate of *drop cover* packets sent by U	N/A
μ	s^{-1}	Reciprocal of mean packet delay at each M_i	`average_packet_delay`
λ_M	s^{-1}	Mean rate of *loop cover* packets sent by M_i	N/A
f	s^{-1}	Fixed frequency at which U polls G_U for messages	N/A
k	–	Fixed number of messages in G_U's response to each poll	N/A

Table 2.1: Summary of Loopix Parameters. As discussed in DIFF4, in Nym, λ_P is the mean rate of *real/*loop *cover (real)* packets sent by U.

2.1.1 Bitwise Unlinkability: Sphinx Packet Format

Sphinx is an efficient (high payload to overhead ratio) mixnet packet format, used as a building block of Loopix. It provides *bitwise unlinkability* of packets across any node, such that there is no correlation between their bits before and after the transformation within a mix node. It also ensures that intermediate hops learn only essential routing information. I present the high-level concepts here; a deeper mathematical account is presented in Appendix B.

For each message m that U sends to S, U first pre-computes new shared secrets s_i independently with six parties: its own provider G_U, three randomly chosen mix nodes

M_1, M_2, M_3, S's provider G_S, and S. Sufficient information $\{I_1, \ldots, I_6\}$ is included in Sphinx headers for each of the six to reconstruct their respective shared secrets, using a mechanism similar to Diffie-Hellman key exchange. U then independently samples delays $\{d_1, \ldots d_5\}$ from $\text{Exp}(\mu)$, to be used at the corresponding intermediate nodes, as explained later in Section 2.1.2.

As shown in Figure 2.1, a Sphinx packet comprises of a header and a separately treated payload. The payload that U sends to G_U is simply m encrypted once for each of the six nodes on the path to S, as in onion routing, such that each hop can peel off only one layer of encryption. Similarly, the header is layer encrypted to expose to each node only information required I_i to derive its shared secret, its own delay d_i, and the next hop address.

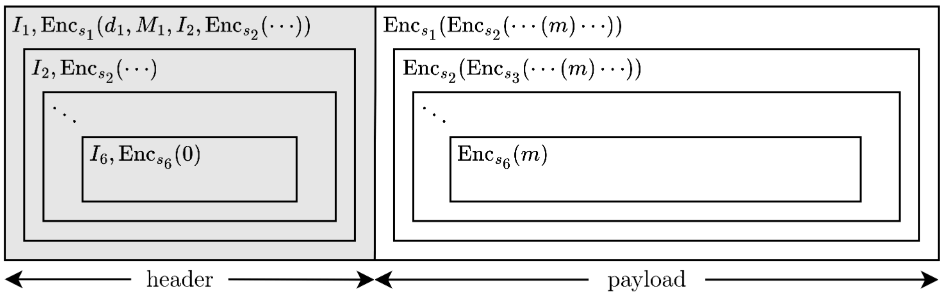

$I_1, \text{Enc}_{s_1}(d_1, M_1, I_2, \text{Enc}_{s_2}(\cdots))$
$I_2, \text{Enc}_{s_2}(\cdots)$
$\ddots$
$I_6, \text{Enc}_{s_6}(0)$
$\text{Enc}_{s_1}(\text{Enc}_{s_2}(\cdots(m)\cdots))$
$\text{Enc}_{s_2}(\text{Enc}_{s_3}(\cdots(m)\cdots))$
$\ddots$
$\text{Enc}_{s_6}(m)$
header
payload

Figure 2.1: Sphinx packet sent from U to G_U.

2.1.2 Metadata Unlinkability: Poisson Mix

The *Poisson Mix* is a specific mixing strategy that doesn't require a mix node's buffer to be full [15]. Upon arrival at a mix node, a Sphinx packet is transformed in preparation for forwarding to the next hop, but forwarded only after waiting for the delay d that U sampled from $\text{Exp}(\mu)$ when preparing the message for S.

It has the property that all packets currently held at a mix node have the same probability of being the next emitted one, regardless of how long they have been waiting. A mathematical explanation is provided in Appendix C.2. This drastically reduces end-to-end latency, and is the primary advantage of Loopix over other mixing strategies [11, §3.3]. Together with the Sphinx packet format, it immensely increases the difficulty for a GPA to link and deanonymise traffic flows.

2.1.3 Message Streams and Security Properties

A *Poisson process* describes random events with a known mean rate λ (i.e. λ events per unit time), such that the number of events occurring in a unit time interval is modelled by the distribution $\text{Pois}(\lambda)$, and the time interval between occurrences is modelled by $\text{Exp}(\lambda)$.

User "Push" Message Streams and Sender Anonymity

Each user emits three independent traffic streams to its provider, each a Poisson process:

- A stream of *real* messages, emitted at intervals $\sim \text{Exp}(\lambda_P)$. When a sampled interval has elapsed and there are no real messages to send, a *drop cover (real)* message is sent instead, destined to a random provider where it is dropped. This hides the fact that U is actively communicating at any time, and also increase the size of U's anonymity set, in the case where the current active user count in the mixnet is low.
- A stream of *loop cover* messages, emitted at intervals $\sim \text{Exp}(\lambda_L)$. These pass through the mixnet and are received by the sender again.
- A stream of *drop cover* messages, emitted at intervals $\sim \text{Exp}(\lambda_D)$. These are dropped at a random provider.

These combine to form an aggregate Poisson process (proof in Appendix C.1), which prevents an eavesdropper between users and their providers from distinguishing between messages from the component streams. This provides *sender anonymity*, where GPAs cannot tell which of two users are communicating with a service [6, 16].

User "Pull" Message Streams and Receiver Anonymity

Users U send pull requests at a fixed frequency f to their providers G_U, who reply with k messages, comprising any combination of *real* or *loop cover* messages addressed to U, plus cover messages to make up k messages as required. Because all packets use Sphinx, they are cryptographically indistinguishable, preventing GPAs from determining if there are any *real* messages in the set. This provides *receiver anonymity* [6, 16].

Mix Node Message Streams and Sender-Receiver Anonymity

Apart from forwarding Sphinx packets, each mix node also emits at intervals $\sim \text{Exp}(\lambda_M)$ a stream of *loop cover* messages, which decreases the probability of successfully linking messages across a mix node [11, Theorem 2]. This strengthens *sender-receiver anonymity*, making it harder for GPAs to identify the two end users of packets at a mix node.

Figure 2.2 depicts a summary of these Loopix message streams.

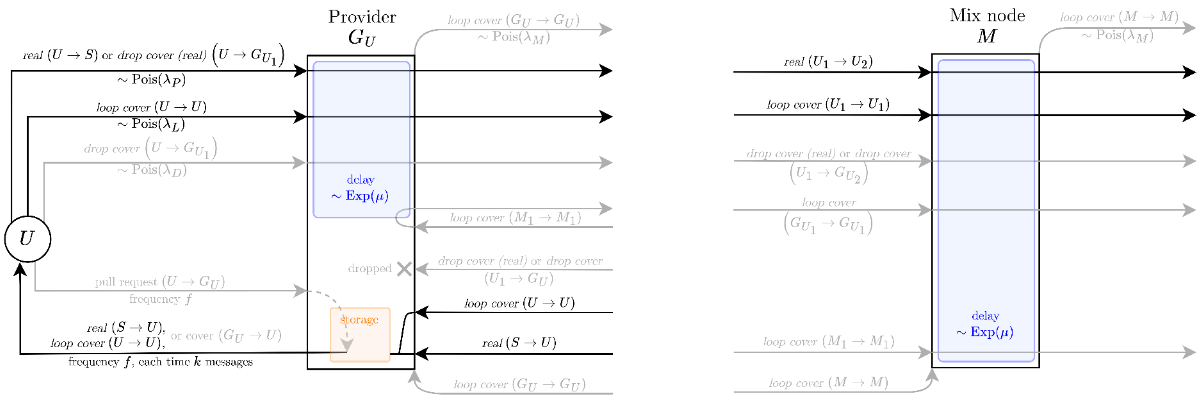

Provider
G_U
real $(U \to S)$ or *drop cover (real)* $\left(U \to G_{U_1}\right)$
$\sim \text{Pois}(\lambda_P)$
loop cover $(U \to U)$
$\sim \text{Pois}(\lambda_L)$
drop cover $\left(U \to G_{U_1}\right)$
$\sim \text{Pois}(\lambda_D)$
U
pull request $(U \to G_U)$
frequency f
real $(S \to U)$,
loop cover $(U \to U)$,
or cover $(G_U \to U)$
frequency f, each time k messages
delay
$\sim \text{Exp}(\mu)$
dropped
storage
loop cover $(G_U \to G_U)$
$\sim \text{Pois}(\lambda_M)$
loop cover $(M_1 \to M_1)$
drop cover (real) or *drop cover*
$(U_1 \to G_U)$
loop cover $(U \to U)$
real $(S \to U)$
loop cover $(G_U \to G_U)$
Mix node
M
loop cover $(M \to M)$
$\sim \text{Pois}(\lambda_M)$
real $(U_1 \to U_2)$
loop cover $(U_1 \to U_1)$
drop cover (real) or *drop cover*
$\left(U_1 \to G_{U_2}\right)$
loop cover
$\left(G_{U_1} \to G_{U_1}\right)$
delay
$\sim \text{Exp}(\mu)$
loop cover $(M_1 \to M_1)$
loop cover $(M \to M)$

Figure 2.2: Loopix message streams, with the greyed out ones not present in Nym. Numerical subscripts denote arbitrary nodes. When both sides of a "$\rightarrow$" use the same subscript, they both refer to the same node (e.g. $U_1 \rightarrow U_1$).

2.2 Nym Mixnet: Discrepancies from Loopix

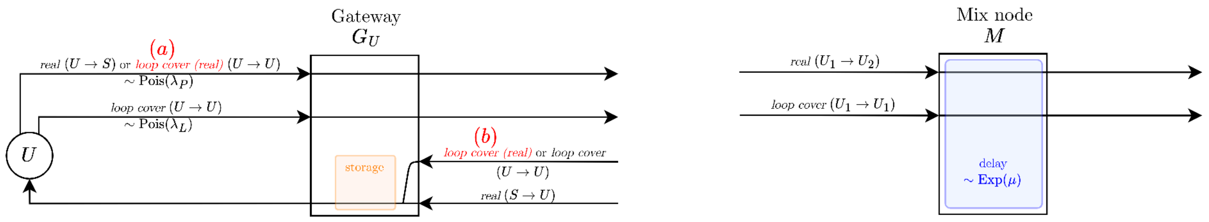

Gateway
G_U
(a)
real $(U \to S)$ or *loop cover (real)* $(U \to U)$
$\sim \text{Pois}(\lambda_P)$
loop cover $(U \to U)$
$\sim \text{Pois}(\lambda_L)$
U
storage
(b)
loop cover (real) or *loop cover*
$(U \to U)$
real $(S \to U)$
Mix node
M
real $(U_1 \to U_2)$
loop cover $(U_1 \to U_1)$
delay
$\sim \text{Exp}(\mu)$

Figure 2.3: Nym message streams. Name changes are indicated in red.

Although Nym is based on Loopix, closer inspection of the Nym codebase reveals that it is not a faithful implementation. I have identified the following differences:

DIFF1 "Providers" are renamed to "gateways", and no longer serve the purpose of a mix node. Gateways communicate with their users via symmetric encryption using long-term private keys, instead of using Sphinx packets (details in Appendix B.3).

DIFF2 Gateways also no longer delay packets. This is done for speed, as gateways also perform other functions such as ensuring end users have enough Nym credits to use the mixnet (details outside the scope of this work).

DIFF3 U no longer sends *drop cover* messages.

DIFF4 U replaces *drop cover (real)* messages with *loop cover (real)* messages. These loop back to U (a, b in Figure 2.3).

DIFF5 U no longer polls G_U at fixed frequency f, and G_U does not respond with a fixed number of packets k per poll (Table 2.1). Instead, packets are forwarded from G_U to U as soon as they are delivered at G_U, provided U is online.

DIFF6 Mix nodes do not send *loop cover* messages, despite this being promised in the whitepaper [17, §4.6]. This is marked as a "TODO" in line 10 of Listing 2.1. This weakens the *sender-receiver anonymity* property (§2.1.3) for Nym.

```
1   // nym/mixnode/src/node/listener/connection_handler/mod.rs:51
2   fn handle_received_packet(&self, framed_sphinx_packet: FramedSphinxPacket) {
3       match self.packet_processor.process_received(framed_sphinx_packet) {
4           Err(e) => /*...*/,
5           Ok(res) => match res {
6               MixProcessingResult::ForwardHop(forward_packet, delay) => {
7                   self.delay_and_forward_packet(forward_packet, delay)
8               }
9               MixProcessingResult::FinalHop(..) => {
10                  warn!("Somehow processed a loop cover message that we haven't implemented yet!")
11              }
12          },
13      }
14  }
```

Listing 2.1: Part of the code executed by mix nodes upon receiving a Sphinx packet.

Due to these differences, the only parameters in Table 2.1 that still apply in Nym are $\lambda_P, \lambda_L, \mu$.

2.3 Technologies Used

With the essential theory of mixnets covered, I now introduce the technologies used in the implementation chapter. I ensured all third-party dependencies used in my code are open-sourced under *permissive* licenses such as Apache 2.0, MIT and BSD. Specifically, I avoid using libraries licensed under *copyleft* licenses such GPL, which would force my derived work to use the same license. I have released my code publicly under the MIT license to make usage by other developers easy. The Nym codebase is licensed under Apache 2.0, which permits me to make and distribute modifications, as I have done as a forked repository. The fork continues to be licensed under Apache 2.0 for compatibility.

2.3.1 Rust and Community Crates

High-Level Language with Low-Level Guarantees. Nym is written in Rust, a high-level system programming language popular for writing fast and reliable code, such as in network applications. It has similar performance to C as it provides low-level memory operations. At the same time, it provides high-level programming ergonomics, guaranteeing memory safety and bug-free concurrency at compile-time through language features such as "borrow checking". For these reasons, Google writes critical Android native OS components in Rust [18].

Async Runtimes: Tokio community crate[1]. Rust did not initially ship with an *async runtime*, which is required to drive execution of asynchronous code and handle interprocess communication (IPC). The community has instead developed libraries, the most popular of which is *Tokio*. It is the de facto standard for writing code with structured concurrency today.

Async Runtimes: Tokio and the Nym Codebase. The Nym codebase makes extensive use of Tokio channels, which are concurrency-safe queues of which the transmitter and receiver ends can be passed independently throughout the codebase to pass transfer data between threads. They are conceptually similar to UNIX pipes. This presented a major challenge when working on both the main implementation and evaluation: it was difficult to track the web of data flows and to identify the correct places of interest to insert timestamp logging code. Appendix A shows one of the diagrams I drew to get a grasp of the complicated data flows. Table 2.2 shows the parts the Nym codebase that are relevant to this project, with numbers to give a general sense of its size.

Limited Support for Compiling to Android. The Rust compiler supports cross-compilation to a wide range of target architectures, including PC (typically *x86-64*) and Android (typically *ARM64 Android* for newer devices and *ARMv7a Android* for older devices). However, Android targets are supported by Rust only in the "Tier 2" category, wherein Rust code is guaranteed to cross-compile, but the output may contain subtle bugs that cause unexpected crashes at runtime [19]. In contrast, PC targets are supported in the "Tier 1" category, wherein Rust code is guaranteed to cross-compile and work as

[1]A "crate" is a Rust compilation unit, and is analogous to a Python package. I use "community crate" to refer to Rust repositories that are not developed by me.

Repository	**Line Count**	`fn` **Count**	`struct` **Count**
.			
`+-- clients` `|` *users use a client to connect to a gateway*			
`|   +-- client-core` `|   |` *library code for clients*	7263	440	71
`|   +-- native` `|` *code that runs on a user client; ported over to Android*	2766	143	15
`+-- common`			
`|   +-- client-libs`			
`|   |   +-- gateway-client` `|   |   |` *user clients user this to talk to a gateway*	1261	58	8
`|   |   +-- mixnet-client` `|   |` *gateways use this to talk to mix nodes*	332	12	4
`|   +-- mixnode-common` `|   |` *library code for gateways and mix nodes*	1357	75	17
`|   +-- nymsphinx` `|` *Sphinx operations*	7005	343	40
`+-- gateway` `|` *code that runs on a gateway*	5345	319	43
`+-- mixnode` *code that runs on a mix node*	2575	168	34
$\sum$	27904	1558	232

Table 2.2: Line (`.rs` files only, including comments), function definition and struct definition counts in the unmodified Nym codebase at commit `d92d687`.

intended by the programmer. As I explain later in Section 3.2, this presented significant problems when constructing the compilation toolchain.

2.3.2 The Android OS

The mobile platform of choice is Android primarily because it is the most popular one [20]. It is also open-source, which allows me to inspect the kernel source code for low-level OS operations. It is also more amenable to measurements and benchmarking than other popular OSes. As the project focuses on the general differences between desktop and mobile platforms, the results obtained should in principle translate well to iOS and other mobile platforms.

The Android OS runs a modified form of the PC Linux kernel. Applications are only exposed to a restrictive subset of OS APIs. In particular, files cannot be stored in arbitrary parts of the filesystem, since most locations are writeable by root only, and applications do not run with root permissions. Implementing offline storage for Nym on Android requires adherence to Android's file storage APIs exposed through Kotlin, which presented a core challenge that I describe in Section 3.5.1.

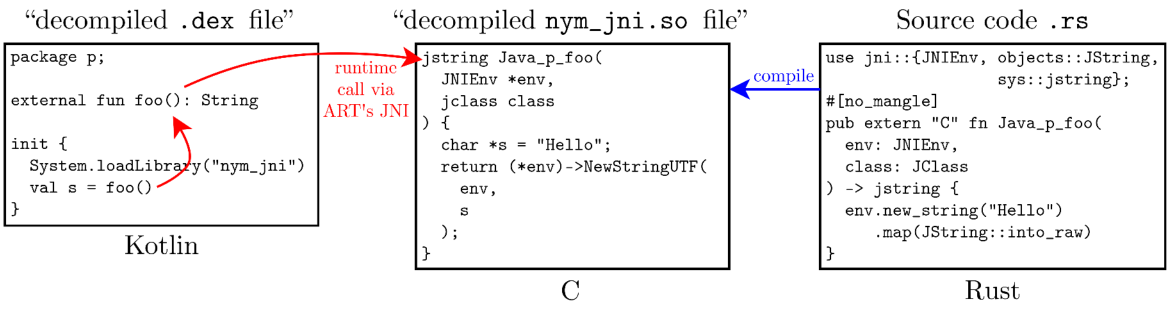
Additionally, the Nym client is designed as a command-line program. On PC, it can be launched from any shell prompt and runs interrupted unless terminated by the out-of-memory killer, only under extreme resource demands. On Android, launching arbitrary processes is more complicated, and resource management policies (CPU and memory) are more aggressive. Keeping the Nym client alive in spite of these proved to be another core challenge, which I describe in Section 3.9.1.

2.3.3 Kotlin and Java Native Interface

For development of the prototype, I use Kotlin, the Java Virtual Machine (JVM)-based language recommended by Google for Android development. Rewriting Nym's codebase in Kotlin is not desirable as the Nym codebase is huge and already written in Rust. Reuse of the Rust code is ideal both for development speed and maintainability, as there is no need for anyone to painstakingly maintain a separate "`nym-kotlin`" repository that is in sync with the main codebase.

However, Rust and Kotlin are quite different. Whereas Kotlin code uses a garbage collector at runtime, Rust's compile-time borrow checker guarantees no dangling pointers and hence eliminates the need for one. Kotlin code is compiled into bytecode meant to be executed on the *Android RunTime (ART)*, while Rust is compiled directly to native code (`.so` library file in Figure 3.1).

The Java Native Interface (JNI) is a foreign function interface framework that allows developers to write Kotlin code that calls C code, and write C code that invokes functionality of the ART (e.g. to instantiate Kotlin objects). With the help of the Rust `jni` community crate, the C code that Kotlin code calls at runtime is generated from Rust source code during compilation, as shown in Figure 2.4.



```
package p;

external fun foo(): String

init {
  System.loadLibrary("nym_jni")
  val s = foo()
}
```

```
jstring Java_p_foo(
  JNIEnv *env,
  jclass class
) {
  char *s = "Hello";
  return (*env)->NewStringUTF(
    env,
    s
  );
}
```

```
use jni::{JNIEnv, objects::JString,
                  sys::jstring};
#[no_mangle]
pub extern "C" fn Java_p_foo(
  env: JNIEnv,
  class: JClass
) -> jstring {
  env.new_string("Hello")
     .map(JString::into_raw)
}
```

Figure 2.4: JNI acts as a shim between Kotlin and Rust. The "decompiled `.dex` file" and "decompiled `.so` file" are human-readable versions of installed artefacts on Android (see Figure 3.1), for illustrative purposes.

2.3.4 Android Frameworks and Libraries

Jetpack. *Jetpack* is an umbrella of libraries that speed up Android development. The *Compose* framework allows declarative specification of the prototype's user interface (UI) layout. The alternative to this is using the legacy XML file format, which is much more tedious and error-prone. *WorkManager* is a framework to handle background tasks. *Room* is a high-level library for working with SQLite databases. It abstracts away boilerplate of common database operations, and presents data as `Flows`, which interoperate nicely with the *Compose* framework and automatically refreshes the UI when database values update. The alternative of using low-level cursors is tedious and error-prone. Saving Nym contacts and messages in a text file would also have required manual implementation of UI synchronisation, on top of the four database properties: atomicity, consistency, isolation and durability.

Foreground Services. A *Foreground Service (FGS)* is a type of background process supported by the Android OS, and is the chosen alternative to *WorkManager* and *Bound Services*. It is simpler and more closely matches the process model of PCs, where processes are allowed to execute continuously in the background without intervention from the OS, except during times of extreme resource demand.

Android Debug Bridge. *Android Debug Bridge* (ADB) is the primary means of sending commands from a shell on my development PC to Android devices. I use it extensively to construct a semi-automatic testing pipeline in Section 3.10.

2.4 Requirements Analysis

The requirements gathered here will guide the evaluation chapter. Hard requirements must be fulfilled in its entirety; soft requirements may require trade-offs.

Hard Requirements

RH1 The prototype must run on Android continuously, allowing the user to send and receive messages across the Nym test network to/from a PC without crashing. This is the bare minimum of a prototype.

RH2 The prototype must be indistinguishable, to GPAs observing network traffic, from a PC Nym client with regards to timing characteristics of message streams (unless this is impossible). A successful port should not deviate from the expected behaviour of a PC client.

Soft Requirements

RS1 The prototype should minimise power consumption while achieving privacy protection of the mixnet. My Pixel device should be able to last a day with the prototype running continuously in the background.

RS2 The prototype should not cause device to lag. It is expected to maintain below 50% CPU utilisation for extended periods of time.

RS3 The prototype should not take up more space than popular messaging applications and anonymity network clients.

RS4 The prototype should be compatible with the majority of Android devices, in line with the philosophy of promoting widespread usage of mixnets.

RS5 Implementation should be structured such that code maintenance is sustainable, in line with good software engineering practices such as testing. This will not appear in the evaluation chapter, but will guide the implementation.

Chapter 3

Implementation

In this chapter, I outline and justify the strategies employed for the implementation of the prototype, starting with a review of the codebase. I then outline the difficulties with the compilation toolchain, before describing the details of individual components, which are the result of iterations of design experimentation, necessitated by the general lack of online documentation.

3.1 Repository Overview

Repository	Language	Implementation Aspects
.		
+-- `nym-jni-android` | *main implementation*		
| +-- `nym-android-port` (§3.9) | | *Nym client for Android*	Kotlin	self-authored, using *Jetpack* libraries
| +-- `nym-jni` (§3.4–3.5) | | *glues* `nym` *with* `nym-android-port` | | *contains* `jvm_kotlin_typing` *library*	Rust	self-authored, using `jni` community crate
| +-- `nym-pc` (§3.8) | *Nym client for PC*	Rust	minor modifications of example code from `nymtech/nym`
+-- `nym-data-analysis` | *Jupyter notebooks*	Python	self-authored, using data analysis libraries (e.g. `pandas`, `seaborn`)
+-- `nym` (§3.7) *fork of* `nymtech/nym` based at commit `d92d687` *(Table 2.2)*	Rust	minor additions of timestamp logging code

Table 3.1: Overview of repositories.

In this chapter and the next, I refer to the original Nym repository as `nymtech/nym`, and my fork of it as `nym`. Table 3.1 outlines the three repositories that I maintain for this dissertation. I've chosen a mix of the mono-repo and multi-repo strategies as both have their advantages. The main implementation is kept together in a mono-repo as its sub-repositories are strongly coupled with one another. It contains two branches, `main` and `probe-effect-evaluation`, that I frequently switch between during evaluation, and all three sub-repositories behave slightly differently in these two branches. These two branches also exist for `nym`, but as it is a fork of a public repository, I maintained it as a separate repository.

While `nym-data-analysis` is coupled with the rest of the codebase (it makes assumptions about formatting of timestamps), it doesn't change between `main` and `probe-effect-evaluation`, and is thus maintained separately.

3.2 Compilation Toolchain

Initially, I used a *Gradle*[1] plugin called `rust-android-gradle` that allows an Android application to call methods defined in a Rust library [21]. This matches this project's requirements, but allows only one Rust crate to be bound to the Android application. With the benefit of hindsight, this is not an issue, but creating a solution that works for more than a single crate also led to a deeper understanding of the compilation pipeline.

3.2.1 Toolchain Construction

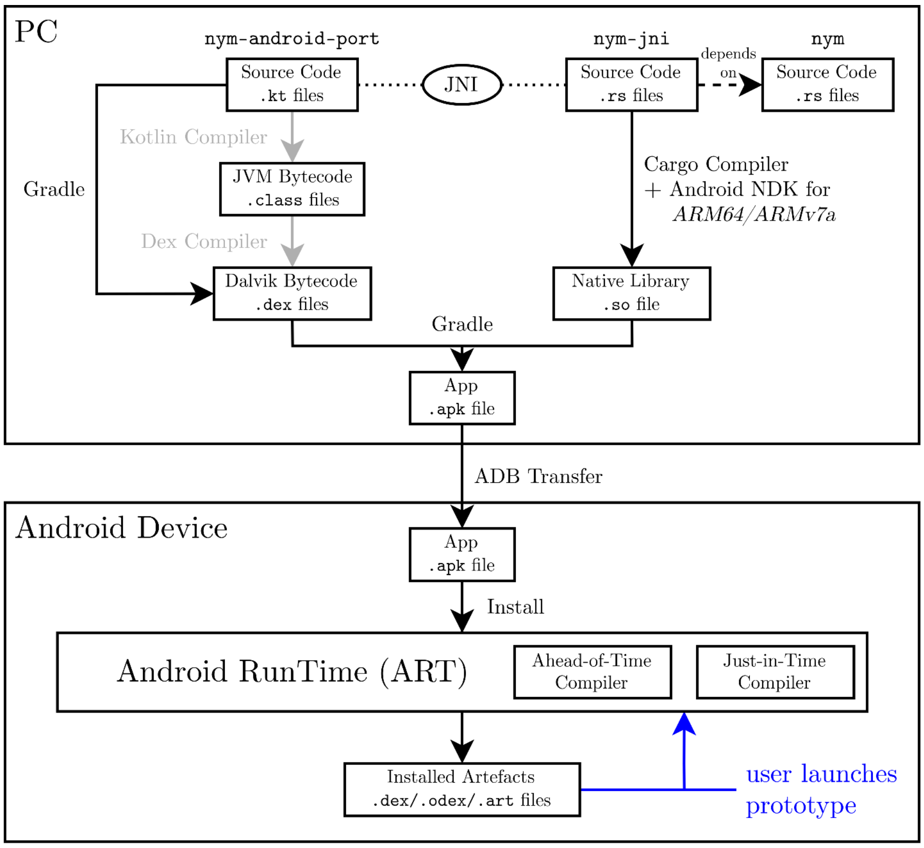


Figure 3.1: The compilation toolchain for an Android application that uses Rust code (Nym client). After installation, information from the `.dex` files and `.so` files are stored as `.vdex`/`.odex`/`.art` files.

Figure 3.1 presents the big picture of the compilation toolchain I put together. On the PC, application development in Kotlin (`nym-android-port` sub-repository) and Rust (`nym-jni` and `nym` crates) uses JNI (§2.3.3). Compilation then proceeds in two stages. First, Rust's *Cargo* package manager is invoked to cross-compile Rust code in the `nym-jni` crate into a target-specific (*ARM64* or *ARMv7a*) `libnym_jni.so` shared library file under the output folder in the `nym-jni` crate ($1, 2$ in Figure 3.2). I copy this into the `jniLibs` folder in `nym-android-port`, where the Gradle build tool expects to find native code (manual copying is automated later in Section 3.10). Secondly, Gradle is invoked to

[1]Gradle is the default build tool for Android development. Dependencies of an Android application are specified through a `build.gradle` file in the project repository (Figure 3.2).

compile Kotlin code in the `nym-android-port` sub-repository into Dalvik Bytecode files, and bundle them together with `libnym_jni.so` into a Android Package Kit (APK) file.

On the Android device, the APK is received through ADB and installed, converting its contents into installed artefacts via a mix of ahead-of-time compilation and runtime just-in-time compilation [22]. When the prototype is launched either programmatically or through the UI, the installed artefacts are invoked by the ART to execute the prototype.

```
.
└── nym-jni-android/
    ├── nym-android-port/app/src/
    │   ├── build.gradle ............... specifies dependencies and arguments to Gradle compiler
    │   ├── androidTest/ ........................................ instrumentation tests (§3.4.2)
    │   └── main/
    │       ├── java/com/.../nymandroidport/ .................... application Kotlin code (§3.9)
    │       └── jniLibs/
    │           ├── arm64-v8a/
    │           │   └── libnym_jni.so ..........................................copied from (1)
    │           └── armeabi-v7a/
    │               └── libnym_jni.so ..........................................copied from (2)
    └── nym-jni/
        ├── .cargo/config.toml .......................... specifies arguments to Cargo compiler
        ├── Cargo.toml ...................................................specifies dependencies
        ├── src/ .................................................. application Rust code (§3.5)
        │   ├── android_instrumented_tests/ ..................... instrumentation tests (§3.4.2)
        │   ├── clients_native_src/ ............................copies of files from nym (§3.5.1)
        │   └── utils/ ............................... Kotlin types interoperability library (§3.4)
        └── target/
            ├── aarch64-linux-android/release/
            │   └── libnym_jni.so ......................................................... (1)
            └── armv7-linux-androideabi/release/
                └── libnym_jni.so ......................................................... (2)
```

Figure 3.2: Directory structure of selected parts of the `nym-jni-android` repository. As part of the compilation toolchain, the target-specific shared library `libnym_jni.so` is copied from $(1, 2)$ into the `jniLibs` folder for the corresponding architecture.

Since many parts of this toolchain can break, a general advice I followed throughout development is to make incremental progress, taking small steps at a time. This allowed me to quickly isolate breaking changes and fix them. I started with a skeletal `nym-android-port` sub-repository containing minimal UI Kotlin code. A skeletal `nym-jni` crate was also created with only minimal Rust code that called some simple methods in the `nym` crate. I then verified that the compilation toolchain worked before continuing with the porting process. The next section describes challenges faced during this undertaking.

3.2.2 Toolchain Debugging

Cross-compilation from the host architecture (*x86-64*) to "Tier 2" target architectures (*ARM64* or *ARMv7a*) failed for a multitude of reasons. Pinpointing causes proved to be challenging because of cryptic compiler and JNI errors. Adding to the difficulty, these errors occurred together and were debugged simultaneously. Furthermore, workarounds are not well-documented online, especially since cross-compilation to Android is not a common use-case. As such, the solutions presented to the problems below are the result of much trial and error.

Cross-compilation requires a pointer to Android NDK's Clang linker. The documentation page for Rust cross-compilation mentions the need to use the Android Native Development Kit (NDK), which contains tools required for an Android application to interact with C code. Importantly, the NDK provides target-specific *Clang* linkers that Cargo requires for cross-compilation. I specified the path to the linkers in `nym-jni`'s `.cargo/config.toml` file (Figure 3.2), using the `linker` attribute under the `[target.aarch64-linux-android]` and `[target.armv7-linux-androideabi]` headers, in order for Cargo to locate them.

Additional requirements of the build script of `nym`'s dependency `ring`. The `nym` crate depends on the `ring` community crate for cryptographic operations. Since it builds on C and assembly code, it uses a custom Rust build script, whose documentation mentions the need to specify paths to the target-specific NDK Clang linkers and LLVM archivers [23, 24]. I did this via the `.cargo/config.toml` file, using the `CC_armv7_linux_androideabi` and `AR_armv7_linux_androideabi` attributes under the `[env]` header. This complication was documented but difficult to pinpoint, because `ring` is a transitive dependency of the `nym-jni` crate which I work directly on.

Compilation of the `nym`'s dependency `openssl`. By default, Rust crates that depend on the `openssl` crate are compiled to use dynamic links to the host's (PC) installation of `openssl`. As mobile devices do not typically have an OpenSSL installation, the *vendored* crate feature of the `openssl` crate must be used, in order Cargo to compile and statically link a copy of OpenSSL built for the target Android architecture, when producing `libnym_jni.so`. I specified `openssl = { ..., features = ["vendored"] }` in `nym-jni`'s `Cargo.toml` file. Despite this, the compilation of `openssl` only succeeds when the `RANLIB_aarch64_linux_android` and `RANLIB_armv7_linux_androideabi` attributes under the `[env]` in `.cargo/config.toml` are specified to point to NDK's target-specific LLVM archive indexers (`llvm-ranlib`). This final complication is undocumented.

Rust's previous incompatibility with Android NDK. This project uses the latest Long-Term Support (LTS) version of NDK, `r25`. As of November 2022, there was an open issue in the Rust community: the Android NDK version upgrade from `r22b` to `r23` introduced a breaking change where the previous `libgcc` stack unwinding implementation was replaced with `libunwind`. The Rust compiler however had not yet officially supported this change, resulting in an obscure bug where using NDK `r23` or newer caused crates to compile without errors, but fail at runtime with an arcane JNI `UnsatisfiedLinkError` [25]. During the implementation of `nym-jni` (§3.5), I inlined methods from its dependencies and commented lines out one by one until I identified the dependency responsible for error, and then resorted to esoteric hacks to get the project to execute [26, 27]. Pinkus, a community contributer to the Rust language, provides a good summary of this issue [28, 29].

This incompatibility was retrospectively fixed in January 2023 in Rust version `v1.68.0`, and I later removed the stopgap hacks by upgrading Rust from `v1.62.0` to `v1.68.1` in March.

3.3 Developer Experience

On Android devices, the UNIX `stdout` and `stderr` file descriptors are redirected to `/dev/null`. This causes the output of `println` calls in both Kotlin and Rust to not appear on Android's system messages viewer, `logcat`. In Kotlin, the logging utility class `android.util.Log` works seamlessly with `logcat`, and also provides support for different log levels (ranging from verbose to error).

Getting log messages from Rust to appear on `logcat` is more involved. I use the `log` community crate to write log messages, as it provides support for log levels. These are output to `stdout`. I then call the `init_once` method from the `android_logger` community crate at application startup, to send the logs to `logcat`, as demonstrated in Listing 3.1.

```
android_logger::init_once(
    android_logger::Config::default()
        .with_min_level(log::Level::Trace)
        .with_tag("nym_jni_log"),
);
log::error!("Test!");
// logcat output (simplified):
// MM-dd HH:mm:ss.SSS <PID> <TID> E nym_jni_log: Test!
```

Listing 3.1: One of the first lines of Rust code executed by my prototype on an Android device, linking Rust `log` messages to `logcat`.

For error handling, Kotlin uses exceptions. Runtime exceptions produce a stack trace on `logcat`. However, Rust does not provide exceptions, and instead represents fallible computations using a `Result<`T`,`E`>` enum type, whose value is either of type `Ok(`T`)` for successful computations, or `Err(`E`)` for failures, where T, E are arbitrary data types. In order for `Err(`E`)`s to be handled by the Android application, a JVM `RuntimeException` must be manually raised using a JNI method provided by the `jni` community crate. Because such code is repetitive and error-prone, I created Rust macros to automatically raise a `RuntimeException` when a function returns an error variant, as illustrated in Listing 3.2. These macros are contained in the `jym_kotlin_typing` library which I describe next.

3.4 `jvm_kotlin_typing` Rust Library

Data types between Rust and Kotlin are not fully compatible, and are shimmed by C types via JNI, as in Figure 2.4. Consider Listing 3.3 where a Kotlin function `printInRust` is implemented in C using JNI. It takes as arguments a 32-bit non-nullable signed integer `Int` and a 64-bit nullable unsigned integer `ULong?`. At runtime, these values are mapped

```
call_fallible!(f, env, class, f_arg1, ..., f_argN);
// expands into
//     if let Err(str) = f(env, class, f_arg1, ..., f_argN) {
//         env.throw(str).expect("Rust: Unable to throw Kotlin Exception");
//     };
```

Listing 3.2: Example usage of my `call_fallible!()` macro which automatically raises a JVM exception on failure of the argument function `f`.

to C types by the Kotlin standard library. On the Rust side, the `jni` community crate exposes the respective C types `jint` and `jobject`. The last step is to map these into Rust types `i32` and `Option<u64>`. However, this may involve calling low-level JNI methods exposed by `jni`, and is repetitive and verbose in general. I therefore implemented `jvm_kotlin_typing`, a Rust library presenting a higher-level API built upon the `jni` community crate, for concise conversion between C-typed and Rust-typed values.

```
package p;

external fun printInRust(v1: Int, v2:
↪  ULong?)
```

```
use jni::{JClass, JNIEnv, objects::sys::jint};
use crate::jvm_kotlin_typing::{consume_kt_int,
↪  consume_kt_nullable_string};

#[no_mangle]
pub extern "C" fn Java_p_printInRust(
    env: JNIEnv,
    _: JClass,
    v1: jint,     // Int in Kotlin
    v2: jobject,  // ULong? in Kotlin
) -> Result<(), JNIError> {
    let v1: i32 = consume_kt_int(value);
    let v2: Option<u64> = consume_kt_nullable_ulong(env,
    ↪  v2)?;
    log::info!("Kotlin sent: {}, {}", v1, v2);
}
```

Listing 3.3: Example usage of `jvm_kotlin_typing`. Calls to the Kotlin function on the left passes the arguments `v1`, `v2` through JNI to the Rust function on the right.

3.4.1 Mapping between Kotlin, C and Rust Types

Table 3.2 presents the mapping from Kotlin types T_{Kotlin} to their equivalents in C and Rust, T_{C} and T_{Rust} respectively. For each T_{Kotlin}, I implemented two `jvm_kotlin_typing` functions `consume_kt_`T_{Kotlin} and `produce_kt_`T_{Kotlin} of the following signatures:

$$\texttt{consume\_kt\_}T_{\text{Kotlin}} : T_{\text{C}} \to T_{\text{Rust}}$$
$$\texttt{produce\_kt\_}T_{\text{Kotlin}} : T_{\text{Rust}} \to T_{\text{C}}$$

which decodes and encodes values from and to JNI respectively. In the cases where the these functions may fail, a `Result<`T`, JNIError>` type is returned instead of T, as with idiomatic Rust.

Booleans. The JNI specification represents booleans using the C `jboolean` type, which is a 8-bit unsigned integer. The values `false` and `true` are represented as `0` and `1` respectively. My Rust function `consume_kt_boolean` thus takes a `jboolean` (provided by `jni`, aliases to `u8`) and returns `false` if the value is `0`, and `true` otherwise.

T_{Kotlin}	T_{C}	**Implementation of** `consume_kt_`$T_{\text{Kotlin}} : T_{\text{C}} \to T_{\text{Rust}}$	T_{Rust}
`Boolean`	`jboolean`	`1` $\mapsto$ `true`, `0` $\mapsto$ `false`	`bool`
`Int`	`jint`	Identity	`i32`
`Int?`	`jobject` (`java/lang/Integer`)	Null check, then delegate to low-level JNI methods	`Result<Option<i32>, JNIError>`
`UInt`	`jint`	Re-interpret bits	`u32`
`UInt?`	`jobject` (`kotlin/UInt`)	Null check, then delegate to low-level JNI methods	`Result<Option<u32>, JNIError>`
`String`	`jstring`	Delegate to low-level JNI methods	`Result<String, JNIError>`
`String?`	`jobject`	Null check, then use above `jstring` $\mapsto$ `Result<String, JNIError>`	`Result<Option<String>, JNIError>`

Table 3.2: Mapping between Kotlin, C and Rust types through `jvm_kotlin_typing`'s API. The full table is presented in Appendix D.

Primitive numbers. Primitive numbers types are numeric types supported by the JNI specification: 8–64 bit signed integers, floats and doubles. I present here the 32-bit signed integer case, which generalises to the other cases. For the non-nullable Kotlin `Int`, my Rust function `consume_kt_int` is simply the identity function mapping `jint` (aliases to `i32`) to `i32`. While this is functionally redundant, I wanted to present a uniform API to Rust programmers, such that all values arriving via JNI should first pass through a `consume_kt_`T_{Kotlin} function before usage within Rust. For the nullable Kotlin `Int?`, the JNI representation is the `jobject` type that is either the null pointer or points to a `java/lang/Integer` instance. Therefore, my Rust function `consume_kt_nullable_int`:

- returns value `Ok(None)` if `null` was passed from Kotlin,
- returns value `Ok(`v`)` where v is a `u32` obtained by using low-level JNI methods exposed by `jni` to invoke the `java.lang.Integer.intValue` method on the `jobject` instance, or
- returns value `Err(_)` (fails) if Kotlin passed another object that is not of type `Int?`.

Unsigned numbers. Kotlin implements `UInt` as a wrapper *inline class* over an inner field `data` of type `Int`, with the same bit representation [30]. For instance, the `UInt` $2^{32}-1$ is stored as the `Int` -1, since they both have the same bit representation `0x11111111`. Since JNI only supports `jint`, I thus have to "re-interpret" the signed bits back to unsigned. The remaining procedure is similar to the primitive numbers case above, except the nullable case uses `kotlin/UInt` objects instead of `java/lang/Integer` objects. The representation of Kotlin inline classes in JNI is poorly documented online, and I discovered it by trial and error.

Strings and Characters. These are actually represented as "modified UTF-8 strings" in C [31]. Conversion to and from a Rust `String` is non-trivial and performed via the low-method JNI method `get_string_utf_chars`. My function `consume_kt_string` directly delegates to it. My function `consume_kt_nullable_string` additionally performs a null check and is fallible, similar to `consume_kt_nullable_int`. I left my methods that deal with Kotlin's `Char` and `Char?` types unimplemented, as their implementation involves dealing with this technical detail, and would take an unjustifiable amount of time with no benefit to this project.

My implementation of the `produce_kt_`T_{Kotlin} functions are inverses of the above (Appendix D). For Kotlin types represented as `jobject`, I construct these objects in the JVM using low-level JNI methods. All of my functions are documented.

3.4.2 Android Instrumentation Tests

To test the correctness of my library, I wrote 140 Android instrumentation tests, which are installed and executed on an Android device using Gradle but controlled from the PC (unlike unit tests which execute on PC). They pass values between Kotlin and Rust in both directions, testing each type's minimal and maximal values, as well as the null case where applicable. This provides confidence in my library's correctness, as I use in `nym_jni` (Section 3.5).

The tests are administered from Kotlin. When passing a value from Kotlin to Rust, the Rust code generates a string that reports what it saw (both value and Rust type), which is sent back to Kotlin, where equality is tested against an expected report. When passing a value from Rust to Kotlin, the Kotlin code simply checks for equality with the expected value.

3.5 `nym-jni` Crate

The `nym-jni` crate is conceptually a wrapper around two Rust methods in the `nymtech/nym` crate: `init(...)` and `run(...)`, making them suitable for invocation from Kotlin. Since `nymtech/nym` is under active development, I work with a snapshot of the repository, corresponding to Nym version `1.1.4` (commit `d92d687`, 20 December 2022).

The functionality of `init(...)` and `run(...)` are largely untouched, but tweaks were necessary for them to execute properly on Android. `init(...)` creates a new Nym client (with a Nym address), registers it with a randomly chosen gateway, and saves Nym configurations in a `config.toml` file on the device, at a given location. `run(...)` runs the Nym client as a continuous background process. I also wrote a `topLevelInit(...)` method which executes first on application startup, in order to setup logging functionality previously seen in Listing 3.1.

3.5.1 Porting Strategy Discussion

The `nymtech/nym`'s `init(...)` method saves the Nym configuration `config.toml` into the home folder at `~/.nym/<client-id>/config/config.toml`, a destination which is hardcoded. However, Android applications are not permitted to write to this path; they are only permitted to save to *app-specific storage*, whose path is accessible on Kotlin via `applicationContext.filesDir.absolutePath`. There is thus a need to override certain subroutines in `init(...)` to support receiving this path from Kotlin and using it instead of the hardcoded path.

Because the `init(...)` method in `nymtech/nym` is not amenable to the object-oriented inheritance pattern, the first strategy adopted was to copy lines from `init(...)` in `nymtech/nym` into a single method in `nym-jni`, then add code to accept the path from Kotlin. However, these lines reference structs and methods that have crate-local visibility in `nymtech/nym` (`pub(crate)` visibility modifier) and could not be accessed from `nym-jni`.

For `nym-jni` to be part of the `pub(crate)` scope, a lot of code had to be copied over, which became unwieldy.

The next and current strategy is to copy entire files, adapting lines as necessary. Because these files came from the `clients/native/` folder in `nymtech/nym`, I've parked them under the `clients_native_src/` folder in `nym-jni` (Figure 3.2). The alternative of publishing a new branch in `nymtech/nym` is not feasible as I do not have the permissions to do so. The current strategy also allows me to manage all Android-related Nym code in one `nym-jni` crate.

My guiding principles were to mirror the original directory structure, and to make minimal modifications (including ignoring linter warnings). Each copied file contains a comment that explains why the copying was necessary (e.g. to get around `pub(crate)` restrictions) and where the original file can be found within `nymtech/nym`. All structs and methods in copied files are prefixed with a comment describing the modifications I made (or lack thereof). This makes tracking differences with `nymtech/nym` easier, and improves readability for `nymtech/nym` maintainers who are familiar with the original crate. The necessary minimal modifications are described next.

Avoiding prop drilling. The Rust code that controls the destination of the Nym configuration file is hidden behind layers of calls to functions and Rust trait implementations. Passing the path received from Kotlin to this code would require modifying function signatures down a call chain ("prop drilling"), which would modify a lot of code. Instead, environment variables were exploited, since they are a second form of arguments to any function. `topLevelInit(...)` receives the path from Kotlin and saves it to a runtime environment variable under a key that uniquely describes where it is used in code. When `init(...)` saves the Nym configuration file, the destination is read from the same environment variable (Listing 3.4).

Expanding visibility. All structs and methods referenced by the copied `init(...)` must be accessible from `nym-jni`. This necessitated the widening of visibility modifiers of selected structs and methods from `pub(crate)` to `pub`. Wherever possible, the most restrictive visibility modifier is used.

Handling errors gracefully. As described in Section 3.3, I wrap the top-level `init(...)`, `run(...)` and `topLevelInit(...)` methods with my `call_fallible!()` macro, in order for any failure to be propagated to Android via JNI as a `RuntimeException`. Separately, calls to an `Err(`E`)` variant's `.expect()` method immediately crashes the Rust program. Instead, I want the `Err(`E`)`-typed value to propagate up to the top-level methods in order to gracefully raise a `RuntimeException`. This was achieved by replacing `.expect()` with `.with_context()?` from the `anyhow` community crate.

3.5.2 Implementing `topLevelInit(...)` and Porting `init(...)`

On Kotlin, the native methods `external fun topLevelInit(storageAbsPath: String)` and `external fun nymInit(...)` are defined in a file called `NymHandler.kt`, and are called on application startup. These invoke corresponding Rust code through JNI, as shown in Listing 3.4 with matching names mandated by the JNI specification. The Rust method names may appear to be plucked out of thin air, but are actually revealed in runtime `UnsatisfiedLinkError` exceptions when wrong names are used.

```
#[no_mangle]
pub extern "C" fn Java_com_p_NymHandlerKt_topLevelInit(/*...*/ storage_abs_path:
↪  JString)
{
    call_fallible!(topLevelInit, /*...*/, storage_abs_path);
}

fn topLevelInit(/*...*/ storage_abs_path: JString) -> Result<(), anyhow::Error> {
    let storage_abs_path: String = consume_kt_string(env, storage_abs_path)?;
    /*...*/
    std::env::set_var(
        "IMPL_NYMCONFIG_FOR_CONFIGANDROID_STORAGE_ABS_PATH",
        storage_abs_path
    );
}

#[no_mangle]
pub extern "C" fn Java_com_p_NymHandlerKt_nymInit(/*...*/) {
    call_fallible!(init, /*...*/);
    /* later when deciding where to save the Nym Configuration file Config.toml,
    ↪  calls:
       std::env::var("IMPL_NYMCONFIG_FOR_CONFIGANDROID_STORAGE_ABS_PATH") */
}

#[no_mangle]
pub extern "C" fn Java_com_p_NymHandlerKt_nymRun(/*...*/) {
    call_fallible!(run, /*...*/);
}
```

Listing 3.4: Top-level Rust implementations of the `topLevelInit(...)`, `init(...)` and `run(...)`, wrapped with `call_fallible!()` and callable from Kotlin.

3.5.3 Porting of `run(...)`

Because `clients_native_src` mirrored the original directory structure, much of the architectural groundwork has been heavy-lifted. I was able to easily scale up the amount of code I ported while maintaining repository structure and cleanliness. Similar to `init(...)`, the porting of `run(...)` was accomplished by copying over the top-level `run(...)` method from `nymtech/nym` and code referenced by it (as necessary), then making adjustments. The native method `external fun nymRun(...)` is also defined in `NymHandler.kt`.

3.6 Logging Timestamps

To characterise the timing characteristics of message streams sent by the prototype, timestamps must be logged. This section presents the timeline of a message, and the points at which timestamps are recorded.

Clock selection. I am only interested in time intervals between events. This makes monotonic clocks suitable, as they are not affected by Network Time Protocol stepping. Furthermore, the experiments in the evaluation chapter are designed such that messages are sent and received on the same device, making the monotonic timestamps comparable. From `man clock_gettime(3)`, `CLOCK_BOOTTIME` increments from boot and even when the system is suspended, which makes it ideal. This clock is available on both my PC and Android devices, which both run Linux. It is accessible on Rust via `nix::time::clock_gettime(nix::time::ClockId::CLOCK_BOOTTIME)` with the help of the `nix` community crate, and from Kotlin via `SystemClock.elapsedRealtimeNanos()`. Because the latter method's documentation does not specify that it gets timestamps from `CLOCK_BOOTTIME`, this fact was verified via inspection of Android's source code [32].

Timestamps collected. Each message generated during evaluation passes through the codebase as illustrated in Figure 3.3, and timestamps are collected as described in Table 3.3. Because the `nymtech/nym` codebase does not provide any facility for collecting timestamps, tagging timestamps to messages or a global key-value store, I had to manually insert the timestamps at appropriate parts of the `nym` codebase (§3.7).

`main` and `probe-effect-evaluation` branches. I maintain these two branches in the repositories (§3.1). The `main` branch logs timestamps t_1–t_8 as well as battery level information, while the `probe-effect-evaluation` branch only logs timestamps t_1 and t_8 in order to investigate the strength of the probe effect (§4.4).

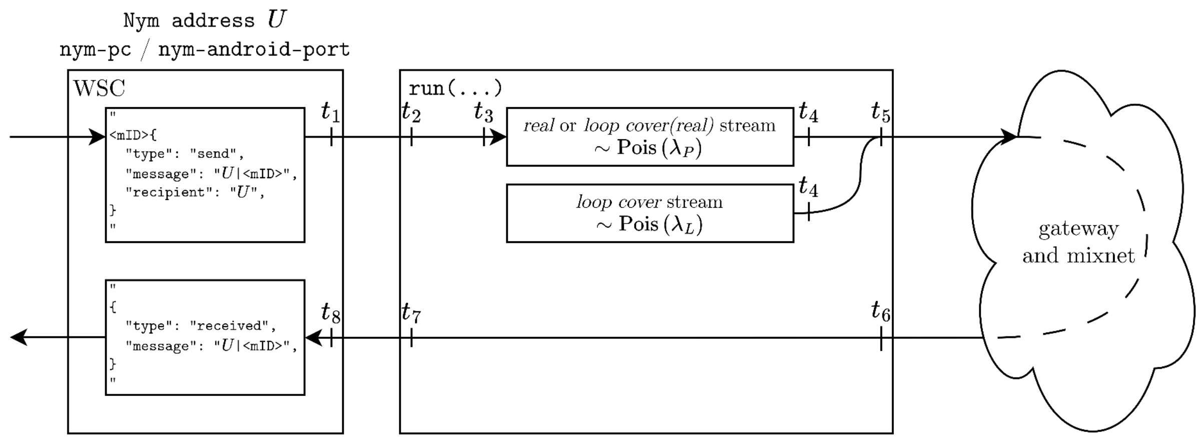

```
"
<mID>{
  "type": "send",
  "message": "U|<mID>",
  "recipient": "U",
}
"
```
```
"
{
  "type": "received",
  "message": "U|<mID>",
}
"
```
Nym address U
nym-pc / nym-android-port
WSC
run(...)
t_1
t_2
t_3
real or *loop cover(real)* stream $\sim \text{Pois}(\lambda_P)$
loop cover stream $\sim \text{Pois}(\lambda_L)$
t_4
t_4
t_5
t_6
t_7
t_8
gateway and mixnet

Figure 3.3: Overview of timestamps described in Table 3.3. This diagram fits into the larger architecture in Figure 3.4. The WSC is introduced in Section 3.8.

3.7 nym Crate (fork)

`nym-jni` initially depended directly on `nymtech/nym` (not a fork). However, logging timestamps requires extensive changes in Nym's code that are applicable to both Android and PC clients. Therefore, rather than continuing to copy files over to `nym_jni` (Android-specific), I created a fork named `nym`, and had `nym_jni` point to it instead of `nymtech/nym`.

Timestamp ID	Event	Collected in Repository
t_1	WSC Sending	`nym-pc` and `nym-android-port`
t_2	Received from WSC	`nym`
t_3	Enqueuing	
t_4	Dequeued	
t_5	Sending to Gateway	
t_6	Recevied from gateway	
t_7	Sending to WSC	
t_8	WSC Received	`nym-pc` and `nym-android-port`

Table 3.3: Overview of timestamps collected.

Finding the right places to place log statements was a huge undertaking due to the highly concurrent nature of the codebase (Section 2.3.1). I outline the general principles here.

Timestamps are collected from `CLOCK_BOOTTIME` as soon as the relevant event (Table 3.3) is triggered. Because printing them to `stdout` (PC) or `logcat` (Android) may incur significant overhead, this is delayed until after important processing is done. For instance, t_1 is collected right before the WSC (§3.8) sends the message to `nym` but printed after the message is sent; t_2 is collected right after WSC receives the message but printed after sending it off on a Tokio channel towards the Poisson mix queue. As further processing of the timestamp is done asynchronously, the impact on runtime behaviour is minimised.

Because the printing the timestamps may be delayed, some level of prop drilling was performed, to propagate timestamps through function calls until an appropriate time. This sometimes involves piggybacking timestamps onto existing structs via a new struct field. Where these are done, I prefixed my modifications with "`log_`". There was also a complication where the augmented code is reused for multiple purposes. For instance, the same code is used to send payload messages (of interest) and Nym acknowledgements (irrelevant to this project). In such cases, instead of passing `u64` timestamps around, `Option<u64>` is used. The `None` variant is drilled instead of `Some(_)` when logging is not necessary.

3.8 `nym-pc` Crate

This sub-repository contains an adaptation of example code provided in `nymtech/nym`. It executes a WebSocket Client (WSC), a long-running process that serves as a frontend to the `run(...)` program, saving messages to a local database (Figure 3.4). The WSC and `run(...)` processes are analogous to a chat application and background worker respectively, and continue to run as shell programs until they are terminated via `SIGINT` (control-C keystrokes).

3.9 `nym-android-port` Android Application

This section walks through the major components of the prototype, depicted in Figure 3.4. The prototype runs two processes, namely a UI process alive only when the user interacts with the UI, and a persistent background process executing the WSC and `run(...)`

programs in separate threads (§3.9.1). A *Room* database is used to store messages, contact information, and perform IPC (§3.9.2–3.9.3).

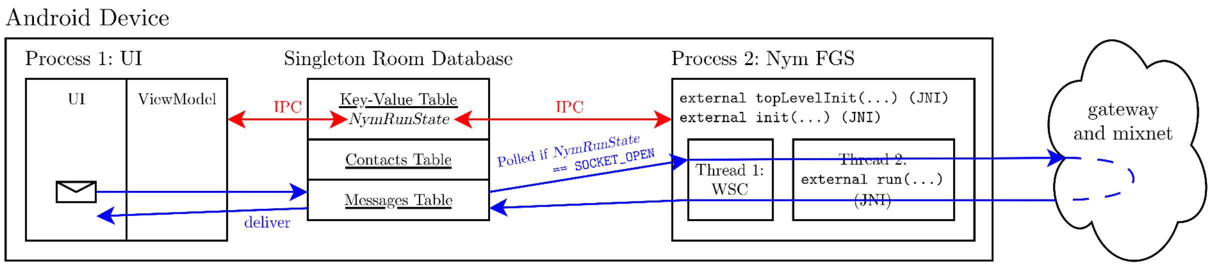


Figure 3.4: Data flows in the prototype.

3.9.1 Foreground Service (FGS)

The WSC and `run(...)` processes described in Section 3.8 must run continuously in the background, even when the device screen is off, in order for the device to continuously communicate with the gateway according to the mixnet design. However, Android applications have only one (UI) process by default, which is stopped when the user closes the application. A FGS serves as the second persistent process necessary for correct operation.

The prototype UI provides three screens: *ClientInfoScreen* for controlling the FGS, *ContactsScreen* for selecting a conversation target, and *ChatScreen* to send and receive messages. The user first starts the FGS from *ClientInfoScreen*. When the user sends a message from *ChatScreen*, it gets saved into the messages table of the *Room* database. The FGS references a `Flow` pointing from the messages table, which automatically emits the latest unsent message to the WSC thread, which propagates it to the `run(...)` thread. Messages received from the gateway are saved in the same table. *ChatScreen* automatically shows new messages on receipt, as it also references a `Flow` from the messages table, which emits the conversation's messages to the UI. The use of *Room* and `Flow`s greatly simplifies and automates UI synchronisation across the three screens.

Additionally, a wake lock is also held by the FGS. This provides immunity to most battery optimisations, in order for the WSC and `run(...)` threads to execute continuously without being terminated by the Android OS [33].

Previous iterations used Android's *WorkManager* library and *Bound Services* to handle background tasks. While that is the approach recommended by Google, there exists an undocumented 30 min execution limit that, once elapsed, triggers a cancellation and immediate restart of the background work. This led to misbehaviour of the `run(...)` thread, as explained later in Section 3.9.3. WorkManager was thus abandoned in favour of the stability of FGSs.

3.9.2 Database Tables

For debugging purposes, *ClientInfoScreen* provides a facility to switch between different Nym configurations (user accounts). There is thus a need to store the Nym address of the active account on the device. Android's *SharedPreferences* API was used in earlier iterations, but switched in favour of *Room* as it lacked the automatic UI synchronisation provided by *Room* and `Flows`. This, together with the state machine's state *NymRunState* (§3.9.3), are stored in a key-value table.

Each account ("owner") has another account as a "contact" if they have exchanged messages before. The contacts table maps owner addresses to contact addresses. The messages table contains four columns: the source Nym address, destination Nym address, message string and a boolean flag indicating whether the message was successfully sent.

Nym leaves the design of the string payload to end-applications such as my prototype. The sender's address is not delivered to the recipient by default, but this information is necessary for the prototype to function as a chat application. Therefore, all message payloads take the form "`<sender Nym address>|<message>`", as illustrated in Figure 3.3.

I have written unit tests to verify the correct operation of the *Room* database. Finally, it uses the singleton design pattern to ensure each process only instantiates one expensive connection to the database.

3.9.3 Database for IPC and State Machine

The `run(...)` thread misbehaves if its lifecycle is mismanaged (e.g. termination during the setup phase results in an immediate crash rather than graceful exit). To prevent it from crashing the prototype, I implemented a state machine that controls the FGS's lifecycle, as in Figure 3.5.

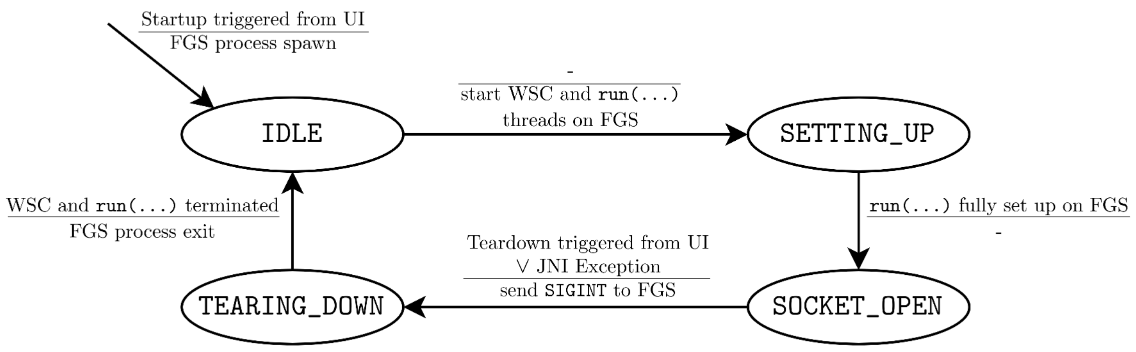

Startup triggered from UI
FGS process spawn
IDLE
-
start WSC and run(...) threads on FGS
SETTING_UP
run(...) fully set up on FGS
-
SOCKET_OPEN
Teardown triggered from UI
∨ JNI Exception
send SIGINT to FGS
TEARING_DOWN
WSC and run(...) terminated
FGS process exit

Figure 3.5: State transition diagram of the prototype's state machine.

The current state is saved as *NymRunState* in the key-value table of the database (Figure 3.4). From the user's point of view, pressing "start" on the *ClientInfoScreen* triggers a cascade of state changes from `IDLE` to `SOCKET_OPEN`, and pressing "end" triggers transitions back to `IDLE`. Importantly, the FGS process is up in all states except `IDLE`, and the UI buttons are locked during the `SETTING_UP` and `TEARING_DOWN` phases to enforce proper lifecycle management of the FGS from the UI process. Similarly to the PC client (§3.8), the dismantling of the FGS is triggered by a `SIGINT` signal sent using `Process.sendSignal(pid, 2)` in Kotlin.

IPC could be done via other avenues such as *Bound Services* and *Messengers* [34], but using *Room* is not only simpler, but also provides automatic UI synchronisation (blocking buttons, displaying state for debugging) as discussed previously.

3.10 Semi-Automatic Testing Framework (Extension)

The compilation toolchain described in Section 3.2 involved many moving parts. As bugs occurred frequently during development, automating part of the compilation and data collection pipeline promised to save a lot of time.

All experiments in the evaluation chapter use this pipeline. After startup, the FGS generates one message every second containing an upwards counting message ID. These messages are sent across the mixnet back to itself. After receiving N such messages, the FGS sends a `SIGINT` signal to itself, concluding the experiment.

I have written shell scripts to aid with the data collection. They take arguments via the command-line interface, specifying the target architecture and experimental parameters (Table 4.2). I designed the scripts such that upon running a single command, the compilation, application launch and data collection phases occur automatically in cascade, as outlined below. After conclusion of a single experiment, the scripts restart the entire loop with the next combination of parameters.

Compilation. First, the script performs `git checkout` operations to ensure all repositories are in the correct branch (`main` or `probe-effect-evaluation`), then writes N into a Kotlin file within `nym-android-port`, and the parameters λ_P, λ_L into a file within `nym`. These values are referenced by the codebase at runtime. Compilation is performed, including copying of the `libnym_jni.so` into the correct location. ADB commands are used to install the application bundle onto the Android device, grant Android OS permissions required for foreground services, switch to the correct data connectivity mode, set the application's battery optimisation mode, and configure Power Save Mode according to experiment parameters.

Application Launch. The prototype is started via ADB. Two files `evaluationRunning.txt` and `evaluationMessagesReceived.txt`, each always containing a single integer, are created in Android's filesystem for synchronisation between PC and Android (2–3 in Figure 3.6). In order to kick-start the FGS without interacting with the UI, another short-lived kick-starter FGS is launched in order to emulate a user interacting with the screen: change the *NymRunState* in the database from `IDLE` to `SETTING_UP` and launch the main FGS (5). No UI process is created; everything runs in the background.

Data Collection. A new log file is initialised on the PC with filename and lines describing the experiment's parameters (1). The `logcat` shell program is launched on Android using ADB to capture `logcat` messages (4). As the experiment proceeds, the prototype increments the value in `evaluationMessagesReceived.txt` every 100 messages received (6), and this value is printed on the PC shell for me to check that the prototype is not stuck (9). Once the prototype receives N messages (7), it terminates itself and writes `0` to `evaluationRunning.txt` (8), which is checked every minute by the PC shell (10). Polling for values in these two files from the PC shell is a potentially expensive operation, since it involves filesystem operations on Android. As checking for termination is more important than getting an up-to-date count of received messages, the frequencies 1 minute and 6 minutes are chosen respectively to offer a balance between real-time fidelity and overhead (9–10). Finally, the `logcat` process started in (4) is terminated (11), and contents of `logcat.txt` copied from Android and appended onto the `experiment_(...).txt` file created at the beginning (12).

The implementation of the extension task allowed me to run experiments one after another with minimal supervision (e.g. overnight), and yielded significant time savings when exploring the experiments' parameter space.

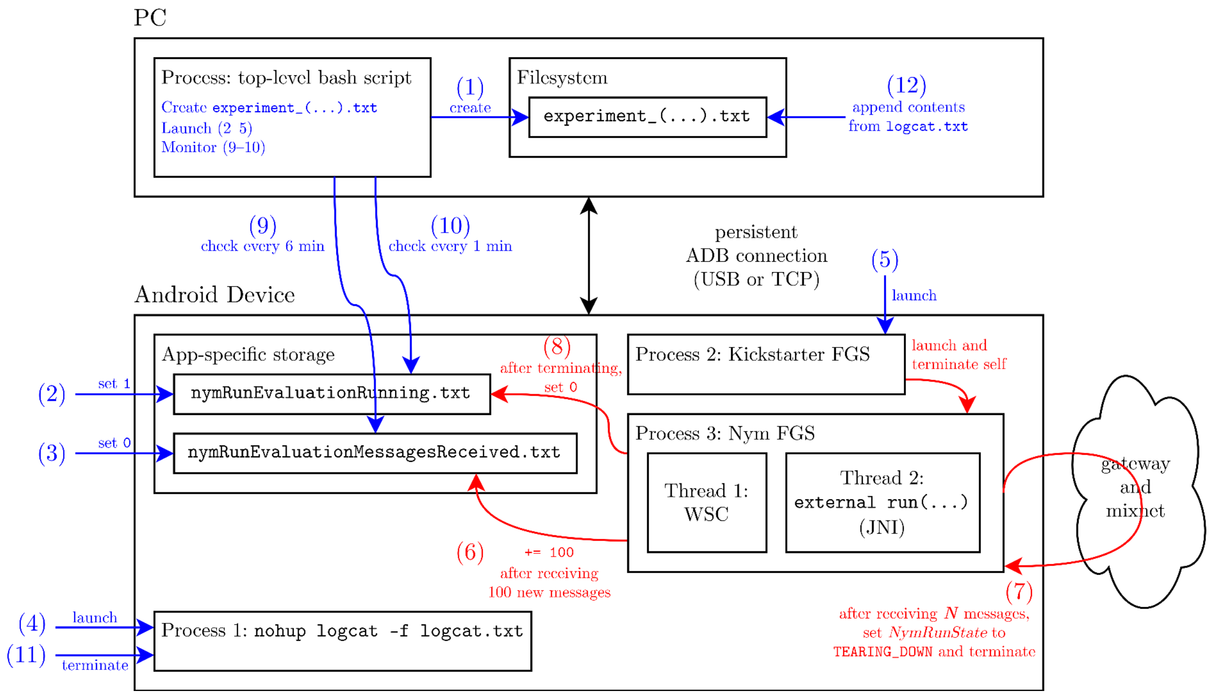


Figure 3.6: Overview of automatic data collection process. The blue and red text indicate steps initiated from the PC and Android devices respectively. Details of the persistent ADB connection are explained in Section 4.7.1.

Chapter 4

Evaluation

In this chapter, I present qualitative and quantitative analyses to verify the extent that the hard and soft requirements (§2.4) have been met.

4.1 Methodology

Table 4.1 lists the devices used for evaluation, and I will use the short name in the text. The PC Client (§3.8) is run on PC (1.80 GHz quad-core Intel Core i7-10510U), and the Android Client (§3.9) is run on two devices. Pixel (1.80 GHz octa-core Qualcomm Snapdragon 765G) is used to gather qualitative data over the parameter space, while Moto (2.0 GHz octa-core Cortex-A53) is primarily used to gather quantitative data about battery power consumption. Importantly, all unused applications and services are disabled on Moto, as far as the OS allows. This minimises external events not related to the prototype.

Operating System	**Name**	**Short Name**	**Architecture**	**RAM**/GB	**Battery Capacity**/Wh
Linux (Arch)	Thinkpad T490	PC	*x86-64*	16	N/A
Android 13	Pixel 4A 5G	Pixel	*ARM64*	6	14.7
Android 9	Motorola Moto E6 Plus	Moto	*ARMv7a*	2	11.7

Table 4.1: Overview of devices used for experiments.

I explored numerous parameters. For Pixel and Moto, compilation outputs either an unoptimised build (“debug”) or an optimised one (“release”) (§4.3). All t_1–t_8 timestamps could be logged (`main` branch), or only t_1 and t_8 (`probe-effect-evaluation` branch) (§4.4). The prototype’s battery optimisation mode could be “unoptimised” or “optimised” (§4.5). The device’s *Power Save Mode (PSM)* may be “on” or “off” (§4.5). The phone can be connected to the Internet via cellular “data” or “Wi-Fi”. These are summarised in Table 4.2. For PC, all the above apply except battery optimisation mode and PSM. Connectivity is also fixed to “Wi-Fi”.

The Nym parameters (λ_P, λ_L) are initially fixed at the default values ($50\,\mathrm{s}^{-1}, 5\,\mathrm{s}^{-1}$), and I vary these from Section 4.7 onwards. μ only affects the time taken for a message to traverse the mixnet, and is thus fixed at the default $50\,\mathrm{s}^{-1}$. The number of messages N sent in an experiment is fixed at 3600, and together with a message generation rate of $1\,\mathrm{s}^{-1}$ (§3.10), experiments are expected to last 1 h, in order to capture long-running behaviour of the prototype. The messages take the form “`<sender Nym address>|<mID>`”, where `mID` is a counter (Figure 3.4). Each message fits into a single Sphinx packet.

Two kinds of measurements are taken: `logcat` timestamps (Table 3.3) and power consumption logs. Each permutation of parameters defines an experiment and an associated

Parameter	Possible Values
Build	{ debug, release }
Probe Effect evaluation	{ false, true }
Battery optimisation mode	{ unoptimised, optimised }
Power Save Mode (PSM)	{ off, on }
Connectivity	{ data, Wi-Fi }

Parameter	Values Explored
(λ_P, λ_L)	$\left\{ \begin{array}{c} (50, 5), \\ (5, 0.5), \\ (0.5, 0.05), \\ (0.05, 0.005) \end{array} \right\}$
μ	{ 50 }
N	{ 3600 }

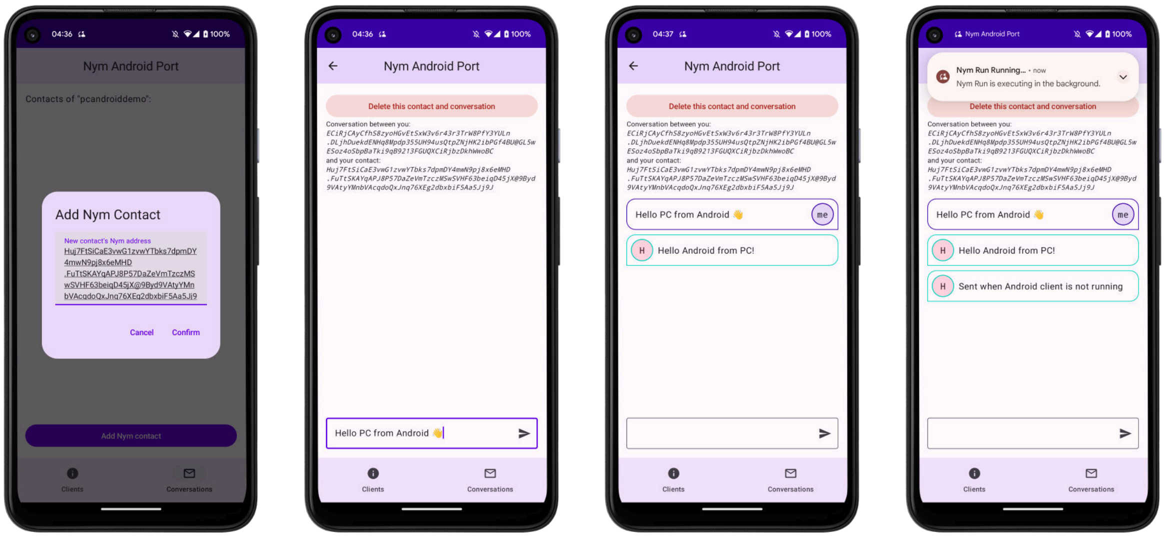
Table 4.2: Overview of experimental parameters explored. $\lambda_P, \lambda_L, \mu$ are defined in Table 2.1.

"dataset". Each dataset is represented by a single box plot in the graphs that follow. Timestamp datasets, augmented with with some battery statistics, are collected as in Section 3.10 and discussed in the coming sections. Power datasets are discussed from Section 4.7 onwards. Because the parameter space is huge, I first present the results of exploratory work that reveal the redundancy of some parameters choices.

4.2 Ability to Communicate with a PC Client

Before analysing the data, I demonstrate that the prototype performs to the requirement RH1. On PC, `nym`'s `run(...)` command-line process is launched. `nym-pc`'s WSC command-line process is then started, which first obtains its own Nym address, then waits to communicate with the Android client. On Pixel, the prototype is launched. The PC client's Nym address is pasted in, and messages are sent between Pixel (Figure 4.1) and PC (Figure 4.2).

Both the PC and Android clients are able to sustain the connection for over 1 h, until the processes were manually terminated. The fourth screenshot of Figure 4.1 demonstrates the offline storage capability of Nym: a message was sent from PC when Pixel's state machine was in the `IDLE` state, and later received when reactivated to the `SOCKET_OPEN` state (exemplified by the notification). RH1 is satisfied.


Contacts of "pcandroiddemo":

Add Nym Contact

New contact's Nym address
Huj7FtSiCaE3vwG1zvwYTbks7dpmDY4mwN9pj8x6eMHD.FuTtSKAYqAPJ8P57DaZeVmTzczMSwSVHF63beiqD45jX@9Byd9VAtyYMnbVAcqdoQxJnq76XEg2dbxbiF5Aa5Jj9

Delete this contact and conversation

Conversation between you:
ECiRjCAyCfhS8zyoHGvEtSxW3v6r43r3TrW8PfY3YULn.DLjhDuekdENHq8Mpdp355UH94usQtpZNjHK2ibPGf4BU@GL5wESoz4oSbpBaTki9qB9213FGUQXCiRjbzDkhWwoBC
and your contact:
Huj7FtSiCaE3vwG1zvwYTbks7dpmDY4mwN9pj8x6eMHD.FuTtSKAYqAPJ8P57DaZeVmTzczMSwSVHF63beiqD45jX@9Byd9VAtyYMnbVAcqdoQxJnq76XEg2dbxbiF5Aa5Jj9J

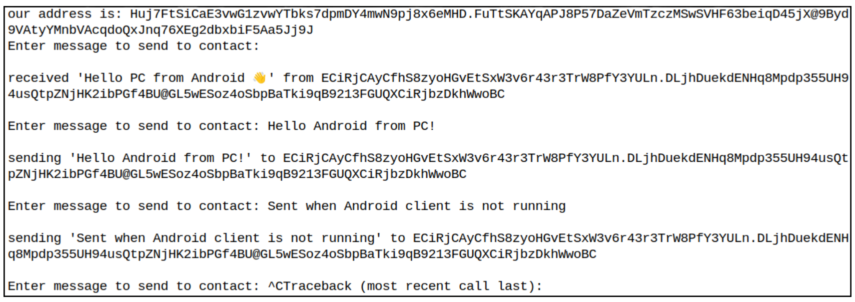
Hello PC from Android 👋 — me

H — Hello Android from PC!

H — Sent when Android client is not running

Nym Run Running... • now
Nym Run is executing in the background.

Figure 4.1: Screenshots from Pixel as it communicates with PC.



```
our address is: Huj7FtSiCaE3vwG1zvwYTbks7dpmDY4mwN9pj8x6eMHD.FuTtSKAYqAPJ8P57DaZeVmTzczMSwSVHF63beiqD45jX@9Byd
9VAtyYMnbVAcqdoQxJnq76XEg2dbxbiF5Aa5Jj9J
Enter message to send to contact:

received 'Hello PC from Android 👋' from ECiRjCAyCfhS8zyoHGvEtSxW3v6r43r3TrW8PfY3YULn.DLjhDuekdENHq8Mpdp355UH9
4usQtpZNjHK2ibPGf4BU@GL5wESoz4oSbpBaTki9qB9213FGUQXCiRjbzDkhWwoBC

Enter message to send to contact: Hello Android from PC!

sending 'Hello Android from PC!' to ECiRjCAyCfhS8zyoHGvEtSxW3v6r43r3TrW8PfY3YULn.DLjhDuekdENHq8Mpdp355UH94usQt
pZNjHK2ibPGf4BU@GL5wESoz4oSbpBaTki9qB9213FGUQXCiRjbzDkhWwoBC

Enter message to send to contact: Sent when Android client is not running

sending 'Sent when Android client is not running' to ECiRjCAyCfhS8zyoHGvEtSxW3v6r43r3TrW8PfY3YULn.DLjhDuekdENH
q8Mpdp355UH94usQtpZNjHK2ibPGf4BU@GL5wESoz4oSbpBaTki9qB9213FGUQXCiRjbzDkhWwoBC

Enter message to send to contact: ^CTraceback (most recent call last):
```

Figure 4.2: Screenshot from PC's WSC process as it communicates with Pixel.

4.3 Debug versus Release Builds

During development, I built debug artefacts as they compile faster. These are the default compiler settings (for both Gradle and Cargo). However, since end users run release artefacts, those are also built during the evaluation phase of the project. For Gradle, compilation of a release build requires cryptographic signing using a keystore. I created a temporary keystore on my PC for this project and modified `build.gradle` (Figure 3.2) to reference the keystore secrets indirectly (such that secrets are not uploaded onto Git in plaintext) [35]. For Cargo, the `--release` flag was used during compilation.

For this section, we look at the Pixel and PC datasets only. (λ_P, λ_L) are fixed at $(50, 5)$. Figure 4.3(a) compares the per-message overhead before the mixnet, which is the time elapsed before sending to the mixnet $(t_5 - t_1)$ excluding the time spent in the queue $(t_4 - t_3)$. For release builds, median overhead is around 15 ms on Pixel, and around 5 ms on PC. However, in debug builds, the Pixel median overhead per message is around 75 ms, which is around 5 times that of the release builds. For PC, the factor is around 4, and the overhead sits at around 20 ms. In ideal mixnet clients, this overhead is zero. The larger additional probabilistic overhead in the debug builds cause the inter-packet timings of their real message streams to deviate further from the ideal exponential distribution $\text{Exp}(\lambda_P)$, shown in green in Figure 4.3(e).

Figures 4.3(d) and 4.3(f) shows comparisons of the queue duration of *real* messages ($\sim \text{Exp}(\lambda_P = 50)$), and inter-packet timings of the *loop cover* stream ($\sim \text{Exp}(\lambda_L = 5)$) respectively. In both figures, the distributions for debug builds deviate further from the ideal distributions than the release builds. The runtime performance of the Nym client queues are therefore closer to ideal for release builds.

Finally, end users eventually only install release APKs. For all these reasons, I will consider only release builds from now on.

4.4 Probe Effect

The presence of logging code fundamentally alters the runtime behaviour of the prototype. There is a non-zero overhead incurred when retrieving timestamps from the OS and writing them to a log file, particularly so for the `main` branch (§3.6) where all t_1–t_8 are logged.

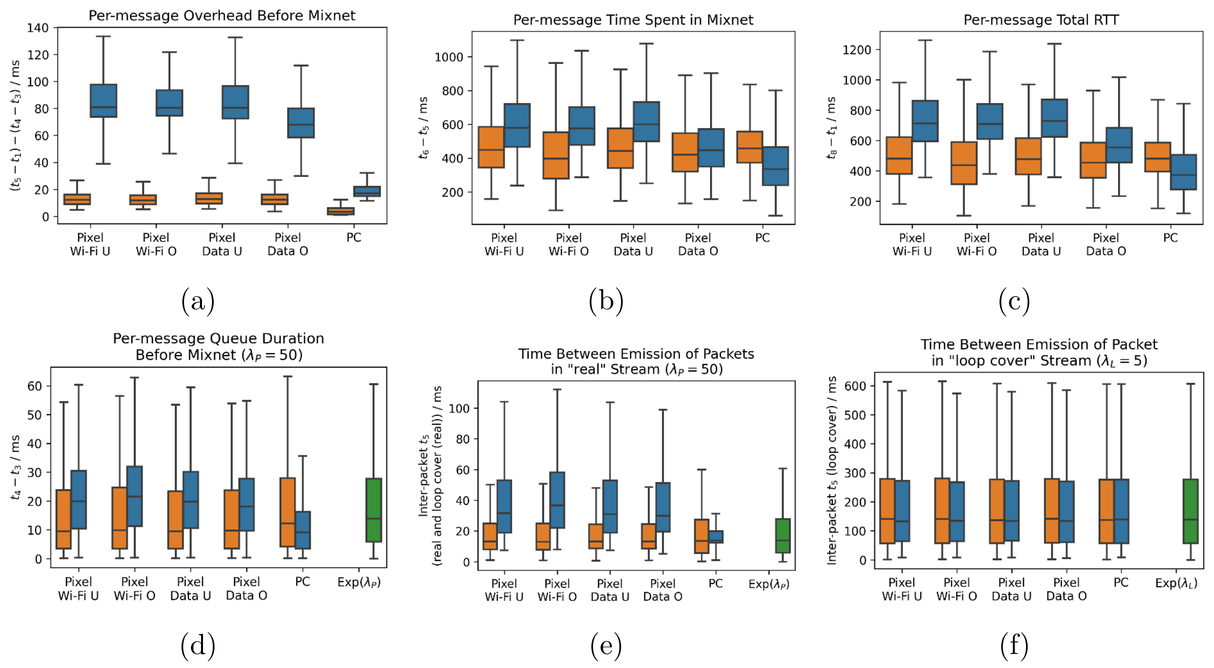


Figure 4.3: Comparison between debug (blue) and release (orange) builds. PSM is off. Ideal distributions are in green, where applicable. `U` refers to unrestricted and `O` refers to optimised (see Table 4.2).

Figure 4.4 compares the per-message round-trip time (RTT) between datasets collected on the two repository branches. Comparing each green box plot to the adjacent red ones, there is no consistent or significant difference, and all RTTs have a median of around 450 ms. The time spent by each message on the mixnet (part of the RTT) is assumed to be constant in this analysis, which may not be realistic.

However, the overhead introduced by logging statements is likely in the order of milliseconds, which is at least 2 orders of magnitude smaller than the RTT, which is dominated by the time spent in the mixnet, exemplified by Figures 4.3(b)–4.3(c). There is therefore no significant probe effect; data collection does not significantly alter the runtime behaviour of the prototype.

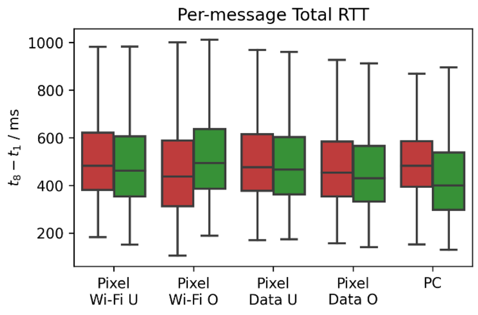


Figure 4.4: Comparison between `main` (red) and `probe-effect-evaluation` (green) branches. Dataset names are as in Figure 4.3.

4.5 Effect of Android OS Power Management

Battery Optimisation Mode. In Android, every application is assigned one of three battery optimisation modes $\in$ {restricted, optimised, unrestricted}. Restricted mode completely blocks background processing, and is thus not considered.

PSM. PSM is an OS-wide toggle that can be activated by the user to conserve battery, primarily by limiting background processing [36].

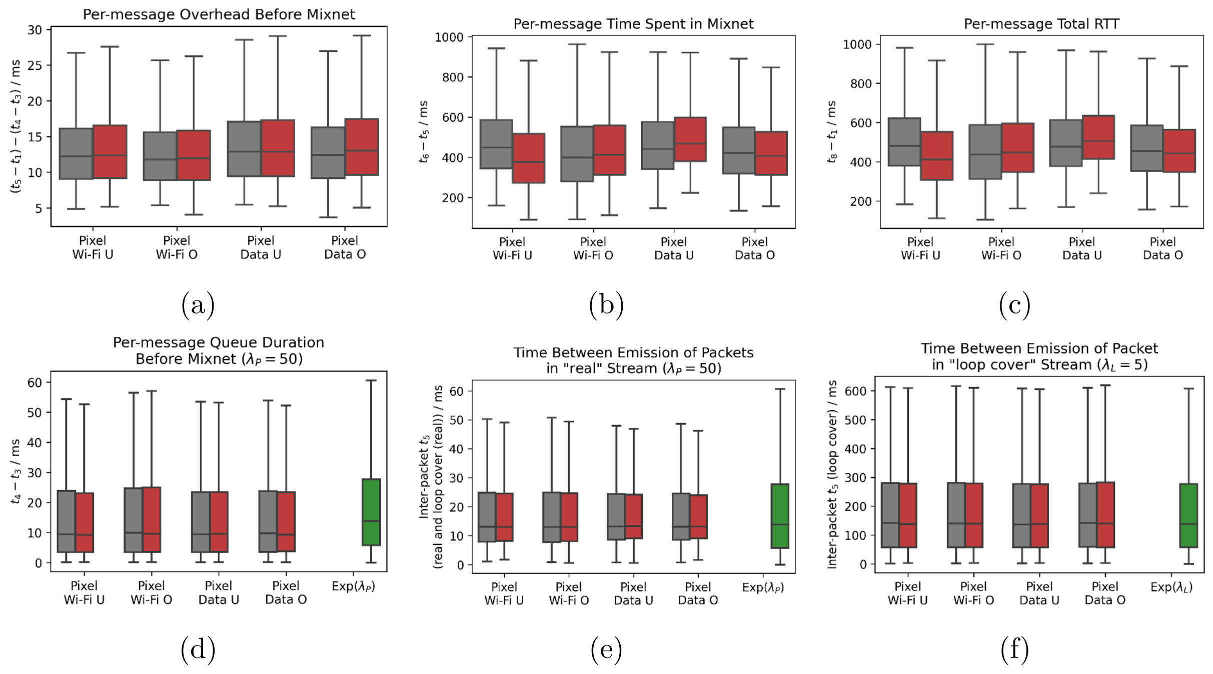


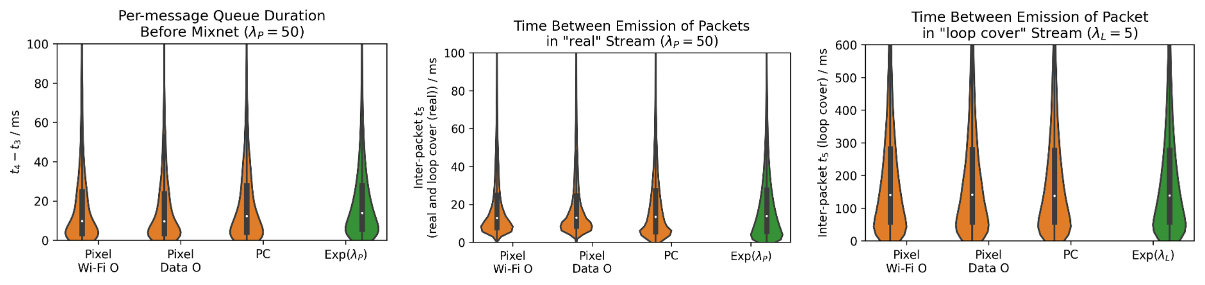
Figure 4.5: Comparison across battery optimisation mode and PSM modes (grey for "off", red for "on"). Dataset labels are as in Figure 4.3.

As exhibited in Figure 4.5, the choice of battery optimisation mode and the usage of PSM both made no significant difference in performance. The median per-message overhead before mixnet is around 13 ms across the board, and all configurations adhere closely to the ideal distributions. Figures 4.5(b)–4.5(c) are displayed for completeness.

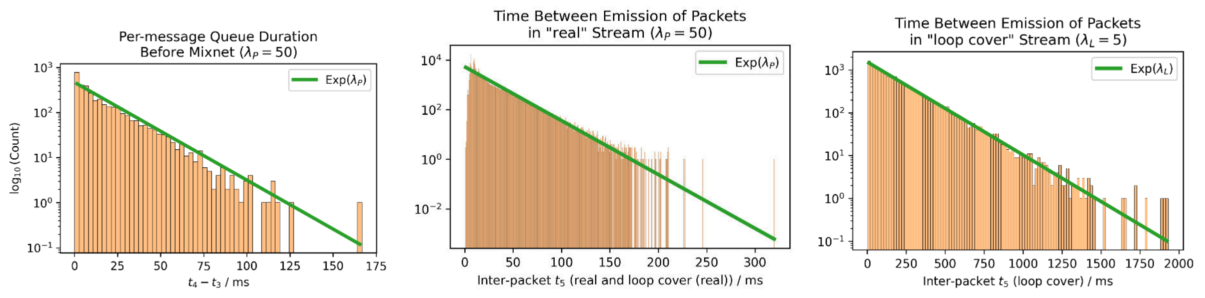
Foreground services are the same mechanism used by music player applications to continuously playback audio when the device screen is off. Together with wake locks, the prototype is immune to most battery optimisation measures such as *Doze* and *App Standby* [37, 38, 39], which can explain why no differences are observed between the datasets in Figure 4.5. From now on, I'll only consider optimised mode, since that is the default mode that applications are installed with. PSM is assumed off from now on.

4.6 Differences with PC and Ideal Distributions

The behaviour of *real* and *loop cover* streams are similar in both Pixel and PC, as seen exemplified by the similar shape of the distribution of the datasets in Figure 4.6(a). Figure 4.6(b) plots the $\log_{10}$(count) of the values in the "Pixel Data O" dataset, and they indeed follow the ideal distribution very closely. The same result holds for the "Pixel



(a) The violin plot (kernel density estimation) of select distributions from Figures 4.3(d)–4.3(f).



(b) Detailed distributions of the "Pixel Data O" dataset. The y-axis is scaled logarithmically.

Figure 4.6: Android and PC clients both adhere closely to the ideal distributions.

Wi-Fi O" and PC datasets. Therefore, I conclude that there are no obvious deviations from the observable message timings. Thus, RH2 is satisfied.

4.7 Energy Consumption (Extension)

Having met the hard requirements, I now evaluate the Android port against its soft requirements, beginning with battery efficiency, which is the core premise of this work. From this point on, the only parameters left to explore are connectivity (data or Wi-Fi) and the four values of (λ_P, λ_L), which I now express as a delay factor $d \in \{1, 10, 100, 1000\}$ for the respective values in Table 4.2. The repository branches are fixed at `probe-effect-evaluation`, since we no longer perform any timing analyses.

4.7.1 Motivation: The Need for Quantitative Results

In early-phase experiments, I logged the Android devices' battery levels at regular intervals, then compared across experimental parameters the total percentage drop after one hour. In general, the prototype drains less battery when running on Wi-Fi compared to mobile data. Increasing the delay factor also reduces battery usage.

However, to allow controlling *batches* of experiments using the semi-automatic pipeline in Section 3.10, these experiments were conducted while maintaining an persistent ADB connection to my PC (Figure 3.6). For Pixel, this was done over USB as there is an option to maintain a USB connection without receiving power from the PC, allowing me to observe how the prototype consumes battery. For Moto, this feature is not available, thus the ADB connection is persisted over TCP, resulting in significant power overhead. The battery drain statistics are hence an overestimate, particularly for Moto.

Furthermore, the battery percentage values reported by the Android OS are coarse (integer-valued) and innately inaccurate. More precise measurements of the prototype's energy usage may lead to interesting insights. I thus used an energy measurement kit from my supervisor, described next.

4.7.2 Experimental Setup

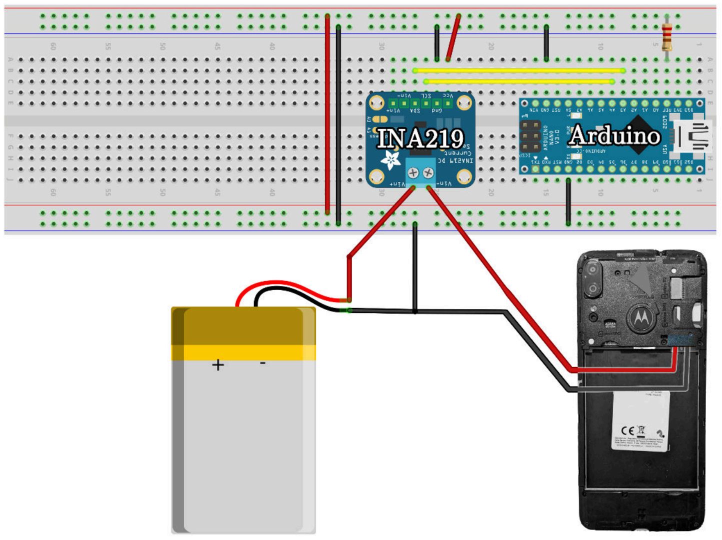


Figure 4.7: Schematic of the hardware used to collect power consumption measurements from a phone battery. (Credits: Adapted from the documentation of the energy kit)

I removed the battery from the Moto. The negative terminal is re-connected to the phone, but the positive terminal is connected first to an INA219 power monitor chip on the breadboard, before re-connecting to the phone, as shown in Figure 4.7. As the phone operates, current flows from the battery to the phone through the INA219, which continuously outputs current measurements to the Arduino [40]. The Arduino transfers the measurements via USB to my PC, where it is captured by a Rust-based logger program written by my supervisor.

Each experiment lasts an hour, and results in a dataset of M readings each of the form (t_i, P_i) where t_i is the time elapsed since the start of the logger program, and P_i is the mean power consumed from the detached battery since the previous reading. The total energy consumed can be calculated using the Riemann sum $\sum_{i=1}^{M-1} P_i \cdot (t_i - t_{i-1})$.

4.7.3 Results and Discussion

Table 4.3 summarises the results, including the baselines where the prototype is not running. There is some variability in the data usage across runs, particularly when $d = 1$, due to the probabilistic nature of the message streams. From Figure 4.8, we can see that the total energy consumed decreases approximately linearly with the logarithm of the delay factor.

The default $d = 1$ results in huge battery usage. On data, after just an hour of continuous operation, 995.06 mWh (6.8% and 8.5% of the Pixel and Moto battery capacities respectively) was drained. Extrapolating this power consumption rate to Pixel, the Pixel's

Connectivity	d	**Power Range** / mW	**Mean Power** / mW **Total Energy** / mWh	**Est. Life (Moto)** / h	**Est. Life (Pixel)** / h	**Data** / MB
Wi-Fi	1	$[8, 2623]$	388.42	30.1	37.8	496.64
	10	$[7, 2461]$	266.56	43.9	55.1	94.64
	100	$[3, 2210]$	104.73	111.7	140.4	11.83
	1000	$[7, 1696]$	60.11	194.7	244.6	3.82
	(idle)	$[7, 1964]$	41.76	280.2	352.0	N/A
Data	1	$[10, 3611]$	995.06	11.8	14.8	689.46
	10	$[10, 3491]$	643.48	18.2	22.8	93.62
	100	$[1, 3311]$	304.92	38.4	48.2	12.20
	1000	$[2, 3357]$	168.62	69.4	87.2	3.60
	(idle)	$[5, 3434]$	27.91	419.2	526.7	N/A
Airplane Mode	(idle)	$[5, 1688]$	11.96	978.1	1228.8	N/A

Table 4.3: Quantitative comparison of battery and data consumption rates of the prototype in an hour.

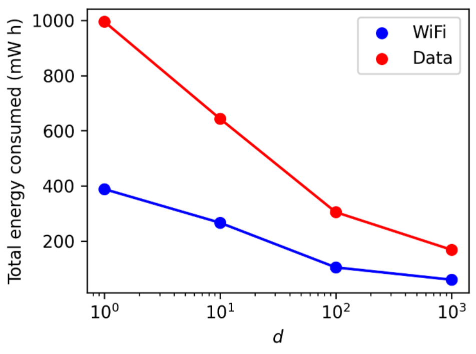


Figure 4.8: Plot of total energy consumed against delay factor d.

battery lasts for about 15 h. With typical mobile usage patterns outside of experimental conditions, the battery life is barely half a day. This does not satisfy RS1. The data usage cannot be ignored too: 690 MB per hour translates to over 16 GB in just a day. Unless end users have unlimited data, this alone would be a huge disincentive from using the prototype.

For instant messaging applications, if the user can tolerate a bit of latency, higher delay factors can be considered. When $d = 100$, the *real* stream outputs a message every 2 s on average, and improves the estimated battery life on Pixel to 48.2 h, which now satisfies RS1. The data usage rate is also more reasonable, at about 300 MB per day.

4.8 CPU and Memory Usage

Observations of the CPU and memory usage of the prototype were performed on Moto, using Android Studio's Profiler. Regardless of the parameters used (connectivity and d), the memory usage hovers in the range 39–45 MB, oscillating due to the device's garbage collector. For CPU usage, there is no difference between Wi-Fi and data, but d makes a difference. The CPU utilisation by the prototype hovers in the range 10–25% when $d = 1$, and generally decreases to 0–5% when $d = 1000$, at which point the CPU is mostly idle except a heartbeat every second, corresponding to the creation of a message and saving it to *Room*. RS2 is thus satisfied, even at $d = 1$.

4.9 Application Size

Table 4.4 shows the space used by the prototype, excluding application data. The prototype uses less than 50 MB on both Pixel and Moto, which is less than the developed applications, as expected of a prototype. When Application Binary Interface (ABI) splitting is done, the APK only contains the `libnym_jni.so` compiled for the device's target architecture, otherwise all versions of the shared binary is included. If the shared library increases in size, or if more target ABIs are considered for implementation, then ABI splitting would be a useful technique. RS3 is satisfied, especially since the average storage space on mobile devices is above 100 GB [41].

Application	**Space Used on Pixel (*ARM64*) / MB**		**Space Used on Moto (*ARMv7a*) / MB**	
	without ABI split	**with ABI split**	**without ABI split**	**with ABI split**
Prototype	48.05	39.43	41.25	31.62
Tor Browser	274	267	216	213
Whatsapp	118	-	85.27	-

Table 4.4: Space taken up on Moto by the prototype, compared to popular anonymity network and chat clients. The space appears to be device-dependent likely due to the different architectures involved.

4.10 Generalisability of Results

During data collection, some of the 3600 messages have missing timestamps, possibly due to the OSes skipping log statements when busy. However, this does not affect the validity of the discussed results, as I have verified that all datasets considered are 3500 ± 100 in size.

My implementation's minimum Android API level is set to `21`, which is the current oldest possible setting. According to Android Studio, the cumulative distribution of Android users using API `21` or newer is 99.3% as of April 2023. This allows my prototype to be used by almost all Android users today, fulfilling RS4.

The Pixel and Moto devices were chosen because these vendors make minimal changes to the stock Android OS. According to "Don't kill my app!" (DKMA), some vendors are known to impose draconian limitations on background processes without workarounds for developers and users [42], which would complicate implementation and evaluation. The results discussed in this chapter thus should generalise to other devices not "blacklisted" by DKMA.

Chapter 5

Conclusions

All success criteria set out in the initial proposal were met. I successfully demonstrated a working implementation of an Android mixnet client, and presented an evaluation of a wide range of experimental parameters, which were determined as the project progressed, as planned. I further exceeded the base requirements by completing two out of the three proposed extension tasks of constructing an automated testing pipeline and performing power measurements using custom hardware from my supervisor. Admittedly, the compilation and testing pipeline I constructed is not *fully* automatic from compilation to experiment to data plots as initially envisioned, but accomplishing that would detract from the main focus of the project.

I successfully debugged and assembled a complicated compilation toolchain, and put to use a collection of open source libraries and frameworks for a complete implementation. The highly concurrent multi-threaded Nym codebase was understood in order to collect timestamp code at specific points during the codebase's execution. As a side product, the `jvm_kotlin_typing` library was implemented to ease the Rust programmer's job when interacting across JNI with a Kotlin application. It supports all primitive data types in Kotlin and Rust, except `Char`, and is thus almost ready for publication as a community crate.

During the baseline implementation, some of the problems faced were open issues still undergoing discussion in the open source community. A central theme throughout this work has been the general unavailability of documentation, and the need for time-intensive searching and exploratory work, which dominated the first half of this project's timeline. I hope that the presentation of the difficulties, chosen solutions and alternative strategies in this dissertation serves as a helpful resource for maintainers of similar projects.

5.1 Reflections

This project started off rather ambitious, but I quickly learnt the harsh realities of experimental work. I gained valuable experience in upholding consistency of good software engineering techniques, such as code documentation and recording motivations behind key decisions for the future me, which is especially applicable for long drawn projects. My key takeaways from discussions with my supervisor are that of making incremental progress in the face of monolithic undertakings, and that exploration, much as it is tedious and sometimes undirected, is necessary for informing later exploitation.

Exploration of the explosion in parameter combinations became dauntingly unwieldy. I now appreciate the importance of proper forward planning and bookkeeping (such as maintaining a experiment table and naming log files descriptively) to keep afloat of confusion. With the benefit of hindsight, I would have taken a more systematic and organised approach to cope with complexity. In particular, I would spend less time trying to collect multiple datasets for all possible parameter combinations, but instead start off with the aim of quickly reducing the parameter space, as I have structured the evaluation chapter.

The people working on Nym have expressed interest to see my work. While the project was challenging, it also gave me a great sense of achievement and purpose, knowing that my work could be useful to the mixnet community in the near future.

5.2 Future Work

Here are some proposed continuations of this project that were not feasible given the project timeline:

- Further modularisation of the `nym-android-port` repository to fully decouple Nym logic from the chat application, allowing Nym logic to be reused easily in multiple projects.
- Automatic restarting of the FGS after an error occurs. One common problem is with intermittent or unreliable connection: the state machine can still misbehave when this happens.
- Long-term survival of the FGS. Increasing the delay seems to increase the probability of an unexpected error, forcing the shutdown of the FGS. It is not currently clear whether this is due to the Android OS terminating the prototype, the Nym gateway resetting the connection, or other reasons. The prototype has failed during local testing after $(9\,\mathrm{h}, 5.5\,\mathrm{h}, 3\,\mathrm{h})$ for $d = (1, 100, 1000)$ respectively.
- The third extension task: implementation of multicast over mixnets [43].

Bibliography

[1] Cambridge Dictionary. Privacy. https://dictionary.cambridge.org/dictionary/english/privacy, 2023. Last accessed 10 May 2023.

[2] United Nations General Assembly. Universal Declaration of Human Rights. https://www.ohchr.org/sites/default/files/UDHR/Documents/UDHR_Translations/eng.pdf, 1948. Last accessed 10 May 2023.

[3] United Nations Conference on Trade and Development. Data protection and privacy legislation worldwide. https://unctad.org/page/data-protection-and-privacy-legislation-worldwide, 1948. Last accessed 10 May 2023.

[4] Ewen Macaskill and Gabriel Dance. NSA files decoded: Edward Snowden's surveillance revalations explained. https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded#section/1, 2013. Last accessed 6 April 2023.

[5] Phillip Rogaway. The moral character of cryptographic work. *Cryptology ePrint Archive*, 2015.

[6] A Pfitzmann and Marit Hansen. A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management. *URL: http://dud. inf. tu-dresden. de/literatur/Anon_Terminology_v0*, 34, 01 2010.

[7] Simon Kemp. Digital 2023: Global overview report. https://datareportal.com/reports/digital-2023-global-overview-report, 2023. Last accessed 10 May 2023.

[8] P.F. Syverson, D.M. Goldschlag, and M.G. Reed. Anonymous connections and onion routing. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, pages 44–54, 1997.

[9] S.J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 183–195, 2005.

[10] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, feb 1981.

[11] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1199–1216, Vancouver, BC, August 2017. USENIX Association.

[12] Nym. Nym roadmap. https://nymtech.net/#roadmap, 2022. Last accessed 10 May 2023.

[13] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, USA, 2018.

[14] Roger Dingledine, Vitaly Shmatikov, and Paul Syverson. Synchronous batching: From cascades to free routes. volume 3424, 09 2004.

[15] Dogan Kesdogan, Jan Egner, and Roland Büschkes. Stop- and- go-mixes providing probabilistic anonymity in an open system. volume 1525, pages 83–98, 04 1998.

[16] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. AnoA: A framework for analyzing anonymous communication protocols. *Journal of Privacy and Confidentiality*, 7, 01 2017.

[17] Aggelos Kiayias Claudia Diaz, Harry Halpin. The Nym network, the next generation of privacy infrastructure. `https://nymtech.net/nym-whitepaper.pdf`, 2021. Last accessed 10 May 2023.

[18] Jeff Vander Stoep and Stephen Hines. Rust in the Android platform. `https://security.googleblog.com/2021/04/rust-in-android-platform.html`, 2021. Last accessed 10 May 2023.

[19] The Rust Foundation. Platform support – the rustc book. `https://doc.rust-lang.org/nightly/rustc/platform-support.html#tier-2-with-host-tools`, 2023. Last accessed 10 May 2023.

[20] statista. Mobile operating systems' market share worldwide from 1st quarter 2009 to 4th quarter 2022. `https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/`, 2022. Last accessed 10 May 2023.

[21] Mozilla. Nym. `https://github.com/mozilla/rust-android-gradle`, 2022. Last accessed 10 May 2023.

[22] Google. Configuring ART – how ART works. `https://source.android.com/docs/core/runtime/configure#how_art_works`, 2023. Last accessed 10 May 2023.

[23] Rust. Build scripts - the Cargo book. `https://doc.rust-lang.org/cargo/reference/build-scripts.html`, 2023. Last accessed 10 May 2023.

[24] Brian Smith. Building *ring* – cross compiling. `https://github.com/briansmith/ring/blob/main/BUILDING.md#cross-compiling`, 2022. Last accessed 10 May 2023.

[25] Henrik Grimler. android/ndk repository issue 1614: [bug] with ndk r23 and newer, builtin symbols cannot be found when a program is linked with libtool. `https://github.com/android/ndk/issues/1614`, 2021. Last accessed 10 May 2023.

[26] xtkoba. Comment 1369150244 on termux/termux-packages repository issue 8029: Several configure&make packages give binaries with `cannot locate symbol "__extendsftf2"`. `https://github.com/termux/termux-packages/issues/8029#issuecomment-1369150244`, 2022. Last accessed 10 May 2023.

[27] ssrlive. Comment 1096266946 on rust-lang/rust repository issue 85806: Support Android ndk versions `r23-beta3` and up. `https://github.com/rust-lang/rust/pull/85806#issuecomment-1096266946`, 2022. Last accessed 10 May 2023.

[28] Alex Pinkus. Comment 1289987284 on rust-lang/rust repository issue 102332: Update CI to use Android NDK r25b. https://github.com/rust-lang/rust/pull/102332#issuecomment-1289987284, 2022. Last accessed 10 May 2023.

[29] Alex Pinkus. rust-lang/rust repository issue 103673: Android NDK r25b changes will break developers using r22b or older. https://github.com/rust-lang/rust/issues/103673, 2022. Last accessed 10 May 2023.

[30] Kotlin. Unsigned integer types. https://kotlinlang.org/docs/unsigned-integer-types.html, 2023. Last accessed 10 May 2023.

[31] Oracle. Java Native Interface specification: 3 – JNI types and data structures – modified UTF-8 strings. https://docs.oracle.com/en/java/javase/19/docs/specs/jni/types.html#modified-utf-8-strings, 2022. Last accessed 10 May 2023.

[32] Google. Systemclock.cpp - Android Code Search. https://cs.android.com/android/platform/superproject/+/master:system/core/libutils/SystemClock.cpp;l=64, 2008. Last accessed 10 May 2023.

[33] Google. Keep the device awake – keep the CPU on. https://developer.android.com/training/scheduling/wakelock#cpu, 2023. Last accessed 10 May 2023.

[34] Google. Bound services overview – use a Messenger. https://developer.android.com/guide/components/bound-services#Messenger, 2023. Last accessed 10 May 2023.

[35] Google. Remove signing information from your build files. https://developer.android.com/studio/publish/app-signing#secure-shared-keystore, 2023. Last accessed 10 May 2023.

[36] Google. Power management – battery saver improvements. https://developer.android.com/about/versions/pie/power#battery-saver, 2021. Last accessed 10 May 2023.

[37] Google. Foreground services. https://developer.android.com/guide/components/foreground-services, 2023. Last accessed 10 May 2023.

[38] Google. Optimize for Doze and App Standby – understanding app standby. https://developer.android.com/training/monitoring-device-state/doze-standby#understand_app_standby, 2023. Last accessed 10 May 2023.

[39] Google. Services overview – types of services. https://developer.android.com/guide/components/services#Types-of-services, 2023. Last accessed 10 May 2023.

[40] Texas Instruments. INA219 data sheet, product information and support | ti.com. https://www.ti.com/product/INA219, 2014. Last accessed 10 May 2023.

[41] statista. Mobile operating systems' market share worldwide from 1st quarter 2009 to 4th quarter 2022. https://www.statista.com/statistics/1230433/average-smartphone-nand-memory-capacity-by-brand/, 2021. Last accessed 10 May 2023.

[42] Urbandroid Team. Don't kill my app! | hey android vendors, don't kill my app! `https://dontkillmyapp.com/`, 2023. Last accessed 10 May 2023.

[43] Daniel Hugenroth, Martin Kleppmann, and Alastair R. Beresford. Rollercoaster: An efficient Group-Multicast scheme for mix networks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3433–3450. USENIX Association, August 2021.

[44] George Danezis and Ian Goldberg. Sphinx: A compact and provably secure mix format. volume 2008, pages 269–282, 05 2009.

[45] Loopix. Loopix. `https://github.com/UCL-InfoSec/loopix/`, 2017. Last accessed 10 May 2023.

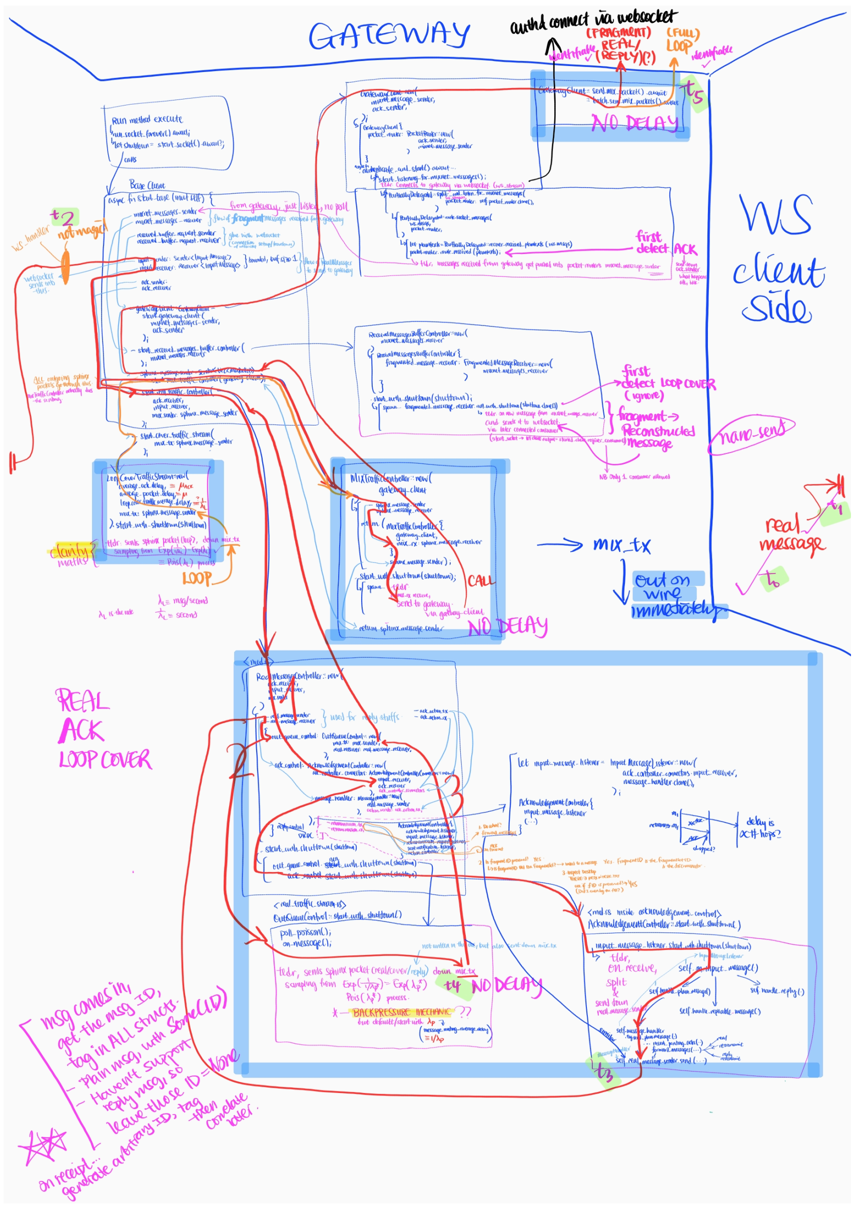
[46] Nym. Nym. `https://github.com/nymtech/nym`, 2022. Last accessed 10 May 2023.

[47] S.M. Ross. *Introduction to Probability Models*. Elsevier Science, 2006.

Appendix A

Difficulties with the Nym Codebase

Figure A.1 shows one of the diagrams I drew out to understand data flow through Nym's codebase. The codebase is highly concurrent due to liberal usage of channels for inter-thread communication. Each channel's transmitter and receiver ends are passed independently throughout the codebase, which was difficult to track.



GATEWAY
authd connect via websocket
(FRAGMENT) REAL (REPLY)(?)
(FULL) LOOP
identifiable
NO DELAY
WS client side
first detect ACK
first detect LOOP COVER (ignore)
fragment → Reconstructed message
nano-send
real message
mix_tx
out on wire immediately
NO DELAY
CALL
LOOP
clarify maths
REAL ACK LOOP COVER
NO DELAY
BACKPRESSURE MECHANIC ??
msg comes in, get the msg_ID, tag in ALL structs. Plain msg, with Some(ID). Haven't support reply msg, so leave these ID=None. tag then condense later.
on receipt... generate arbitrary ID

Figure A.1: Each box represents a Rust struct and relevant parts of its implementation. The red arrows represent the path messages take to get from my Android application ("WS client side" on the right) to the Nym mixnet ("GATEWAY" on top). The red lines represent channels.

Appendix B

Cryptographic Schemes

The schemes presented in this appendix are meant as a introductory companion guide to the formal cryptographic schemes used in Sphinx, Loopix and Nym. Loopix and Nym do not formally specify the mathematical notation; the ones presented here are based on my understanding of how they work.

B.1 Sphinx

Sphinx presents the following scheme for a single message from U to S through 3 mix nodes N_0, N_1, N_2 in an arbitrary mixnet [44] . Sphinx packets are made of a Sphinx header and payload, which are handled independently. The Sphinx header that N_i receives takes the form $(\alpha_i, \beta_i, \gamma_i)$, where $\alpha_i = g^x$ and γ_i is a message authentication code (MAC) of β_i.

The details of the blinding of the α_is (such that they become bitwise unlinkable across a hop, but still serve their purpose) and padding are elided from this presentation for simplicity.

$$U \to N_0 : \left(\underbrace{g^x}_{\alpha_0}, \underbrace{\mathrm{Enc}_{s_0}(N_1, \beta_1, \gamma_1)}_{\beta_0}, \underbrace{\mathrm{Mac}_{s_0}(\beta_0)}_{\gamma_0} \right) \Bigg\| \mathrm{Enc}_{s_0}(\mathrm{Enc}_{s_1}(\mathrm{Enc}_{s_2}(N_3, m)))$$

$$N_0 \to N_1 : \left(\underbrace{g^x}_{\alpha_1}, \underbrace{\mathrm{Enc}_{s_1}(N_2, \beta_2, \gamma_2)}_{\beta_1}, \underbrace{\mathrm{Mac}_{s_1}(\beta_1)}_{\gamma_1} \right) \Bigg\| \mathrm{Enc}_{s_1}(\mathrm{Enc}_{s_2}(N_3, m))$$

$$N_1 \to N_2 : \left(\underbrace{g^x}_{\alpha_2}, \underbrace{\mathrm{Enc}_{s_2}(*, 0, 0)}_{\beta_2}, \underbrace{\mathrm{Mac}_{s_2}(\beta_2)}_{\gamma_2} \right) \Bigg\| \mathrm{Enc}_{s_2}(N_3, m)$$

$$N_2 \to S \triangleq N_3 : m$$

where $*$ is a distinguished value in the space of mixnet addresses that indicates that the current node is the final mix node. Note that the recipient does not receive a Sphinx packet.

For each message that U wants to send to S, U performs:

1. Choose mix node addresses N_0, N_1, N_2. Obtain their public keys pk_{N_i}.
2. Package the payload, recipient address and routing information into the Sphinx packet format.

   For the header, given $g \in \mathbb{G}$ a publicly known group element, U calculates:

   - A single $x \in_R \mathbb{Z}^*_{|\mathbb{G}|}$ chosen uniformly at random

- Shared secret with each intermediate hop and recipient:

$$s_i = pk_{M_i}^x$$

where $pk_{M_i} = g^{sk_{M_i}}$ by Diffie-Hellman, so each node M_i can calculate this shared secret by raising g^x to the power of its secret key:

$$s_i = g^{x \cdot sk_{M_i}}$$

The header is then an "onion encrypted" version of the final header (received by N_2) as presented above.

Separately, the Sphinx payload is simply the raw message m, encrypted once for each $i \in \{0, \dots, 4\}$, as in onion routing.

3. Sends the Sphinx packet to P_U.

Each successive node in the mixnet peels off a layer of encryption from both the Sphinx header and Sphinx payload.

B.2 Loopix

Loopix adds the concept of a provider, a special kind of mix node. A message from U to S thus travels through 5 mix nodes $M_0, \dots, M_4$ where $M_0 \triangleq P_U$ (U's provider) and $M_4 \triangleq P_S$ (S's provider).

The following scheme conforms to the Loopix implementation [45], though not presented in their paper. The main differences with the previous section are (1) the addition of the delay parameter d_i in the header, for each mix node to know how long to hold packets for; (2) the contents of the unencrypted payload, which now includes a MAC; (3) the recipient now receives a Sphinx packet too.

$$\begin{aligned}
U \to P_U \triangleq M_0 :\ & \left(\underbrace{g^x}_{\alpha_0}, \underbrace{\mathrm{Enc}_{s_0}(M_1, \beta_1, \gamma_1, d_0)}_{\beta_0}, \underbrace{\mathrm{Mac}_{s_0}(\beta_0)}_{\gamma_0} \right) \Bigg\| \\
& \mathrm{Enc}_{s_0}(\cdots \mathrm{Enc}_{s_5}(S, m, \mathrm{Mac}_{s_5}(S, m)) \cdots) \\
\forall i \in \{1,2,3,4\}.\ M_{i-1} \to M_i :\ & \left(\underbrace{g^x}_{\alpha_i}, \underbrace{\mathrm{Enc}_{s_i}(M_{i+1}, \beta_{i+1}, \gamma_{i+1}, d_i)}_{\beta_i}, \underbrace{\mathrm{Mac}_{s_i}(\beta_i)}_{\gamma_i} \right) \Bigg\| \\
& \mathrm{Enc}_{s_i}(\cdots \mathrm{Enc}_{s_5}(S, m, \mathrm{Mac}_{s_5}(S, m)) \cdots) \\
P_S \triangleq M_4 \to S :\ & \left(\underbrace{g^x}_{\alpha_5}, \underbrace{\mathrm{Enc}_{s_5}(*, 0)}_{\beta_5}, \underbrace{\mathrm{Mac}_{s_5}(\beta_5)}_{\gamma_5} \right) \Bigg\| \\
& \mathrm{Enc}_{s_5}(S, m, \mathrm{Mac}_{s_5}(S, m))
\end{aligned}$$

where $*$ is a distinguished value in the space of Loopix addresses that indicates that the current node is the recipient S.

Note that although P_S receives a delay d_4, it is ignored in code. Therefore, delays are only done on intermediate hops.

B.3 Nym

The following scheme conforms to the Nym implementation [46], though not presented in their whitepaper. The main differences with the Loopix scheme are presented in Section 2.2.

$$U \to G_U \triangleq M_0 : \mathrm{Enc}_{\textsc{long-term}(U,G_U)}\left(\text{what } G_U \triangleq M_0 \text{ sends to } M_1\right)$$

$$\forall i \in \{1,2,3\}.\ M_{i-1} \to M_i : \left(\underbrace{g^x}_{\alpha_i}, \underbrace{\mathrm{Enc}_{s_i}(M_{i+1}, \beta_{i+1}, \gamma_{i+1}, d_i)}_{\beta_i}, \underbrace{\mathrm{Mac}_{s_i}(\beta_i)}_{\gamma_i}\right) \Big\| \ \mathrm{Enc}_{s_i}(\cdots \mathrm{Enc}_{s_4}(x' \parallel \mathrm{Enc}_{s_S}(m)) \cdots)$$

$$M_3 \to G_S \triangleq M_4 : \left(\underbrace{g^x}_{\alpha_4}, \underbrace{\mathrm{Enc}_{s_4}(S, 0)}_{\beta_4}, \underbrace{\mathrm{Mac}_{s_4}(\beta_4)}_{\gamma_4}\right) \Big\| \ \mathrm{Enc}_{s_4}(x' \parallel \mathrm{Enc}_{s_S}(m))$$

$$M_4 \triangleq G_S \to S : \mathrm{Enc}_{\textsc{long-term}(S,G_S)}\left(x' \parallel \mathrm{Enc}_{s_S}(m)\right)$$

where similar to x, x' is a ephemeral secret, that is accessible only by U, S and used to derive the shared secret s_S without conducting key exchange. The message m is also protected by MAC but via another mechanism necessary to support Single Use Reply Blocks (SURBs), details of which are outside the scope of this work.

Appendix C

Poisson Processes

C.1 Proof: Closure under Summation

Let P, L, D be the number of messages emitted per second by the independent streams mentioned in Section 2.1.3. For a process whose events occur with intervals $\sim$ Exp(λ) with rate parameter λ (units: s^{-1}), the number of events per second has distribution $\sim$ Pois(λ). Therefore, $P \sim \mathrm{Pois}(\lambda_P), L \sim \mathrm{Pois}(\lambda_L)$ and $D \sim \mathrm{Pois}(\lambda_D)$.

We show that $P + L + D \sim \mathrm{Pois}(\lambda_P + \lambda_L + \lambda_D)$:

$$\begin{aligned}
\mathbb{P}(P+L+D=n) &= \sum_{a=0}^{n}\sum_{b=0}^{n-a}\mathbb{P}(P=a \wedge L=b \wedge D=n-a-b) \\
&\quad \text{(by Law of Total Probability)} \\
&= \sum_{a=0}^{n}\sum_{b=0}^{n-a}\mathbb{P}(P=a)\cdot\mathbb{P}(L=b)\cdot\mathbb{P}(D=n-a-b) \\
&\quad \text{(by independence)} \\
&= \sum_{a=0}^{n}\sum_{b=0}^{n-a}\frac{\lambda_P^a e^{-\lambda_P}}{a!}\cdot\frac{\lambda_L^b e^{-\lambda_L}}{b!}\cdot\frac{\lambda_D^{n-a-b}e^{-\lambda_D}}{(n-a-b)!} \\
&= e^{-(\lambda_P+\lambda_L+\lambda_D)}\sum_{a=0}^{n}\frac{\lambda_P^a}{a!}\sum_{b=0}^{n-a}\frac{\lambda_L^b\lambda_D^{n-a-b}}{b!(n-a-b)!} \\
&= e^{-(\lambda_P+\lambda_L+\lambda_D)}\sum_{a=0}^{n}\frac{\lambda_P^a}{a!(n-a)!}\sum_{b=0}^{n-a}\binom{n-a}{b}\lambda_L^b\lambda_D^{n-a-b} \\
&= e^{-(\lambda_P+\lambda_L+\lambda_D)}\sum_{a=0}^{n}\frac{\lambda_P^a}{a!(n-a)!}(\lambda_L+\lambda_D)^{n-a} \\
&= \frac{e^{-(\lambda_P+\lambda_L+\lambda_D)}}{n!}\sum_{a=0}^{n}\binom{n}{a}\lambda_P^a(\lambda_L+\lambda_D)^{n-a} \\
&= \frac{(\lambda_P+\lambda_L+\lambda_D)^n e^{-(\lambda_P+\lambda_L+\lambda_D)}}{n!}
\end{aligned}$$

which is indeed the probability mass function of Pois$(\lambda_P + \lambda_L + \lambda_D)$. This is in fact an instantiation of the more general property for independently distributed Poisson distributions:

$$\sum_i \mathrm{Pois}(\lambda_i) = \mathrm{Pois}\left(\sum_i \lambda_i\right)$$

C.2 How Poisson Mixes foil Timing Analysis Attacks

The exponential distribution is memoryless. Let $D \sim \mathrm{Exp}(\mu)$ be an R.V. representing a Sphinx packet's delay. Its tail distribution function $\mathbb{P}(D > t)$ is $e^{-\mu t}$, which fully describes

$\mathrm{Exp}(\mu)$ just like the probability density function, satisfies:

$$\begin{aligned}
\mathbb{P}(\underbrace{D > s + t}_{\substack{\text{remaining} \\ \text{delay is } t \\ \text{after time } s}} \mid D > s) &= \frac{\mathbb{P}(D > s + t \wedge D > t)}{\mathbb{P}(D > s)} \\
&= \frac{\mathbb{P}(D > s + t)}{\mathbb{P}(D > s)} \\
&= \frac{e^{-\mu(s+t)}}{e^{-\mu s}} \\
&= e^{-\mu t} \\
&= \mathbb{P}(D > t)
\end{aligned}$$

This is exactly the definition of memorylessness [47]. It states that the distribution of delay D at time $t = 0$, $\mathbb{P}(D > t)$, is identical to the the distribution of the remaining delay $D - s$ at time $t = s$ given that D is at least t, $\mathbb{P}(D > s + t \mid D > s)$.

From the perspective of an attacker observing a mix, the delays of each packet is unknown and is thus represented by R.V.s $D_i \sim \mathrm{Exp}(\mu)$ for each packet i. To perform linking, the adversary must determine which packet is the next one to be forwarded. Suppose packets 1 and 2 enter the mix node, one at $t = 0$ and one at $t = s$. At $t = s$, the probability distributions of their remaining delays D_1, D_2 are given by $\mathbb{P}(D_1 > s + t)$ and $\mathbb{P}(D_2 > t)$ respectively, which by memorylessness are identical. Knowledge of the amount of time packets have already spent waiting in the mix node grants the adversary no advantage in ascertaining which packet is next to leave: in other words, once in a pool, the timing characteristics of all packets are indistinguishable.

Appendix D

Mapping between Kotlin and Rust Data Types

T_{Kotlin}	T_{C}	**Implementation of** `consume_kt_`$T_{\text{Kotlin}} : T_{\text{C}} \to T_{\text{Rust}}$	T_{Rust}
`Boolean`	`jboolean`	`1` $\mapsto$ `true`, `0` $\mapsto$ `false`	`bool`
`Byte`	`jbyte`	Identity	`i8`
`UByte`	`jbyte`	Re-interpret bits	`u8`
`Char`	`jchar`	*Not implemented*	`char`
`Short`	`jshort`	Identity	`i16`
`UShort`	`jshort`	Re-interpret bits	`u16`
`Int`	`jint`	Identity	`i32`
`UInt`	`jint`	Re-interpret bits	`u32`
`Long`	`jlong`	Identity	`i64`
`ULong`	`jlong`	Re-interpret bits	`u64`
`Float`	`jfloat`	Identity	`f32`
`Double`	`jdouble`	Identity	`f64`
`String`	`jstring`	Delegate to low-level JNI methods	`Result<String, JNIError>`
`Boolean?`	`jobject` (`java/lang/Boolean`)	Null check, then delegate to low-level JNI methods	`Result<Option<bool>, JNIError>`
`Byte?`	`jobject` (`java/lang/Byte`)	Null check, then delegate to low-level JNI methods	`Result<Option<i8>, JNIError>`
`UByte?`	`jobject` (`kotlin/UByte`)	Null check, then delegate to low-level JNI methods	`Result<Option<u8>, JNIError>`
`Char?`	`jobject`	*Not implemented*	`Result<Option<char>, JNIError>`
`Short?`	`jobject` (`java/lang/Short`)	Null check, then delegate to low-level JNI methods	`Result<Option<i16>, JNIError>`
`UShort?`	`jobject` (`kotlin/UShort`)	Null check, then delegate to low-level JNI methods	`Result<Option<u16>, JNIError>`
`Int?`	`jobject` (`java/lang/Integer`)	Null check, then delegate to low-level JNI methods	`Result<Option<i32>, JNIError>`
`UInt?`	`jobject` (`kotlin/UInt`)	Null check, then delegate to low-level JNI methods	`Result<Option<u32>, JNIError>`
`Long?`	`jobject` (`java/lang/Long`)	Null check, then delegate to low-level JNI methods	`Result<Option<i64>, JNIError>`
`ULong?`	`jobject` (`kotlin/ULong`)	Null check, then delegate to low-level JNI methods	`Result<Option<u64>, JNIError>`
`Float?`	`jobject` (`java/lang/Float`)	Null check, then delegate to low-level JNI methods	`Result<Option<f32>, JNIError>`
`Double?`	`jobject` (`java/lang/Double`)	Null check, then delegate to low-level JNI methods	`Result<Option<f64>, JNIError>`
`String?`	`jstring`	Null check, then use above `jstring` $\mapsto$ `Result<String, JNIError>`	`Result<Option<String>, JNIError>`

Table D.1: Full table of the mapping between Kotlin, C and Rust type through `jvm_kotlin_typing`'s API. This table describes all `consume_kt_`T_{Kotlin} functions as in Section 3.4.1.

T_{Rust}	**Implementation of `produce_kt_`T_{Kotlin} : $T_{\text{Rust}} \to T_{\text{C}}$**	T_{C}	T_{Kotlin}
`bool`	`true` $\mapsto$ 1, `false` $\mapsto$ 0	`jboolean`	`Boolean`
`i8`	Identity	`jbyte`	`Byte`
`u8`	Re-interpret bits	`jbyte`	`UByte`
`char`	*Not implemented*	`jchar`	`Char`
`i16`	Identity	`jshort`	`Short`
`u16`	Re-interpret bits	`jshort`	`UShort`
`i32`	Identity	`jint`	`Int`
`u32`	Re-interpret bits	`jint`	`UInt`
`i64`	Identity	`jlong`	`Long`
`u64`	Re-interpret bits	`jlong`	`ULong`
`f32`	Identity	`jfloat`	`Float`
`f64`	Identity	`jdouble`	`Double`
`String`	Delegate to `jni` crate	`jstring`	`String`
`Option<bool>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Boolean`)	`Boolean?`
`Option<i8>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Byte`)	`Byte?`
`Option<u8>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`kotlin/UByte`)	`UByte?`
`Option<char>`	*Not implemented*	`jobject`	`Char?`
`Option<i16>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Short`)	`Short?`
`Option<u16>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`kotlin/UShort`)	`UShort?`
`Option<i32>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Integer`)	`Int?`
`Option<u32>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`kotlin/UInt`)	`UInt?`
`Option<i64>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Long`)	`Long?`
`Option<u64>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`kotlin/ULong`)	`ULong?`
`Option<f32>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Float`)	`Float?`
`Option<f64>`	`None` check, then delegate to low-level JNI methods (∗)	`jobject` (`java/lang/Double`)	`Double?`
`Option<String>`	`None` check, then use above `String` $\mapsto$ `jstring`	`jstring`	`String?`

Table D.2: The `produce_kt_`T_{Kotlin} functions in `jvm_kotlin_typing`'s API, as introduced in Section 3.4.1. All rows tagged with (∗) actually return `Result<`T_{C}`, JNIError>` in rust, because the low-level JNI methods exposed by the `jni` community crate may fail.